The security protocol verifier ProVerif and its Horn clause resolution algorithm

Bruno Blanchet

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

April 2022
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 or “Dolev-Yao model” is due to Needham and Schroeder (1978) and Dolev and Yao (1983).

- Cryptographic primitives are blackboxes.
- Messages are terms on these primitives.
- The attacker is restricted to compute only using these primitives.  
  ⇒ 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/](https://proverif.inria.fr/)

**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

\[ M, N ::= \]
\[ x, y, z, \ldots \]
\[ a, b, c, s, \ldots \]
\[ f(M_1, \ldots, M_n) \]

\[ P, Q ::= \]
\[ \text{out}(M, N); P \]
\[ \text{in}(M, x : T); P \]
\[ 0 \]
\[ P \parallel Q \]
\[ !P \]
\[ \text{new} \ a : T; P \]
\[ \text{let} \ x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q \]
\[ \text{if } M = N \text{ then } P \text{ else } Q \]
Constructors and destructors

Two kinds of operations:

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

**Example:** Shared-key encryption $senc(M, N)$

```latex
fun senc(bitstring, key) : bitstring.
```

- **Destructors** $g$ manipulate terms: $\text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q$

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

**Example:** Decryption $sdec(senc(m, k), k) \rightarrow m$

```latex
fun sdec(bitstring, key) : bitstring
reduc forall m : bitstring, k : key; sdec(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. $A \rightarrow B: \{\{k\}_{sk_A}\}_{pk_B}$ $k$ fresh
Message 2. $B \rightarrow A: \{s\}_k$

\[
\text{new } sk_A : \text{sskey}; \text{new } sk_B : \text{eskey}; \text{let } pk_A = \text{spk}(sk_A) \text{ in}
\]
let $pk_B = \text{pk}(sk_B)$ in out$(c, pk_A); \text{out}(c, pk_B)$;

(A) $! \text{in}(c, x_{pk_B} : \text{epkey}); \text{new } k : \text{key};$
\[
\text{out}(c, \text{penc}(\text{sign}(k, sk_A), x_{pk_B}));
\text{in}(c, x : \text{bitstring}); \text{let } s = \text{sdec}(x, k) \text{ in } 0
\]

(B) $! \text{in}(c, y : \text{bitstring}); \text{let } y' = \text{pdec}(y, sk_B) \text{ in}$
\[
\text{let } k = \text{checksign}(y', pk_A) \text{ in out}(c, \text{senc}(s, k))
\]
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 \textit{att}:

\[
\text{att}(M) \quad \text{means} \quad \text{“the attacker may have } M \text{”}.
\]

We can model actions of the attacker and of the protocol participants thanks to this predicate. Processes are \textit{automatically translated} into Horn clauses (joint work with Martín Abadi).
Coding of primitives

- **Constructors** \( f(M_1, \ldots, M_n) \)
  
  \[
  \text{att}(x_1) \land \ldots \land \text{att}(x_n) \rightarrow \text{att}(f(x_1, \ldots, x_n))
  \]

  **Example:** Shared-key encryption \( senc(m, k) \)
  
  \[
  \text{att}(m) \land \text{att}(k) \rightarrow \text{att}(senc(m, k))
  \]

- **Destructors** \( g(M_1, \ldots, M_n) \rightarrow M \)

  \[
  \text{att}(M_1) \land \ldots \land \text{att}(M_n) \rightarrow \text{att}(M)
  \]

  **Example:** Shared-key decryption \( sdec(senc(m, k), k) \rightarrow m \)
  
  \[
  \text{att}(senc(m, k)) \land \text{att}(k) \rightarrow \text{att}(m)
  \]
Coding of a protocol

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

$$\text{att}(M_1) \land \ldots \land \text{att}(M_n) \rightarrow \text{att}(M).$$

**Example**

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

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

The attacker sends penc(sign($y, sk_A$), $pk_B$) to $B$, and intercepts his reply senc($s, y$).
Proof of secrecy

**Theorem (Secrecy)**

*If* $\text{att}(M)$ *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

\[ R = H \rightarrow F \quad R' = F'_1 \land H' \rightarrow F' \]

\[ H\sigma \land H'\sigma \rightarrow F'\sigma \]

where \( \sigma \) 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 \( \text{att}(x) \) if possible,
- the conclusion otherwise.

Key idea: avoid resolving on facts \( \text{att}(x) \).

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.

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

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

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Paper/Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Simplification of data constructors</td>
<td>JCS’09</td>
</tr>
<tr>
<td>2</td>
<td>Blocking predicates</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Disequations</td>
<td>JLAP’08 with Martín Abadi and Cédric Fournet</td>
</tr>
<tr>
<td>4</td>
<td>Natural numbers</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Temporal correspondence queries</td>
<td>IEEE S&amp;P’22 with Vincent Cheval and Véronique Cortier</td>
</tr>
<tr>
<td>6</td>
<td>Precise actions</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Axioms, Restrictions, Lemmas</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Proofs by induction</td>
<td></td>
</tr>
</tbody>
</table>
Simplification of data constructors

Data constructors $f$ are constructors that come with associated projections $\pi^f_i$ (e.g., tuples):

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

Clauses:

$$\text{att}(x_1) \land \cdots \land \text{att}(x_n) \rightarrow \text{att}(f(x_1, \ldots, x_n))$$
$$\text{att}(f(x_1, \ldots, x_n)) \rightarrow \text{att}(x_i)$$

Hence

$$\text{att}(M_1) \land \cdots \land \text{att}(M_n) \leftrightarrow \text{att}(f(M_1, \ldots, M_n))$$

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

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

query m: bitstring, ts, cur_t: time_t;
  event (Accept(m, ts, cur_t)) \implies event (Send(m, ts)) \land\land 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) \land\land 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)
```

Bruno Blanchet (Inria)
Blocking predicates: example 2

To prove the query

\[
\text{query } x : \text{spkey}, y : \text{epkey}, z : \text{key}; \text{event}(eB(x, y, z)) \implies \text{event}(eA(x, y, z))
\]

we use a blocking event predicate \( b\text{-event} \) for event \( eA \).

1. Suppose the action corresponding to \( C \) is executed after event \( eA \) and when \( H \) holds. We generate clauses

\[
b\text{-event}(eA(M_1, M_2, M_3)) \land H \implies C
\]

2. Suppose event \( eB \) is executed when \( H \) holds. We generate

\[
H \implies \text{event}(eB(M_1, M_2, M_3))
\]

3. After resolution, if all clauses that conclude \( \text{event}(eB(\ldots)) \) are of the form

\[
b\text{-event}(eA(M_1, M_2, M_3)) \land H \implies \text{event}(eB(M_1, M_2, M_3))
\]

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:

$$\text{att}'(x, y) \land \text{att}'(x, y') \land y \neq y' \rightarrow \text{bad}$$

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

$$\text{att}'(M, M') \land \text{att}'(M, M'') \land H \rightarrow C$$

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

```plaintext
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), ¬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: $F@i$.
- $\text{event}(ev)@n$ holds when event $ev$ is executed at the $n$-th step of the trace.
- Can compare temporal variables:

\[
\text{query } i,j : \text{time}, x : \text{bitstring}; \\
\quad \text{event}(A(x))@i \land \text{event}(B(x))@j \implies i < j.
\]

- Integrated with nested and injective queries:

\[
\text{query } i,j : \text{time}, x,y : \text{bitstring}; \\
\quad \text{event}(A(x)) \land \text{inj-event}(B(x))@j \implies \\
\quad \text{(inj-event}(A(y)) \implies \text{inj-event}(B(y))@i \land i \not< j).
\]

- Encoded as special natural number constraints $i < j$ and $i \leq j$. 
Precise actions: toy example

\[ A \]
\[ \text{senc}(s, (k1, k2)) \]
\[ \text{senc}(k1, k) \]
\[ \text{senc}(k2, k) \]

\[ B \]
\[ \text{senc}(y, k) \]
\[ y \]

\textit{B acts as an oracle for decryption with the key} \( k \) \textit{but only one time!}
Precise actions: process and clauses

Process

```plaintext
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:
    att(senc(s, (k_1, k_2)))
    att(senc(k_1, k))
    att(senc(k_2, k))
  – B:
    att(senc(y, k)) → att(y)

– for the attacker
  att(x) ∧ att(y) → att(senc(x, y))
  att(senc(x, y)) ∧ att(y) → att(x)
  att(x) ∧ att(y) → att((x, y))

Secrecy of s is proved when att(s) is not derivable from the clauses.
Precise actions: why does it fail?

Process

```plaintext
free s, k_1, k_2, k: bitstring [private].

let A =
  out(c, senc(s, (k_1, k_2)));
  out(c, senc(k_1, k));
  out(c, senc(k_2, k)).

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

process A | B
```

Clauses

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

- for the process
  \[ \text{att(senc(k_2, k))} \]
  \[ \text{att(senc(y, k)) \rightarrow att(y)} \]

- for the attacker
  \[ \text{att(x) \land att(y) \rightarrow att(senc(x, y))} \]
  \[ \text{att(senc(x, y)) \land att(y) \rightarrow att(x)} \]
  \[ \text{att(x) \land att(y) \rightarrow att((x, y))} \]

Secrecy of \( s \) is proved when \( \text{att}(s) \) is not derivable from the clauses.
Precise actions: why does it fail?

\[
\begin{align*}
&\text{att}(\text{senc}(k_1, k)) \\
\quad &\text{att}(\text{senc}(k_1, k)) \rightarrow \text{att}(y) \\
\quad &\text{att}(k_1) \\
&\text{att}(\text{senc}(s, (k_1, k_2))) \\
\quad &\text{att}(\text{senc}(s, (k_1, k_2))) \\
\quad &\text{att}((k_1, k_2)) \\
&\text{att}(\text{senc}(x, y)) \land \text{att}(y) \rightarrow \text{att}(x) \\
\quad &\text{att}(s) \\
&\text{att}(\text{senc}(k_2, k)) \\
\quad &\text{att}(\text{senc}(k_2, k)) \rightarrow \text{att}(y) \\
\quad &\text{att}(k_2) \\
&\text{att}(\text{senc}(x, y)) \land \text{att}(y) \rightarrow \text{att}(x) \\
\quad &\text{att}(s) \\
\end{align*}
\]
Precise actions: what to do?

- Add a \([\text{precise}]\) option to the problematic input.

\[
\begin{align*}
\text{free } s, k1, k2, k : \text{bitstring} & \ [\text{private}] . \\
\text{let } A = \\
& \begin{align*}
& \text{out}(c, \text{senc}(s, (k1, k2))) ; \\
& \text{out}(c, \text{senc}(k1, k)) ; \\
& \text{out}(c, \text{senc}(k2, k)) .
\end{align*} \\
\text{let } B = \\
& \begin{align*}
& \text{in}(c, x : \text{bitstring}) \ [\text{precise}] ; \\
& \text{out}(c, \text{sdec}(x, k)) .
\end{align*} \\
\text{process } A \ | \ B
\end{align*}
\]

- Global setting: set \text{preciseActions} = \text{true}.
- Adding \([\text{precise}]\) options may increase the verification time or lead to non-termination.
Restrictions, axioms, lemmas

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.

\[
\text{let } B = \\begin{align*}
& \text{in}(c, x: \text{bitstring}) \text{[precise]}; \\
& \text{out}(c, \text{sdec}(x, k)).
\end{align*}
\]

\[
\text{event } \text{Precise}(\text{occurrence, bitstring}).
\]

\[
\text{axiom } \text{occ: occurrence, x1, x2: bitstring}; \\
\text{event}(\text{Precise}(\text{occ, x1})) \&\& \text{event}(\text{Precise}(\text{occ, x2})) \implies x1 = x2.
\]

\[
\text{let } B = \text{in}(c, x: \text{bitstring}); \\
\text{new } \text{occ[]} : \text{occurrence}; \\
\text{event } \text{Precise}(\text{occ, x}); \\
\text{out}(c, \text{sdec}(x, k)).
\]
Using restrictions, axioms, and lemmas (simplified)

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

$$
H \implies C \quad \text{for all } i, F_i\sigma \in H \text{ or } F_i\sigma = C
$$

$$
H \land \phi_j\sigma \implies C
$$

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 $H \land \phi_j\sigma \implies C$ for all $j$.

**Example**

Axiom $\text{event}(\text{Precise}(\text{occ}, x_1)) \land \text{event}(\text{Precise}(\text{occ}, x_2)) \implies x_1 = x_2$.

$\text{event}(\text{Precise}(\text{occ}, \text{senc}(k_1, k))) \land \text{event}(\text{Precise}(\text{occ}, \text{senc}(k_2, k))) \implies \text{att}(s)$

transformed into

$\text{event}(\text{Precise}(\text{occ}, \text{senc}(k_1, k))) \land \text{event}(\text{Precise}(\text{occ}, \text{senc}(k_2, k))) \land$

$senc(k_1, k) = senc(k_2, k) \implies \text{att}(s)$

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 $\bigwedge_i F_i \implies \bigvee_j \phi_j$.
  $\bigvee_j \phi_j$ holds before or at the same time as the latest $F_i$.

$$
\begin{align*}
H \rightarrow C & \quad \text{for all } i, F_i\sigma \in H \\
H \land \phi_j\sigma & \rightarrow C
\end{align*}
$$

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 $H \land \phi_j\sigma \rightarrow C$ 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:
- `mess(cell,0)`
- `mess(cell,i) -> mess(cell,i+1)`

ProVerif stops resolving on `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

```plaintext
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:
mess(cell, 0)
mess(cell, i) => mess(cell, i + 1)

Lemma mess(cell, x) => is_nat(x) transforms
mess(cell, i) => mess(cell, i + 1) into
mess(cell, i) ∧ is_nat(i) => mess(cell, i+1)

nounifIgnoreAFewTimes allows resolution on mess(cell, i) once during verification.

The proof now succeeds.
Expressivity results

**P** Precise actions

I set nounifyIgnoreAFewTimes = auto.

**R** set removeEventsForLemma = true.

Remove events used only for lemmas, when they become useless.

**N** Natural numbers

**A** Axioms, Lemmas
### Expressivity results

#### Published protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Q</th>
<th>O</th>
<th>#</th>
<th>N</th>
<th>P</th>
<th>I</th>
<th>R</th>
<th>N</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCV Otway-Rees</td>
<td>eq</td>
<td>×</td>
<td>1</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCV Needham-Schreder</td>
<td>inj</td>
<td>×</td>
<td>6</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCV Denning-Sacco</td>
<td>inj</td>
<td>×</td>
<td>1</td>
<td>▌</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JFK</td>
<td>cor</td>
<td>×</td>
<td>2</td>
<td>▌</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arinc823</td>
<td>cor</td>
<td>×</td>
<td>6</td>
<td>▌</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Helios-norevote</td>
<td>eq</td>
<td>×</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal</td>
<td>cor</td>
<td>×</td>
<td>2</td>
<td>▌</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLS12-TLS13-draft18</td>
<td>cor</td>
<td>×</td>
<td>1</td>
<td>▌</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Unpublished protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Q</th>
<th>O</th>
<th>#</th>
<th>N</th>
<th>P</th>
<th>I</th>
<th>R</th>
<th>N</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>QBC_4qbits</td>
<td>cor</td>
<td>×</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voting-draft</td>
<td>eq</td>
<td>×</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAK-simplified</td>
<td>cor</td>
<td>×</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PACE_v3-sequence</td>
<td>cor</td>
<td>×</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DP-3T-simpl-draft</td>
<td>cor</td>
<td>×</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>student1</td>
<td>cor</td>
<td>×</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>student2</td>
<td>inj</td>
<td>×</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>student3</td>
<td>cor</td>
<td>×</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>student4</td>
<td>cor</td>
<td>×</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>student5</td>
<td>cor</td>
<td>×</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improved efficiency

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

$H \rightarrow C$ subsumes $H' \rightarrow C'$ 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

$H \rightarrow C$ subsumes $H' \rightarrow C'$ implies $f(H \rightarrow C) \leq f(H' \rightarrow C')$

Clauses are organized in a trie indexed by feature values.
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)**

- **Distribution (134 files):**
  - Gain x3.8
  - Time: 277h 46min 40s

- **Noise (42 files):**
  - Gain x516
  - Time: 1h 20min

- **TLS (3 files):**
  - Gain x16
  - Time: 8h 39min

- **Arinc823 (18 files):**
  - Gain x42
  - Time: 11h 35min

- **Signal (13 files):**
  - Gain x32
  - Time: 3h 52min

- **Neuchâtel (9 files):**
  - Gain x945
  - Time: 3h 33min
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