Symbolic Verification of Cryptographic Protocols
Unbounded Process Verification with Proverif

David Baelde

LSV, ENS Paris-Saclay & Prosecco, Inria Paris

2017
**Proverif**

Protocol verifier developed by Bruno Blanchet at Inria Paris since 2000
- Analysis in formal model: secrecy, correspondences, equivalences, etc.
- Based on applied pi-calculus, Horn-clause abstraction and resolution
- The method is approximate but supports unbounded processes

Highly successful, works for most protocols including industrial ones: certified email, secure filesystem, Signal messaging, TLS draft, avionic protocols, etc.

**These lectures**
- Theory and practice of Proverif
- Secrecy, correspondences, equivalences
As usual in the formal model, messages are represented by terms

- built using constructor symbols from $f \in \Sigma_c$
- quotiented by an equational theory $E$;
- notation: $M \in \mathcal{M} = \mathcal{T}(\Sigma_c, \mathcal{N})$.

In Proverif, computations are also modeled explicitly

- terms may also feature destructor symbols $g \in \Sigma_d$;
- semantics given by reduction rules $g(M_1, \ldots, M_n) \rightarrow M$;
- yields partial computation relation $\downarrow$ over $\mathcal{T}(\Sigma, \mathcal{N}) \times \mathcal{M}$.

**Intuition:**

- use constructors for total functions,
- destructors when failure is possible/observable.
Example primitives

Symmetric encryption

type key.
fun enc(bitstring,key):bitstring.
reduc forall m:bitstring, k:key;
    dec(enc(m,k),k) = m.

Block cipher

type key.
fun enc(bitstring,key):bitstring.
fun dec(bitstring,key):bitstring.
equation forall m:bitstring, k:key; dec(enc(m,k),k) = m.
equation forall m:bitstring, k:key; enc(dec(m,k),k) = m.

Exercise: how would you model signatures?
Processes

Similar to the one(s) seen before, with a few key differences:

- let construct for evaluating computations (destructors);
- variables are typed (more on that later);
- private channels, phases, tables, events, etc.

Concrete syntax

\[
P, Q ::= 0 \mid (P \mid Q) \mid !P \mid \text{new } n:t;P \\
| \text{in}(c, x:t);P \mid \text{out}(c, u);P \\
| \text{if } u = v \text{ then } P \text{ else } Q \\
| \text{let } x = g(u_1, \ldots, u_N) \text{ in } P \text{ else } Q
\]

where \( u, v \) stand for constructor terms.

Reference for more details:

First examples

File structure

- **Declarations**: types, constructors, destructors, public and private data, processes...
- **Queries**, for now only secrecy: query attacker(s).
- **System specification**: the process/scenario to be analyzed.

**Demo**: `hello.pv` (basic file structure and use).

**Demo**: `types.pv` (on the role of types).
How does it work?

Horn clause modeling

Encode the system as a set of Horn clauses $C$:

- attacker’s abilities, e.g. constructor $f$ yields
  \[ \forall M_1, \ldots, M_n. \ (\wedge_i \text{attacker}(M_i)) \Rightarrow \text{attacker}(f(M_1, \ldots, M_n)). \]
- protocol behaviour, e.g. $\text{in}(c, x).\text{out}(c, \text{senc}(x, sk))$ yields
  \[ \forall M. \text{attacker}(M) \Rightarrow \text{attacker}(\text{senc}(M, sk)). \]

Clauses over-approximate behaviours, $C \not\models \text{attacker}(s)$ implies secrecy.

Automated reasoning

Entailment is **undecidable** for first-order Horn clauses but **resolution** (with strategies) provides practical **semi-decision algorithms**.

Proverif’s possible outcomes:

- may not terminate, may terminate with real or false attack;
- when it declares a protocol secure, it really is.
Attacker’s clauses (communication)

Predicates

Only two predicates (for now):

- \texttt{attacker}(M): attacker may know M
- \texttt{mess}(M, N): message N may be available on channel M

Variables range over messages; destructors not part of the logical language.

Communication

Send and receive on known channels:

\[
\forall M, N. \ \texttt{attacker}(M) \land \texttt{attacker}(N) \Rightarrow \texttt{mess}(M, N)
\]

\[
\forall M, N. \ \texttt{mess}(M, N) \land \texttt{attacker}(M) \Rightarrow \texttt{attacker}(N)
\]
Attacker’s clauses (deduction)

Constructors

For each $f \in \Sigma_c$ of arity $n$:

$$\forall M_1, \ldots, M_n. \ (\land_i \text{attacker}(M_i)) \Rightarrow \text{attacker}(f(M_1, \ldots, M_n))$$

Similar clauses are generated for public constants and new names.

Destructors

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

$$\forall M_1, \ldots, M_n. \ (\land_i \text{attacker}(M_i)) \Rightarrow \text{attacker}(M)$$

Equations

Proverif attempts to turn them to rewrite rules, treated like destructors.

For instance $\text{senc}(\text{sdec}(x, k), k) = x$ yields

$$\forall M, N. \ \text{attacker}(\text{sdec}(M, N)) \land \text{attacker}(N) \Rightarrow \text{attacker}(M).$$

Demo: set verboseClauses = short/explained.
Protocol clauses (informal)

Outputs

For each output, generate clauses:

- with all surrounding inputs as hypotheses;
- considering all cases for conditionals and evaluations.

Example:

\[
\text{in}(c, x).\text{in}(c, y).\text{if } y = n \text{ then let } z = \text{sdec}(x, k) \text{ in out}(c, \text{senc}(\langle z, n \rangle, k))
\]

yields the following clause (assuming that \( c \) is public)

\[
\forall M. \text{attacker}(\text{senc}(M, k)) \land \text{attacker}(n) \Rightarrow \text{attacker}(\text{senc}(\langle M, n \rangle, k))
\]
Protocol clauses (informal)

Outputs

For each output, generate clauses:
- with all surrounding inputs as hypotheses;
- considering all cases for conditionals and evaluations.

Example:
\[ \text{in}(c, x).\text{in}(c, y).\text{if } y = n \text{ then let } z = \text{sdec}(x, k) \text{ in } \text{out}(c, \text{senc}(\langle z, n \rangle, k)) \]
yields the following clause (assuming that \( c \) is public)
\[ \forall M. \text{attacker}(\text{senc}(M, k)) \land \text{attacker}(n) \Rightarrow \text{attacker}(\text{senc}(\langle M, n \rangle, k)) \]

Replication

Replication is ignored, as clauses can already be re-used in deduction.
Protocol clauses (informal)

Outputs

For each output, generate clauses:
- with all surrounding inputs as hypotheses;
- considering all cases for conditionals and evaluations.

Example:
\[ \text{in}(c, x). \text{in}(c, y). \text{if } y = n \text{ then let } z = \text{sdec}(x, k) \text{ in out}(c, \text{senc}([z, n], k)) \]
yields the following clause (assuming that \( c \) is public)
\[
\forall M. \text{attacker}(\text{senc}(M, k)) \land \text{attacker}(n) \Rightarrow \text{attacker}(\text{senc}([M, n], k))
\]

Replication

Replication is ignored, as clauses can already be re-used in deduction.
For Proverif $P$ is the same as $!P$.
More generally $Q = C[P]$ is the same as $Q' = C[!P]$.

**Exercise**

Find $Q = C[P]$ and $Q' = c[!P]$ such that
- $Q$ ensures the secrecy of some value;
- $Q'$ does not.

Analyze $Q$ in Proverif; what happens?
For Proverif $P$ is the same as $!P$.
More generally $Q = C[P]$ is the same as $Q' = C[!P]$.

**Exercise**

Find $Q = C[P]$ and $Q' = c[!P]$ such that
- $Q$ ensures the secrecy of some value;
- $Q'$ does not.

Analyze $Q$ in Proverif; what happens?

A possible solution: `repeat.pv`. 
Protocol clauses (informal)

Nonces
Treated as (private) constructors taking surrounding inputs as argument.

For example, new $a$. in($c, x$).new $b$.in($c, y$).out($c, u(x, y, a, b)$) yields
$\forall M, N. \text{attacker}(M) \land \text{attacker}(N) \Rightarrow \text{attacker}(u(M, N, a[], b[M]))$.

Exercise
In our process semantics, secrecy is not affected by the exchange of new and in operations. Find $Q$ and $Q'$ related by such exchanges such that
- both ensure the secrecy of some value;
- Proverif only proves it for $Q$. 

David Baelde (ENS Saclay & Inria Paris)
Nonces

Treated as (private) constructors taking surrounding inputs as argument.

For example, new a. in(c, x).new b.in(c, y).out(c, u(x, y, a, b)) yields
\[ \forall M, N. \text{attacker}(M) \land \text{attacker}(N) \Rightarrow \text{attacker}(u(M, N, a[], b[M])). \]

Exercise

In our process semantics, secrecy is not affected by the exchange of new and in operations. Find Q and Q’ related by such exchanges such that

- both ensure the secrecy of some value;
- Proverif only proves it for Q.

A possible solution: freshness.pv.
Exercise: Needham-Schroeder

The file `nspk.pv` contains a partial definition of the original Needham-Schroeder public-key protocol.

1. Complete the definition of the responder role.
2. Following what we did in lecture 2 (`ded.pdf`, slide 6) model the secrecy of $n_b$ when the responder (believes he) is interacting with the honest agent $A$.
4. Fix the protocol, check with Proverif.

A solution: `nsl-secrecies.pv`

The encoding is slightly more general than above.

Demo HTML output with attack diagram.
Protocol clauses

\[
\begin{align*}
[0]_\rho^H &= \emptyset \\
[P | Q]_\rho^H &= [P]_\rho^H \cup [Q]_\rho^H \\
[!]P]_\rho^H &= [P]_\rho^H
\end{align*}
\]
\[
\begin{align*}
\llbracket 0 \rrbracket^H_\rho &= \emptyset \\
\llbracket \text{in}(c, x). \ P \rrbracket^H_\rho &= \llbracket \text{in}(c, x). \ P \rrbracket^H_\rho \cup \{\text{mess}(c_\rho, x)\} \\
\llbracket \text{out}(c, u). \ P \rrbracket^H_\rho &= \{H \Rightarrow \text{mess}(c_\rho, u_\rho)\} \cup \llbracket \text{out}(c, u). \ P \rrbracket^H_\rho \\
\llbracket \text{new} \ a. \ P \rrbracket^H_\rho &= \llbracket \text{new} \ a. \ P \rrbracket^H_\rho \cup (a \mapsto a[p'_1, \ldots, p'_n]) \\
\llbracket ! \ P \rrbracket^H_\rho &= \llbracket \ P \rrbracket^H_\rho \\
\llbracket P \mid Q \rrbracket^H_\rho &= \llbracket P \rrbracket^H_\rho \cup \llbracket Q \rrbracket^H_\rho
\end{align*}
\]

where \( H = \land_i \text{mess}(p_i, p'_i) \)
Protocol clauses

\[
\begin{align*}
[0]^H_\rho &= \emptyset \\
[P | Q]^H_\rho &= [P]^H_\rho \cup [Q]^H_\rho \\
[P]^H_\rho &= [P]^H_\rho + (x \mapsto x) \\
[\text{out}(c, u). P]^H_\rho &= \{ H \Rightarrow \text{mess}(c, u) \} \cup [P]^H_\rho \\
[\text{new } a. P]^H_\rho &= [P]^H_\rho + (a \mapsto a[p_1', \ldots, p_n']) \\
[\text{if } u = v \text{ then } P \text{ else } Q]^H_\rho &= [P]^H_\rho\sigma \cup [Q]^H_\rho \\
\end{align*}
\]

where \( H = \wedge_i \text{mess}(p_i, p_i') \)

and \( \sigma = \text{mgu}(u, v) \).
Protocol clauses

\[
[0]_\rho^H = \emptyset \quad [P \mid Q]_\rho^H = [P]_\rho^H \cup [Q]_\rho^H \quad [\! P]_\rho^H = [P]_\rho^H
\]

\[
\text{in}(c, x). \ [P]_\rho^H = [P]_\rho^H \cup \{\text{mess}(c, x)\}
\]

\[
\text{out}(c, u). \ [P]_\rho^H = \{H \Rightarrow \text{mess}(c, u)\} \cup [P]_\rho^H\wedge\text{mess}(c, x)
\]

\[
\text{new} \ a. \ [P]_\rho^H = [P]_\rho^H (a \mapsto a[p_1', \ldots, p_n'])
\]

\[
\text{if} \ u = v \text{ then } P \text{ else } Q]_\rho^H = [P]_\rho^H\sigma \cup [Q]_\rho^H\quad \text{where } \sigma = \text{mgu}(u, v)
\]

\[
\text{let} \ x = g(u_1, \ldots, u_n) \text{ in } P \text{ else } Q]_\rho^H = \bigcup_{(p', \sigma) \in X} [P]_\rho^H\sigma \cup [Q]_\rho^H \quad \text{where } X = \{(p', \sigma) \mid g(p_1', \ldots, p_n') \rightarrow p', \ \sigma = \text{mgu}(\wedge_i u_i \sigma = p_i')\}
\]

Example:

\[
\text{in}(c, x).\text{in}(c, y).\text{if} \ y = n \text{ then let } z = \text{sdec}(x, k) \text{ in } \text{out}(c, \text{senc}(\langle z, n \rangle, k))
\]
Semi-deciding non-derivability

Let $\mathcal{C}$ be the encoding of a system.

**Proposition**

If $m$ is not secret then (roughly) $\text{attacker}(m)$ is derivable from $\mathcal{C}$ using the consequence rule:

$$
\frac{H_1\sigma \; \ldots \; H_n\sigma \; (\vec{H} \Rightarrow C) \in \mathcal{C}}{C\sigma}
$$

Equivalently: if $\text{attacker}(m)$ is not derivable, then $m$ is secret.

**Goal**

Find a semi-decision procedure that allows to conclude often enough that a fact is not derivable from $\mathcal{C}$. 
Resolution with selection

Conventions

Let $\phi = \forall M_1, \ldots, M_k. \ H_1 \land H_n \Rightarrow C$ be a clause. Quantifiers may be omitted: free variables implicitly universally quantified. Hypotheses’ order is irrelevant: $\{H_i\}_i \Rightarrow C$, where $\{H_i\}_i$ is a multiset.

Resolution with selection

For each clause $\phi$, let $\text{sel}(\phi)$ be a subset of its hypotheses.

\[
\phi = (H_1' \land \ldots \land H_m' \Rightarrow C') \quad \psi = (H_1 \land \ldots \land H_n \Rightarrow C) \\
(\land_i H_i \land \land_{j \neq k} H_j' \Rightarrow C')\sigma
\]

With $\sigma = \text{mgu}(C', H_k)$, $\text{sel}(\phi) = \emptyset$, $H_k \in \text{sel}(\psi)$ and variables of $\phi$ and $\psi$ disjoint.
If $C'$ is a set of clauses, let $\text{solved}(C') = \{ \phi \in C' \mid \text{sel}(\phi) = \emptyset \}$.

**Proposition**

Let $C$ and $C'$ be two sets of clauses such that

- $C \subseteq C'$ and
- $C'$ is closed under resolution with selection.

If $F$ is derivable from $C$ then it is derivable from $\text{solved}(C')$, with a derivation of size (number of nodes) $\leq$ the original size.

**Goal:** saturate the initial set of clauses by resolution?
The selection strategy is crucial to obtain termination:

\[
\text{attacker}(x) \land \text{attacker}(y) \Rightarrow \text{attacker}(\text{aenc}(x, y))
\]
Resolution examples

○ The selection strategy is crucial to obtain termination:

\[ \text{attacker}(x) \land \text{attacker}(y) \Rightarrow \text{attacker}(\text{aenc}(x, y)) \]

○ Redundant clauses are often generated:

\[ \text{attacker}(x_{pkb}) \land \text{attacker}(\text{aenc}([na[x_{pkb}], x_{nb}, x_{pkb}], pk(sk_a)))) \Rightarrow \text{attacker}(\text{aenc}(x_{nb}, x_{pkb})) \]

Assume 2\textsuperscript{nd} assumption selected, resolve against constructor clause.
Resolution examples

- The selection strategy is crucial to obtain termination:
  \[ \text{attacker}(x) \land \text{attacker}(y) \Rightarrow \text{attacker}(\text{aenc}(x, y)) \]

- Redundant clauses are often generated:
  \[ \text{attacker}(x_{pkb}) \land \text{attacker}(\text{aenc}(\langle na[x_{pkb}], x_{nb}, x_{pkb} \rangle, \text{pk}(sk_a))) \Rightarrow \text{attacker}(\text{aenc}(x_{nb}, x_{pkb})) \]

  Assume 2\text{nd} assumption selected, resolve against constructor clause.

- Termination not achieved in general, as seen in NS shared-key:
  \[
  B \rightarrow A : \quad \text{senc}(n_b, k) \\
  A \rightarrow B : \quad \text{senc}(n_b - 1, k)
  \]
Logical completeness (2)

Subsumption

\((\{H_i\}_i \Rightarrow C) \sqsubseteq (\{H'_j\}_j \Rightarrow C')\) if there exists \(\sigma\) such that

1. \(C'\sigma = C\) and
2. for all \(j\), \(H'_j\sigma = H_i\) for some \(i\).

Given a set of clauses, let \(\text{elim}(C)\) be a set of clauses such that

for all \(\phi \in C\) there is \(\psi \in \text{elim}(C)\) such that \(\phi \sqsubseteq \psi\).

Saturation of an initial set of clauses \(C_0\)

1. initialize \(C := \text{elim}(C_0)\)
2. for each \(\phi\) generated from \(C\) by resolution, let \(C := \text{elim}(C \cup \{\phi\})\)
3. repeat step 2 until a fixed point is reached, let \(C'\) be the result.

Theorem

If \(F\) is derivable from \(C_0\) then it is derivable from \(\text{solved}(C')\).
Summing up: Proverif’s procedure

**Procedure for secrecy**

- Encode system as $C_0$.
- Saturate it to obtain $C'$.
- Declare secrecy of $m$ if solved($C'$) contains no clause with conclusion $\text{attacker}(m')$ with $m'\sigma = m$.

Remarks:
- Choice of selection function: at most one hypothesis, of the form $\text{attacker}(u)$ where $u$ is not a variable.
- Not covered here: treatment of equations, several optimizations.
- Differences with standard resolution: focus on deducidable facts rather than consistency; factorisation not needed (Horn).
Summing up: Proverif’s procedure

Procedure for secrecy
- Encode system as $C_0$.
- Saturate it to obtain $C'$.
- Declare secrecy of $m$ if solved($C'$) contains no clause with conclusion $\text{attacker}(m')$ with $m'\sigma = m$.

Remarks
- Choice of selection function: at most one hypothesis, of the form $\text{attacker}(u)$ where $u$ is not a variable.
- Not covered here: treatment of equations, several optimizations.
- Differences with standard resolution: focus on deducible facts rather than consistency; factorisation not needed (Horn).
Termination and decidability

Proverif’s procedure works very well in practice, but it offers no a priori guarantee...

Tagging

Secrecy is decidable for (reasonable classes of) tagged protocols.
- Blanchet & Podelski 2003: termination of resolution
- Ramanujan & Suresh 2003: decidability, but forbid blind copies

At most one blind copy

- Comon & Cortier 2003: decidability through (ordered) resolution

Illustration: resolution with selection on tagged NS shared-key
Correspondences

Roughly, express that if X happens then Y must have happened.

- If B thinks he’s completed the protocol with A, then A thinks he’s completed the protocol with B.

Events

Add events to the syntax of protocols:

(* Declaration *)

\text{event evName(type1,..,typeN)}.

(* Use inside processes *)

\text{P ::= ... | event evName(u1,..,uN); P}

Semantics extended as follows:

\[(\text{event } E. \ P \mid Q, \Phi) \xrightarrow{\tau} (P \mid Q, \Phi)\]
Queries

Definition

The query

\[
\text{query } x_1: t_1, \ldots, x_N: t_K; \\
\text{event}(E(u_1, \ldots, u_N)) \implies \text{event}(E'(v_1, \ldots, v_M))
\]

holds if for all traces of the system

- if the trace ends with an event rule for an event of the form \( E(u_i)_i \),
- there is a prior execution of the rule for an event of the form \( E'(v_j)_j \).

Note that variables of \( u_i \) are universally quantified while those only occurring in \( v_j \) are existentially quantified.

Example

\[
\text{query } na:\text{bitstring}, nb:\text{bitstring}; \\
\text{event}(\text{endR}(pka,pkb,na,nb)) \implies \text{event}(\text{endI}(pka,pkb,na,nb)).
\]
Exercise: mutual authentication

Extend \texttt{nsl-secrecies.pv} to check mutual authentication:

1. Declare and emit an event \texttt{endResponder(pka, pkb, na, nb)} expressing that the responder, running with identity \textit{pkb}, has completed an execution with \textit{pka}, and that the negotiated nonces are \textit{na} and \textit{nb}.

2. Do the same for the initiator.

3. Check that, when the responder has finished, an initiator has finished with the same parameters.

4. Consider the converse authentication property.
Exercise: injectivity

Proverif also allows to check injective correspondences:

query x1:t1, .., xN:tK;

\text{inj-event}(E(u_1,..,u_N)) \implies \text{inj-event}(E'(v_1,..,v_M))

holds if for all traces of the system there is an injective \( \phi \) such that
- if an event of the form \( E(u_i) \) is emitted at step \( \tau \),
- an event of the form \( E'(v_j) \) is emitted at step \( \phi(\tau) < \tau \).

Exercise:

1. Check that NSL satisfies mutual authentication in its injective form, which is the proper form.

2. Give a protocol that satisfies mutual authentication only in its non-injective form.
How does it work?

It is natural to encode events as outputs using a dedicated predicate. For example,

\[(\text{in}(c, x). \text{if } x = n_a \text{ then event } E)\]

would yield

\[(\text{attacker}(n_a) \Rightarrow \text{occurs}(E)).\]

Problem #1

This approximate encoding would only express that the event may occur. When checking \(E \Rightarrow E'\) we cannot over-approximate \(E'\)!

- We will see how “must occur” can be encoded in the language of Horn clauses and resolution.
How does it work?

Problem # 2

Because of the approximate encoding of fresh names, messages in the logic do not correspond uniquely to messages in the semantics.

The process:

```
new d : channel;
! new a : bitstring;
in(c, x : bool);
if x = true then event A(a); out(d, ok) else
if x = false then in(d, x : bitstring); event B(a)
```

should not satisfy query x : bitstring; event(B(x)) ==> event(A(x)).
How does it work?

Problem # 2

Because of the approximate encoding of fresh names, messages in the logic do not correspond uniquely to messages in the semantics.

The process:

```plaintext
new d : channel;
! new a : bitstring;
in(c, x : bool);
if x = true then event A(a); out(d, ok) else
if x = false then in(d, x : bitstring); event B(a)
```

should not satisfy query x : bitstring; event(B(x)) ==> event(A(x)).

We will ignore this problem in this lecture.
How does it work?

**Translation**

Use a predicate `begin(·)` for events that **must** occur, and `end(·)` for events that **may** occur.

Treat `event(M)` actions in processes using both **may** and **must**:

\[
\llbracket \text{event } M; P \rrbracket_H^\rho = \llbracket P \rrbracket_H^\rho \land \text{begin}(\text{event}(M\rho)) \cup \{ H\rho \Rightarrow \text{end}(\text{event}(M\rho)) \}
\]

We may look at `nspk-auth.pv` for concrete examples.

**Verification problem**

query \( x_1, \ldots, x_n; \text{event}(E(u_i)_i) \implies \text{event}(E'(v_j)_j) \)

\( \iff \) deriving \( E(u_i)_i \) requires to derive an instance of \( E'(v_j)_j \)
(ignoring problem \( \neq 2 \))

\( \iff \) for all sets \( \mathcal{E} \) of `begin(M)` open facts, \( E'(v_j)_j \) is derivable from \( C \cup \mathcal{E} \)
only if \( \mathcal{E} \) contains \( E'(v_j)_j \) (or a generalization of it).
So we want to verify the following:

for all sets $\mathcal{E}$ of $\text{begin}(M)$ open facts, $E'(v_j)_j$ is derivable from $\mathcal{C} \cup \mathcal{E}$ only if $\mathcal{E}$ contains $E'(v_j)_j$ (or a generalization of it).

But we don’t know $\mathcal{E}$ and can’t enumerate all of them!
Verifying correspondences through resolution

So we want to verify the following:

for all sets $\mathcal{E}$ of $\text{begin}(M)$ open facts, $E'(v_j)_j$ is derivable from $\mathcal{C} \cup \mathcal{E}$ only if $\mathcal{E}$ contains $E'(v_j)_j$ (or a generalization of it).

But we don’t know $\mathcal{E}$ and can’t enumerate all of them!

Key observation

If we never select on $\text{begin}(M)$ hypotheses, saturating on $\mathcal{C} \cup \mathcal{E}$ is the same as saturating on $\mathcal{C}$ and adding $\mathcal{E}$ afterwards.
Verifying correspondences through resolution

So we want to verify the following:

for all sets $E$ of $\text{begin}(M)$ open facts, $E'(v_j)_j$ is derivable from $C \cup E$ only if $E$ contains $E'(v_j)_j$ (or a generalization of it).

But we don’t know $E$ and can’t enumerate all of them!

Key observation

If we never select on $\text{begin}(M)$ hypotheses, saturating on $C \cup E$ is the same as saturating on $C$ and adding $E$ afterwards.

Procedure

- Build $C$ by translating the process and adding attacker clauses.
- Get $C'$ by saturating $C$, without selecting $\text{begin}(\cdot)$ hypotheses.
- Check that for all clauses ($\{H_i\}_i \Rightarrow C) \in \text{solved}(C')$, and all $\sigma$ such that $C\sigma = E(u_i)_i\sigma$, there exists $i$ such that $H_i\sigma$ is an instance of $E'(v_j)_j\sigma$. 
Definitions and usage in Proverif, on the blackboard.
Diff-equivalence: guessing exercises

Exercise

Consider the naive voting protocol where a voter sends his vote encrypted with the authority’s public key.
Can the vote be guessed? Model using diff-equivalence. Propose a fix.

Exercise

Consider the handshake protocol:

\[
A \rightarrow B : \text{senc}(s, pw) \\
B \rightarrow A : \text{senc}(\text{incr}(s), pw)
\]

Can \( k \) be guessed? Model using diff-equivalence and a phase:

\text{phase 1; new } w; \text{ out}(c, \text{choice}[w, pw]).

Exercise

We may also consider a variant of Kerberos: on the board.
Diff-equivalence: unlinkability exercise

A system using blind tokens: on the board.