The security protocol verifier ProVerif and its Horn clause resolution algorithm

Bruno Blanchet

Inria, Paris, France
Bruno.Blanchet@inria.fr

April 2022

Cryptographic protocols

(1)

Cryptographic protocols

- small programs designed to secure communication (various security goals)
- use cryptographic primitives (e.g. encryption, hash function, ...)

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

The symbolic model

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

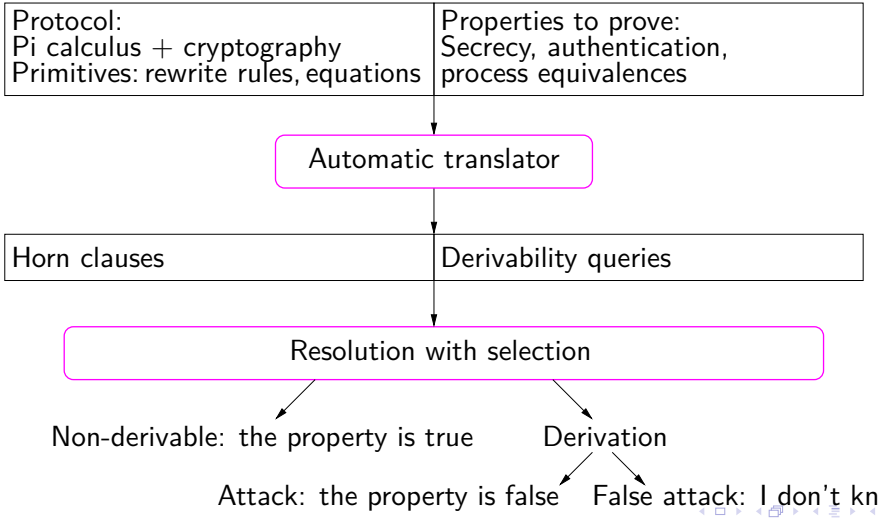
- Cryptographic primitives are blackboxes. senc
- Messages are terms on these primitives. $\mathrm{senc}(Hello, k)$
- The attacker is restricted to compute only using these primitives.
  $\Rightarrow$ perfect cryptography assumption
  - So the definitions of primitives specify what the attacker can do.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

This model makes automatic proofs relatively easy.

Features of ProVerif

- Fully automatic.
- Works for unbounded number of sessions and message space.
  - $\Rightarrow$ undecidable problem
- 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 in a few minutes.
  - Very precise: no false attack in our tests on examples of the literature for secrecy and authentication.

ProVerif, https://proverif.inria.fr/



Syntax of the process calculus

Pi calculus + cryptographic primitives

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

Constructors and destructors

Two kinds of operations:

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

Example: Shared-key encryption $\mathrm{senc}(M, N)$

fun $\mathrm{senc}(\mathrm{bitstring}, \mathrm{key}) : \mathrm{bitstring}$.

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

Example: Decryption $\mathrm{sdec}(\mathrm{senc}(m, k), k) \rightarrow m$

fun $\mathrm{sdec}(\mathrm{bitstring}, \mathrm{key}) : \mathrm{bitstring}$
reduc forall $m : \mathrm{bitstring}, k : \mathrm{key}; \mathrm{sdec}(\mathrm{senc}(m, k), k) = m$.

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

Example: The Denning-Sacco protocol (simplified)

Message 1. $A \to B: \quad \{\{k\}_{sk_A}\}_{pk_B} \quad k$ fresh
Message 2. $B \to A: \quad \{s\}_k$

$$
\begin{array}{ll}
 & \textbf{new}\ sk_A : \mathrm{sskey}; \textbf{new}\ sk_B : \mathrm{eskey}; \textbf{let}\ pk_A = \mathrm{spk}(sk_A)\ \textbf{in} \\
 & \textbf{let}\ pk_B = \mathrm{pk}(sk_B)\ \textbf{in}\ \textbf{out}(c, pk_A); \textbf{out}(c, pk_B); \\
(A) & \quad !\ \textbf{in}(c, x\_pk_B : \mathrm{epkey}); \textbf{new}\ k : \mathrm{key}; \\
 & \quad\ \ \textbf{out}(c, \mathrm{penc}(\mathrm{sign}(k, sk_A), x\_pk_B)); \\
 & \quad\ \ \textbf{in}(c, x : \mathrm{bitstring}); \textbf{let}\ s = \mathrm{sdec}(x, k)\ \textbf{in}\ 0 \\
(B) & \mid\ !\ \textbf{in}(c, y : \mathrm{bitstring}); \textbf{let}\ y' = \mathrm{pdec}(y, sk_B)\ \textbf{in} \\
 & \quad\ \ \textbf{let}\ k = \mathrm{checksign}(y', pk_A)\ \textbf{in}\ \textbf{out}(c, \mathrm{senc}(\mathrm{s}, k))
\end{array}
$$

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 att:

att(M) *means* *"the attacker may have M".*

We can model actions of the attacker 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 $f(M_1, \ldots, M_n)$
  $\mathrm{att}(x_1) \wedge \ldots \wedge \mathrm{att}(x_n) \rightarrow \mathrm{att}(f(x_1, \ldots, x_n))$

Example: Shared-key encryption $\mathrm{senc}(m, k)$

$\mathrm{att}(m) \wedge \mathrm{att}(k) \rightarrow \mathrm{att}(\mathrm{senc}(m, k))$

- Destructors $g(M_1, \ldots, M_n) \rightarrow M$
  $\mathrm{att}(M_1) \wedge \ldots \wedge \mathrm{att}(M_n) \rightarrow \mathrm{att}(M)$

Example: Shared-key decryption $\mathrm{sdec}(\mathrm{senc}(m, k), k) \rightarrow m$

$\mathrm{att}(\mathrm{senc}(m, k)) \wedge \mathrm{att}(k) \rightarrow \mathrm{att}(m)$

Coding of a protocol

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

$$\mathrm{att}(M_1) \wedge \ldots \wedge \mathrm{att}(M_n) \rightarrow \mathrm{att}(M).$$

Example

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

$$\mathrm{att}(\mathrm{penc}(\mathrm{sign}(y, sk_A), pk_B)) \rightarrow \mathrm{att}(\mathrm{senc}(\mathrm{s}, y))$$

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

Proof of secrecy

Theorem (Secrecy)

If att(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.

Example

query attacker(s).

Resolution with free selection

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

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 att(x) if possible,
- the conclusion otherwise.

Key idea: avoid resolving on facts att(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 (1)

Correspondence assertions (authentication):

If an event has been executed, then some other events must have been executed.

$$\begin{array}{ll}
 & \textbf{new}\ sk_A : \text{sskey}; \textbf{new}\ sk_B : \text{eskey}; \textbf{let}\ pk_A = \text{spk}(sk_A)\ \textbf{in} \\
 & \textbf{let}\ pk_B = \text{pk}(sk_B)\ \textbf{in}\ \textbf{out}(c, pk_A); \textbf{out}(c, pk_B); \\
(A) & \quad !\ \textbf{in}(c, x_pk_B : \text{epkey}); \textbf{new}\ k : \text{key}; \textbf{event}\ eA(pk_A, x_pk_B, k); \\
 & \quad \textbf{out}(c, \text{penc}(\text{sign}(k, sk_A), x_pk_B)); \\
 & \quad \textbf{in}(c, x : \text{bitstring}); \textbf{let}\ s = \text{sdec}(x, k)\ \textbf{in}\ 0 \\
(B) & \mid\ !\ \textbf{in}(c, y : \text{bitstring}); \textbf{let}\ y' = \text{pdec}(y, sk_B)\ \textbf{in} \\
 & \quad \textbf{let}\ k = \text{checksign}(y', pk_A)\ \textbf{in}\ \textbf{event}\ eB(pk_A, pk_B, k); \\
 & \quad \textbf{out}(c, \text{senc}(\text{s}, k))
\end{array}$$

$$\textbf{query}\ x : \text{spkey}, y : \text{epkey}, z : \text{key}; \textbf{event}(eB(x, y, z)) \Longrightarrow \textbf{event}(eA(x, y, z))$$

Other security properties (2)

Process equivalences:

- Strong secrecy: the attacker cannot distinguish when the value of the secret changes.
- diff-equivalence: Equivalence 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.

Specificities of the resolution algorithm

1. Simplification of data constructors 2. Blocking predicates	JCS'09
3. Disequations	JLAP'08 with Martín Abadi and Cédric Fournet
4. Natural numbers 5. Temporal correspondence queries 6. Precise actions 7. Axioms, Restrictions, Lemmas 8. Proofs by induction	IEEE S&P'22 with Vincent Cheval and Véronique Cortier

Simplification of data constructors

Data constructors f are constructors that come with associated projections π_i^f (e.g., tuples):

$$\pi_i^f(f(x_1, \ldots, x_n)) \rightarrow x_i$$

Clauses:

$$\mathrm{att}(x_1) \wedge \cdots \wedge \mathrm{att}(x_n) \rightarrow \mathrm{att}(f(x_1, \ldots, x_n))$$
$$\mathrm{att}(f(x_1, \ldots, x_n)) \rightarrow \mathrm{att}(x_i)$$

Hence

$$\mathrm{att}(M_1) \wedge \cdots \wedge \mathrm{att}(M_n) \leftrightarrow \mathrm{att}(f(M_1, \ldots, M_n))$$

1. We replace $\mathrm{att}(f(M_1, \ldots, M_n))$ with $\mathrm{att}(M_1) \wedge \cdots \wedge \mathrm{att}(M_n)$ in hypotheses of clauses.
2. We replace $H \rightarrow \mathrm{att}(f(M_1, \ldots, M_n))$ with $H \rightarrow \mathrm{att}(M_i)$ for all $i \leq n$.

Blocking predicates

Blocking predicates are predicates on which we do not resolve: the selection function never selects them.

They remain in the hypothesis of the clauses.

1. We can prove properties for any definition of such predicates.
2. We can prove that such predicates hold under some conditions. Particularly useful for correspondences.

Blocking predicates: example 1

```
pred check_time(time_t, time_t) [block].

query m: bitstring, ts, cur_t: time_t;
  event(Accept(m, ts, cur_t)) ==> event(Send(m, ts)) && check_time(ts, cur_t).

let Sender(skS: sskey) =
  in(time_ch, cur_time: time_t); in(c, m: bitstring);
  event Send(m, cur_time);
  out(c, (m, cur_time, sign((m,cur_time),skS))).

let Receiver(pkS: spkey) =
  in(c, (m: bitstring, ts:time_t, s:signature));
  in(time_ch, cur_time:time_t);
  if verify((m,ts),s,pkS) && check_time(ts, cur_time) then
  event Accept(m, ts, cur_time).

process new skS: sskey; let pkS = pkgen(skS) in
        !Sender(skS) | !Receiver(pkS)
```

Blocking predicates: example 2

To prove the query

$$\textbf{query}\ x : \text{spkey}, y : \text{epkey}, z : \text{key}; \textbf{event}(eB(x, y, z)) \Longrightarrow \textbf{event}(eA(x, y, z))$$

we use a blocking event predicate b-event for event eA.

1. Suppose the action corresponding to C is executed after event eA and when H holds. We generate clauses
$$\text{b-event}(eA(M_1, M_2, M_3)) \wedge H \rightarrow C$$
2. Suppose event eB is executed when H holds. We generate
$$H \rightarrow \text{event}(eB(M_1, M_2, M_3))$$
3. After resolution, if all clauses that conclude $\text{event}(eB(\ldots))$ are of the form
$$\text{b-event}(eA(M_1, M_2, M_3)) \wedge H \rightarrow \text{event}(eB(M_1, M_2, M_3))$$
then the query is proved.

Disequations

- Disequation constraints $\forall x_1, \ldots, x_n; M \neq N$.
- Fairly standard simplification steps on the disequations themselves.
- For equivalences, we have the clause:

$$\mathrm{att}'(x, y) \wedge \mathrm{att}'(x, y') \wedge y \neq y' \rightarrow \mathrm{bad} \tag{1}$$

We transform any clause other than (1) of the form

$$\mathrm{att}'(M, M') \wedge \mathrm{att}'(M, M'') \wedge H \rightarrow C$$

by unifying M' and M''.
(In case $M' \neq M''$, we can conclude bad with (1).)

Natural numbers

- Type: nat
- Allowed operations:
  - addition, subtraction between variable and natural number
  - less, less or equal, greater, greater or equal
  - predicate testing if a term is a natural number: is_nat

```
free k:key [private]. free cell:channel [private].

(* outputs natural numbers from min to max encrypted with k *)
let Q(max:nat) =
  in(cell,i:nat); out(c,senc(i,k));
  if i < max then out(cell,i+1).

process in(c, (min:nat, max:nat));
        (out(cell,min) | !Q(max))
```

Implemented by constraints is_nat(M), $\neg$is_nat(M), and $M \geq N + n$ in clauses, where n is a constant natural number, simplified using the Bellman-Ford algorithm.

Temporal correspondence queries

- Type `time` for temporal variables.
- Facts can be associated with a temporal variable: $F@i$.
- **event**$(ev)@n$ holds when event ev is executed at the n-th step of the trace.
- Can compare temporal variables:

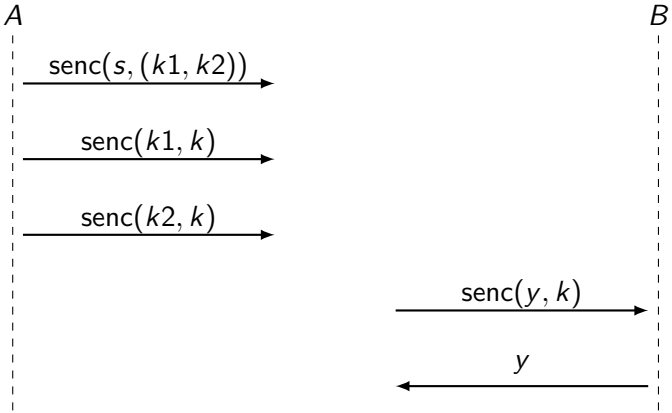
```
query i,j:time, x:bitstring;
  event(A(x))@i && event(B(x))@j ==> i < j.
```

- Integrated with nested and injective queries:

```
query i,j:time, x,y:bitstring;
  event(A(x)) && inj-event(B(x))@j ==>
  (inj-event(A(y)) ==> inj-event(B(y))@i && i <> j).
```

- Encoded as special natural number constraints $i < j$ and $i \leq j$.

Precise actions: toy example



B acts as an oracle for decryption with the key k but only one time!

Precise actions: process and clauses

Process

```
free s,k1,k2,k:bitstring [private].

let A =
  out(c,senc(s,(k1,k2)));
  out(c,senc(k1,k));
  out(c,senc(k2,k)).

let B =
  in(c,x: bitstring);
  out(c,sdec(x,k)).

process A | B
```

Clauses

– for the process

– A:

$\mathrm{att}(\mathrm{senc}(s,(k_1,k_2)))$
$\mathrm{att}(\mathrm{senc}(k_1,k))$
$\mathrm{att}(\mathrm{senc}(k_2,k))$

– B:

$\mathrm{att}(\mathrm{senc}(y,k)) \rightarrow \mathrm{att}(y)$

– for the attacker

$\mathrm{att}(x) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}(\mathrm{senc}(x,y))$
$\mathrm{att}(\mathrm{senc}(x,y)) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}(x)$
$\mathrm{att}(x) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}((x,y))$

Secrecy of s is proved when $\mathrm{att}(s)$ is not derivable from the clauses.

Precise actions: why does it fail?

Process

```
free s,k1,k2,k:bitstring [private].

let A =
  out(c,senc(s,(k1,k2)));
  out(c,senc(k1,k));
  out(c,senc(k2,k)).

let B =
  in(c,x: bitstring);
  out(c,sdec(x,k)).

process A | B
```

Clauses

– for the [illegible]

Horn clauses can be applied an arbitrary number of times for arbitrary instantiations

$\mathrm{att}(\mathrm{senc}(k_2, k))$

– B:

$\mathrm{att}(\mathrm{senc}(y, k)) \rightarrow \mathrm{att}(y)$

– for the attacker

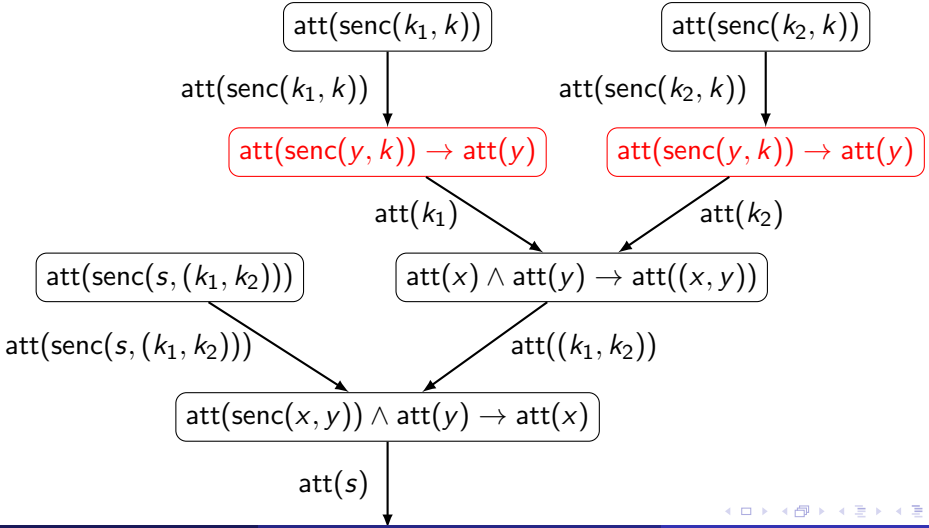
$\mathrm{att}(x) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}(\mathrm{senc}(x, y))$

$\mathrm{att}(\mathrm{senc}(x, y)) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}(x)$

$\mathrm{att}(x) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}((x, y))$

Secrecy of s is proved when $\mathrm{att}(s)$ is not derivable from the clauses.

Precise actions: why does it fail?



Precise actions: what to do?

- Add a [precise] option to the problematic input.

```
free s,k1,k2,k:bitstring [private].

let A =
  out(c,senc(s,(k1,k2)));
  out(c,senc(k1,k));
  out(c,senc(k2,k)).

let B =
  in(c,x: bitstring) [precise];
  out(c,sdec(x,k)).

process A | B
```

- Global setting: `set preciseActions = true.`
- Adding [precise] options may increase the verification time or lead to non-termination.

Restrictions, axioms, lemmas

```
restriction R_1.
...
restriction R_n.

axiom A_1.
...
axiom A_m.

lemma L_1.
...
lemma L_k.

query attacker(s).
```

Restrictions "restrict" the traces considered in axioms, lemmas, and queries.

query attacker(s) holds if no trace satisfying $R_1, \ldots, R_n$ reveals s.

1. ProVerif assumes that the axioms $A_1, \ldots, A_m$ hold.
2. ProVerif tries to prove the lemmas $L_1, \ldots, L_k$ in order, using all axioms and previously proved lemmas.
3. ProVerif tries to prove the query **query attacker**(s) using all axioms and all lemmas.

Implementing precise actions

Option [precise] is encoded as an axiom internally.

```
let B =
  in(c,x: bitstring) [precise];
  out(c,sdec(x,k)).
```

encoded as

```
event Precise(occurrence,bitstring).

axiom occ:occurrence, x1,x2:bitstring;
  event(Precise(occ,x1)) && event(Precise(occ,x2)) ==> x1 = x2.

let B = in(c,x:bitstring);
        new occ[]:occurrence;
        event Precise(occ,x);
        out(c,sdec(x,k)).
```

Using restrictions, axioms, and lemmas (simplified)

Consider a lemma (or restriction or axiom) $\bigwedge_i F_i \Longrightarrow \bigvee_j \phi_j$.

$$\frac{H \rightarrow C \qquad \text{for all } i, F_i\sigma \in H \text{ or } F_i\sigma = C}{H \wedge \phi_j\sigma \rightarrow C}$$

If for all i, $F_i\sigma \in H$ or $F_i\sigma = C$, then the hypothesis of the lemma holds, so the conclusion of the lemma holds. We add it to the hypothesis of the clause, generating clauses $H \wedge \phi_j\sigma \rightarrow C$ for all j.

Example

Axiom $\mathbf{event}(Precise(occ, x_1)) \wedge \mathbf{event}(Precise(occ, x_2)) \Longrightarrow x_1 = x_2$.
$\mathbf{event}(Precise(occ, \mathrm{senc}(k_1, k))) \wedge \mathbf{event}(Precise(occ, \mathrm{senc}(k_2, k))) \rightarrow \mathrm{att}(s)$
transformed into
$\mathbf{event}(Precise(occ, \mathrm{senc}(k_1, k))) \wedge \mathbf{event}(Precise(occ, \mathrm{senc}(k_2, k))) \wedge$
$\quad \mathrm{senc}(k_1, k) = \mathrm{senc}(k_2, k) \rightarrow \mathrm{att}(s)$
Removed.

Proofs by induction

- In order to prove a query, use that query itself as lemma on a strict prefix of the trace, by induction on the length of the trace.
- In a clause $H \rightarrow C$, H happens strictly before C.
- Consider the inductive lemma $\bigwedge_i F_i \Longrightarrow \bigvee_j \phi_j$.
  $\bigvee_j \phi_j$ holds before or at the same time as the latest F_i.

$$\frac{H \rightarrow C \qquad \text{for all } i, F_i\sigma \in H}{H \wedge \phi_j\sigma \rightarrow C}$$

  If for all i, $F_i\sigma \in H$, then the hypothesis of the lemma holds strictly before C, so the conclusion of the lemma holds strictly before C. We add it to the hypothesis of the clause, generating clauses $H \wedge \phi_j\sigma \rightarrow C$ for all j.
- Also works for a group of queries: proofs by mutual induction.

Proofs by induction: example

```
free cell:channel [private].

query x:nat;
mess(cell,x)==>is_nat(x).

let Q =
  in(cell,i:nat);
  out(c,senc(i,k));
  out(cell,i+1).

process out(cell,0) | !Q
```

Clauses:

$\mathrm{mess}(cell, 0)$

$\mathrm{mess}(cell, i) \rightarrow \mathrm{mess}(cell, i+1)$

ProVerif stops resolving on $\mathrm{mess}(cell, i)$ because it would lead to an infinite loop.

The attacker is untyped: a priori, i may not be a natural number.

The proof fails.

Proofs by induction: example solved

```
free cell:channel [private].

set nounifIgnoreAFewTimes = auto.

query x:nat;
mess(cell,x)==>is_nat(x) [induction].

let Q =
  in(cell,i:nat);
  out(c,senc(i,k));
  out(cell,i+1).

process out(cell,0) | !Q
```

Clauses:

$\mathrm{mess}(cell, 0)$

$\mathrm{mess}(cell, i) \rightarrow \mathrm{mess}(cell, i+1)$

Lemma $\mathrm{mess}(cell, x) \Longrightarrow \mathrm{is\_nat}(x)$
transforms
$\mathrm{mess}(cell, i) \rightarrow \mathrm{mess}(cell, i+1)$
into
$\mathrm{mess}(cell, i) \wedge \mathrm{is\_nat}(i) \rightarrow \mathrm{mess}(cell, i+1)$

nounifIgnoreAFewTimes allows resolution on $\mathrm{mess}(cell, i)$ once during verification.

The proof now succeeds.

Expressivity results

P Precise actions

I `set nounifIgnoreAFewTimes = auto.`

R `set removeEventsForLemma = true.`
Remove events used only for lemmas, when they become useless.

$\mathbb{N}$ Natural numbers

A Axioms, Lemmas

Expressivity results

Published protocols

Protocol	Q	O	#	N	P	I	R	ℕ	A
PCV Otway-Rees	eq	✗	1	✓	◆				
PCV Needham-Schreder	inj	✗	6	✓	◆	◆			
			3	ϟ					
PCV Denning-Sacco	inj	✗	1	ϟ					
JFK	cor	✗	2	ϟ		◆			
	inj		2	✓					
Arinc823	cor	✗	6	ϟ				◆	
Helios-norevote	eq	✗	4	✓	◆				
Signal	cor	✗	2	ϟ					
TLS12-TLS13-draft18	cor	✗	1	ϟ					

Unpublished protocols

Protocol	Q	O	#	N	P	I	R	ℕ	A
QBC_4qbits	cor	✗	1	✓	◆				
			1	ϟ					
Voting-draft	eq	✗	1	✓	◆				
LAK-simplified	cor	🕐	1	✓		◆			
PACE_v3-sequence	cor	✗	1	✓	◆			◆	◆
			3	ϟ					
DP-3T-simpl-draft	cor	🕐	1	✓	◆	◆	◆	◆	◆
			2	ϟ					
student1	cor	🕐	2	✓	◆		◆		
	inj		1	ϟ					
student2	inj	✗	1	✓					
student3	cor	🕐	1	ϟ	◆		◆		
student4	cor	✗	2	ϟ	◆		◆		
student5	cor	🕐	1	ϟ					

Improved efficiency

A Subsumption

B Translation of processes into clauses

C Resolution

D Global redundancy

E Pre-treatment of processes

A Subsumption

$H \to C$ subsumes $H' \to C'$ when $C\sigma = C'$ and $H\sigma \subseteq H'$.

Every time a clause is generated by resolution,

- check if it is not subsumed by an existing clause
- remove all existing clauses that are subsumed by this new clause

More than 80% of total execution time!

Idea [Schulz13]: Feature vertex indexing

A feature is a function f on clauses such that
$H \to C$ subsumes $H' \to C'$ implies $f(H \to C) \leq f(H' \to C')$

Clauses are organized in a trie indexed by feature values.

C Resolution

Resolution: One clause against many!

The selection function guarantees that always the same fact of a clause will be used.

Clauses are organized in a trie indexed by the symbol functions of their selected fact (depth first exploration)
[Substitution tree indexing techniques]

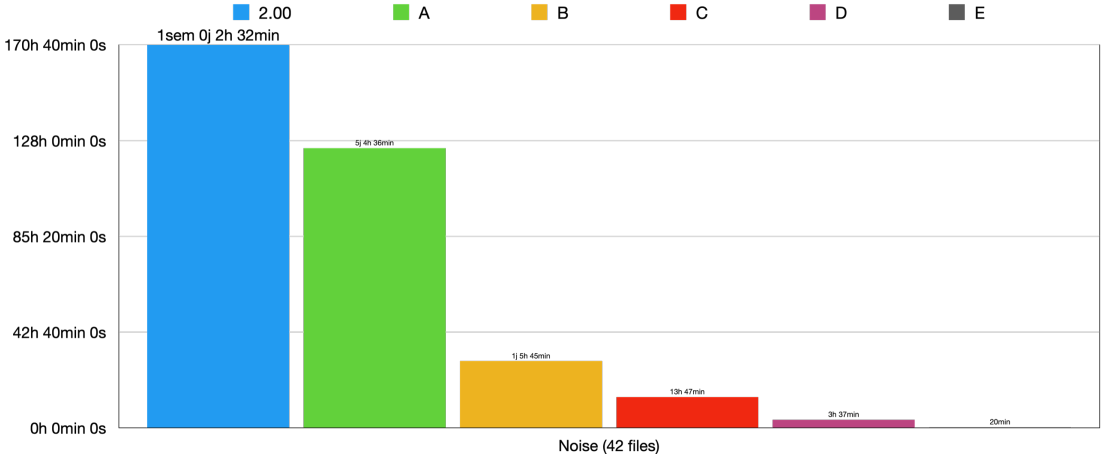
Advantage:

- Fewer unifications
- We know quickly with which clauses we can perform resolution

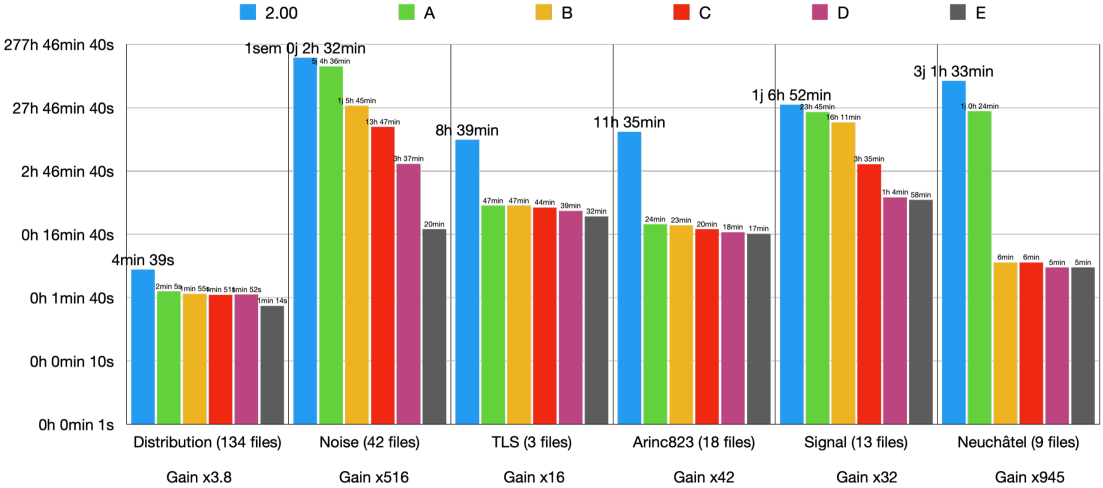
Improved efficiency

- ProVerif 2.00
- A Subsumption
- B Translation of processes into clauses
- C Resolution
- D Global redundancy
- E Pre-treatment of processes

Time gain (linear scale)



Time gain (log scale)



Conclusion

- Horn clauses are a very powerful representation.
- We can implement tricky optimizations and features by tuning the resolution algorithm.
- Huge recent improvements in ProVerif, see the paper with Vincent Cheval and Véronique Cortier at IEEE Security and Privacy 2022
  `https://bblanche.gitlabpages.inria.fr/publications/BlanchetEtAlSP22.html`