ProVerif
Automatic Cryptographic Protocol Verifier
User Manual for Untyped Inputs

Bruno Blanchet*
INRIA Paris-Rocquencourt, France

September 7, 2021

Warning! This manual documents the untyped inputs of ProVerif (untyped Horn clauses, untyped pi calculus). These input formats are no longer actively developed. We recommend coding your protocols in the typed pi calculus format, described in the file `manual.pdf`.

1 Introduction

This manual describes the untyped input syntax and output of ProVerif. It does not describe the internal algorithms used in the system. These algorithms have been described in various research papers [6, 2, 7, 1, 3, 4, 8, 10, 5, 9], that can be downloaded on

`http://www.di.ens.fr/~blanchet/publications/index.html`.

The tool can take two formats as input. The first one is in the form of Horn clauses (logic programming rules), and corresponds to the system described in [6]. The second one is in the form of a process in an extension of the pi calculus, described in [2]. In both cases, the output of the system is essentially the same.

2 Common remarks on the syntax

Comments can be included in input files. Comments are surrounded by `(*` and `*)`. Nested comments are not supported.

Identifiers begin with a letter (uppercase or lowercase) and contain any number of letters, digits, the underscore character (_), the quote character ('), as well as accented letters of the ISO Latin 1 character set. Case is significant. Each input system has a number of keywords that cannot be used as ordinary identifiers.

In case of syntax error, the system indicates the character position of the error (line and column numbers). Please use your text editor to find the position of the error. (The error messages can be interpreted by `emacs`.)

3 Command-line options

The syntax of the command-line is

`./proverif` ⟨options⟩ ⟨filename⟩

where the ⟨options⟩ can be

*This work was partly done while the author was at École Normale Supérieure, Paris and at Max-Planck-Institut für Informatik, Saarbrücken.

- `-in` ⟨format⟩

  Choose the input format (`horn`, `horntype`, `pi`, `pitype`). When the `-in` option is absent, the input format is chosen according to the file extension, as detailed below. The recommended input format is the typed pi calculus, which corresponds to the option `-in pitype`, and is the default when the file extension is `.pv`. It is described in `manual.pdf`. The other formats are no longer actively developed. Input may also be provided using the untyped pi calculus (option `-in pi`, the default when the file extension is `.pi`), typed Horn clauses (option `-in horntype`, the default when the file extension is `.horntype`), and untyped Horn clauses (option `-in horn`, the default for all other file extensions). This manual documents the untyped Horn clauses and the untyped pi calculus input formats.

- `-out` ⟨format⟩

  Choose the output format, either `solve` (analyze the protocol) or `spass` (stop the analysis before resolution, and output the clauses in the format required for use in the Spass first-order theorem prover, see `http://www.spass-prover.org/`). The default is `solve`. When you select `-out spass`, you must add the option `-o` ⟨filename⟩ to specify the file in which the clauses will be output.

- `-TulaFale` ⟨version⟩

  For compatibility with the web service analysis tool TulaFale (see the tool download at `http://research.microsoft.com/projects/samoa/`). The version number is the version of TulaFale with which you would like compatibility. Currently, only version 1 is supported.

- `-color`

  Display a colored output on terminals that support ANSI color codes. (Will result in a garbage output on terminals that do not support these codes.) Unix terminals typically support ANSI color codes. For `emacs` users, you can run ProVerif in a shell buffer with ANSI color codes as follows:

  - start a shell with `M-x shell`
  - load the `ansi-color` library with `M-x load-library RET ansi-color RET`
  - activate ANSI colors with `M-x ansi-color-for-comint-mode-on`
  - now run ProVerif in the shell buffer.

  You can also activate ANSI colors in shell buffers by default by adding the following to your `.emacs`:

  ```
  (autoload 'ansi-color-for-comint-mode-on "ansi-color" nil t)
  (add-hook 'shell-mode-hook 'ansi-color-for-comint-mode-on)
  ```

- `-help` or `--help`

  Display a short summary of command-line options.

4 Input as Horn clauses

By default, the executable program `proverif` takes Horn clauses as input. You can run it as follows:

`./proverif` ⟨filename⟩

where ⟨filename⟩ references a file containing the Horn clauses, in the format explained below. The system then basically determines whether a fact can be derived from the clauses. If true, a proof is given. As shown in [6], this can be used to determine secrecy properties of protocols: if a certain fact cannot be derived from the clauses, then the secrecy of a certain value is preserved. A difference with first-order theorem provers that perform a similar task, is that correctness and completeness are reversed. Here, correctness means that if a value is not secret, the system says so, that is, if a fact is derivable, the system says so. Completeness means that if a fact is not derivable, then the system says so. We sometimes drop completeness (that is, we lose precision; see options below), but never correctness.

⟨clause⟩ ::= [(⟨fact⟩ `&`)*⟨fact⟩ `->`] ⟨fact⟩
| (⟨fact⟩ `&`)*⟨fact⟩ `<->` ⟨fact⟩
| (⟨fact⟩ `&`)*⟨fact⟩ `<=>` ⟨fact⟩
⟨fact⟩ ::= ⟨ident⟩`:`seq⟨term⟩
| ⟨term⟩ `<>` ⟨term⟩
⟨term⟩ ::= ⟨ident⟩`(`seq⟨term⟩`)`
| ⟨ident⟩`[`seq⟨term⟩`]`
| `(`seq⟨term⟩`)`
| ⟨ident⟩
⟨factformat⟩ ::= ⟨ident⟩`:`seq⟨termformat⟩
⟨termformat⟩ ::= ⟨ident⟩`(`seq⟨termformat⟩`)`
| ⟨ident⟩`[`seq⟨termformat⟩`]`
| `(`seq⟨termformat⟩`)`
| ⟨ident⟩
| `*`⟨ident⟩

where seq⟨X⟩ is a sequence of X: seq⟨X⟩ = [(⟨X⟩`,`)*⟨X⟩] = ⟨X⟩`,` ... `,`⟨X⟩. (The sequence can be empty, it can be one element ⟨X⟩, or it can be several elements ⟨X⟩ separated by commas.)

Figure 1: Grammar of facts and clauses

The keywords of this input system are `data`, `elimtrue`, `equation`, `fun`, `not`, `nounif`, `param`, `pred`, `query`, and `reduc`.

The input file consists of a list of declarations, followed by the keyword `reduc` and a list of clauses:

⟨declaration⟩* `reduc` (⟨clause⟩`;`)* ⟨clause⟩`.`

The syntax of facts and clauses is given in Figure 1. In this grammar X^* means any number of repetitions of X, $[X]$ means X or nothing. Text in typewriter style should appear as it is in the input file. Text between ⟨ and ⟩ represents non-terminals.

Declarations can be any of the following:

- `param` ⟨name⟩ `=` ⟨value⟩`.`

  This declaration sets the value of configuration parameters. The following cases are supported:

  - `param verboseRules = false.`
    `param verboseRules = true.`
    Display the number of clauses every 200 clause created during the solving process (`false`) or display each clause created during the solving process (`true`).
  - `param verboseRedundant = false.`
    `param verboseRedundant = true.`
    Display eliminated redundant clauses when `true`.
  - `param verboseCompleted = false.`
    `param verboseCompleted = true.`
    Display completed set of clauses after saturation when `true`.
  - `param verboseEq = true.`
    `param verboseEq = false.`
    Display information on handling of equational theories when `true`.

– `param verboseTerm = true.`
`param verboseTerm = false.`

Display information on termination when `true` (changes in the selection function to improve termination; termination warnings).

– `param maxDepth = none.`
`param maxDepth =` n`.`

Do not limit the depth of terms (`none`) or limit the depth of terms to n, where n is an integer. A negative value means no limit. When the depth is limited to n, all terms of depth greater than n are replaced with new variables. Limiting the depth can be used to enforce termination of the solving process at the cost of precision.

– `param maxHyp = none.`
`param maxHyp =` n`.`

Do not limit the number of hypotheses of clauses (`none`) or limit it to n, where n is an integer. A negative value means no limit. When the number of hypotheses is limited to n, arbitrary hypotheses are removed from clauses, so that only n hypotheses remain. Limiting the number of hypotheses can be used to enforce termination of the solving process at the cost of precision (although in general limiting the depth by the above declaration is enough to obtain termination).

– `param selFun = TermMaxsize.`
`param selFun = Term.`
`param selFun = NounifsetMaxsize.`
`param selFun = Nounifset.`

Chooses the selection function that governs the resolution process. All selection functions avoid unifying on facts indicated by a `nounif` declaration. `Nounifset` does exactly that. `Term` automatically avoids some other unifications, to help termination, as determined by some heuristics. `NounifsetMaxsize` and `TermMaxsize` choose the fact of maximum size when there are several possibilities. This choice sometimes gives impressive speedups.

– `param stopTerm = true.`
`param stopTerm = false.`

Display a warning and wait for user answer when the system thinks the solving process will not terminate (`true`), or go on as if nothing had happened (`false`). This setting applies only to the selection functions `NounifsetMaxsize` and `Nounifset`. (See parameter `selFun`.)

– `param redundancyElim = simple.`
`param redundancyElim = no.`
`param redundancyElim = best.`

An elimination of redundant clauses has been implemented: when a clause without selected hypotheses is derivable from other clauses without selected hypothesis, it is removed. With `redundancyElim = simple`, this is applied for newly generated clauses. With `redundancyElim = no`, this is never applied. With `redundancyElim = best`, this is also applied when an old clause can be derived from other old clauses plus the new clause.

– `param redundantHypElim = beginOnly.`
`param redundantHypElim = false.`
`param redundantHypElim = true.`

When a clause is of the form $H \wedge H' \rightarrow C$, and there exists σ such that $\sigma H \subseteq H'$ and σ does not change the variables of H' and C, then the clause can be replaced with $H' \rightarrow C$ (since there are implications in both directions between these clauses).

This replacement is done when `redundantHypElim = true.`, or when `redundantHypElim = beginOnly.` and H contains a `begin` event. Indeed, testing this property takes time, and slows down small examples. On the other hand, on big examples, in particular when they contain several `begin` events (or blocking facts), this technique can yield huge speedups.

– `param reconstructDerivation = true.`
`param reconstructDerivation = false.`

When a fact is derivable, should we reconstruct the corresponding derivation? (This setting has been introduced because in some extreme cases reconstructing a derivation can consume a lot of memory.)

- `param simplifyDerivation = true.`
  `param simplifyDerivation = false.`
  Should the derivation be simplified by removing duplicate proofs of the same `attacker` facts?
- `param abbreviateDerivation = true.`
  `param abbreviateDerivation = false.`
  When `abbreviateDerivation = true`, ProVerif defines symbols to abbreviate terms that represent names $a[\ldots]$ before displaying the derivation, and uses these abbreviations in the derivation. These abbreviations generally make reading the derivation easier by reducing the size of terms.
- `param displayDerivation = true.`
  `param displayDerivation = false.`
  Should the derivation be displayed? Disabling derivation display is useful for very big derivations.
- `param symbOrder = "`$f_1 > \cdots > f_n$`".`
  ProVerif uses a lexicographic path ordering in order to prove termination of convergent equational theories. By default, it uses a heuristic to build the ordering of function symbols underlying this lexicographic path ordering. This setting allows the user to set this ordering of function symbols.

In the above list, the default value is the first mentioned. The system also accepts `no` instead of `false` and `yes` instead of `true`.

- `fun` ⟨ident⟩`/`n`.`
  Declares a function symbol ⟨ident⟩ of arity n.
- `data` ⟨ident⟩`/`n`.`
  `data` f`/`n`.` declares a data function symbol f of arity n. Data function symbols are similar to tuples: the adversary can construct and decompose them. The system implicitly adds the equivalence $p : f(x_1, \ldots, x_n) \Leftrightarrow p : x_1 \wedge \ldots \wedge p : x_n$ for each predicate p declared `decompData`.
- `equation` ⟨term⟩ `=` ⟨term⟩`.`
  `equation` M_1 `=` M_2 says that the terms M_1 and M_2 are in fact equal. The function symbols in the equation should be only already declared constructors. The treatment of equations is still rather limited. The equation $f(x, g(y)) = f(y, g(x))$, used for Diffie-Hellman key agreements, is known to work. The system may not terminate when more complex equations are entered. In the presence of both equations and inequality constraints, the system is not complete (but still correct): the inequality constraints are deemed true when the terms are syntactically different, without taking into account the equations.
- `query` ⟨fact⟩`.`
  Indicates that the system should determine whether ⟨fact⟩ is true or not. If ⟨fact⟩ contains variables, determine which instances of ⟨fact⟩ are true.
- `nounif` ⟨factformat⟩[`/`n]`.`
  Modifies the selection of facts to be resolved upon, to avoid resolving facts that match ⟨factformat⟩. For a fact ⟨fact⟩ to match ⟨factformat⟩, ⟨fact⟩ must contain a variable when ⟨factformat⟩ contains one, and any term when ⟨factformat⟩ contains `*` followed by a variable name. The optional integer n indicates how much we should avoid resolution on facts that match ⟨factformat⟩: the greater n, the more such resolutions will be avoided.

- `pred` ⟨ident⟩`/`n seq⟨ident⟩`.`

  `pred` p`/`n $i_1, \ldots, i_n$`.` declares a new predicate p, of arity n, with special properties described by $i_1, \ldots, i_n$.

  The following properties are allowed for $i_1, \ldots, i_n$:

  - `block`: Declares the predicate p as a blocking predicate. Blocking predicates may appear in hypotheses or conclusions of clauses, but not both for the same predicate.

    When they appear in hypotheses, instead of trying to prove facts containing these predicates (which is impossible since no clause implies such facts), the system collects hypotheses containing the blocking predicates necessary to prove the queries. This is useful in particular to prove authenticity [7].

    When they appear in conclusions, the system makes sure to really know whether the fact C in question is derivable, and does not return clauses H `->` C for which it is not clear whether H is derivable or not.
  - `elimVarStrict`: Tells the system to that p`:`$new\_name[i], \ldots, new\_name[i]$ holds. Then ProVerif removes the hypothesis p`:`$x_1, \ldots, x_n$ where $x_1, \ldots, x_n$ are variables that do not appear elsewhere in a clause.
  - `elimVar`: Tells the system to that p`:`$new\_name[i], \ldots, new\_name[i]$ holds. Then ProVerif removes the hypothesis p`:`$x_1, \ldots, x_n$ where $x_1, \ldots, x_n$ are variables that do not appear elsewhere in a clause, except possibly in inequality facts. Removing such hypotheses `attacker:`x is complete for proving secrecy or authenticity, because there always exists a value of x that makes true both the inequality facts, and `attacker:`x. In general, however, removing p`:`$x_1, \ldots, x_n$ where $x_1, \ldots, x_n$ may appear in inequalities leads to a sound approximation. (When it is important that no underivable facts are considered derivable, the transformation is applied only when $x_1, \ldots, x_n$ do not appear elsewhere in the clause, as for `elimVarStrict`.)
  - `decompData`: Adds the following clauses, where p is a predicate of arity m and f is any function symbol of arity n declared `data` or a tuple function.

    $$p\texttt{:}f\texttt{(}x_{11}\texttt{,}\ldots\texttt{,}x_{1n}\texttt{),}\ldots\texttt{,}f\texttt{(}x_{m1}\texttt{,}\ldots\texttt{,}x_{mn}\texttt{)}\ \texttt{->}\ p\texttt{:}x_{1i}\texttt{,}\ldots\texttt{,}x_{mi}$$
    $$p\texttt{:}x_{11}\texttt{,}\ldots\texttt{,}x_{m1}\ \texttt{\&}\ \ldots\ \texttt{\&}\ p\texttt{:}x_{1n}\texttt{,}\ldots\texttt{,}x_{mn}\ \texttt{->}$$
    $$\quad p\texttt{:}f\texttt{(}x_{11}\texttt{,}\ldots\texttt{,}x_{1n}\texttt{),}\ldots\texttt{,}f\texttt{(}x_{m1}\texttt{,}\ldots\texttt{,}x_{mn}\texttt{)}$$

    For example, when p is a unary predicate and f is a tuple function, we add:

    $$p\texttt{:(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}\ \texttt{->}\ p\texttt{:}x_i$$
    $$p\texttt{:}x_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ p\texttt{:}x_n\ \texttt{->}\ p\texttt{:(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}$$

    for all n and $i \in \{1, \ldots, n\}$. These clauses are treated in a specially optimized way, since they are used in most protocols.
  - `decompDataSelect`: same as `decompData`, but allows the selection of facts p`:`$x_1, \ldots, x_n$ while `decompData` does not. (It is in general better not to select such facts, because it leads to non-termination in the presence of data decomposition clauses; there are however exceptions for some rare cases.)
  - `memberOptim`: This must be used only when p is defined by

    $$p : x, f(x, y);$$
    $$p : x, y\ \texttt{->}\ p : x, f(x', y).$$

    where f is a data constructor. It turns on the following optimization: $p' : x \wedge p : M_1, x \wedge \ldots \wedge p : M_n, x$ where p' is declared `decompData` and p is declared `memberOptim` is replaced with $p' : x \wedge p' : M_1 \wedge \ldots \wedge p' : M_n$ when x does not occur elsewhere (just take $x = f(M_1, \ldots f(M_n, x'))$ and notice that $p' : x$ if and only if $p' : M_1$, ..., $p' : M_n$, and $p' : x'$), or when the clause has no selected hypothesis. In the last case, this introduces an approximation.

The replacement is also possible when x occurs in several predicates declared `decompData`. However, when x occurs in several `memberOptim` predicates, the transformation may introduce an approximation. (For example, consider p_1 and p_2 defined as above respectively using f_1 and f_2 as data constructors. Then $p_1 : M, x \wedge p_2 : M', x$ is never true: for it to be true, x should be at the same time $f_1(\_, \_)$ and $f_2(\_, \_)$.)

- `elimtrue` ⟨fact⟩`.`

  The declaration `elimtrue` F means that σF is true for all σ; the fact F is added to the set of clauses.

  Then in a clause $R =$ F' `&` H `->` C, if F' unifies with F with most general unifier σ_u and all variables of F' modified by σ_u do not occur in the rest of R then the hypothesis F' can be removed: R is transformed into H `->` C, by resolving with F.

- `not` ⟨fact⟩`.`

  Adds a secrecy assumption, saying that ⟨fact⟩ cannot be proved from the clauses. Then the system can remove all clauses that contain fact in their hypotheses. (These clauses can never be applied.) This speeds up the system. At the end of the solving process, the system checks that ⟨fact⟩ can indeed not be derived from the clauses. If it can be derived, the proof fails with an error message.

Two kinds of functions may appear in terms: constructors and names. Constructors are followed by their parameters between parentheses: f`(`M_1`,`...`,`M_n`)`. A constructor without parameter can be written f`()` or simply f. Constructors must be declared with `fun` f`/`n`.`, as mentioned in the declarations. Names are followed by their parameters between brackets: a`[`M_1`,`...`,`M_n`]`. A name without parameter must be written a`[]`. Names are not declared before being used. At first, constructors were designed to represent cryptographic primitives, and names to represent fresh names created by the protocol. However, there is no difference between names and constructors from the point of view of the solver. We advise you to use constructors rather than names, since the declaration of constructors is a guarantee against typesetting errors.

So terms M can be either constructor applications f`(`M_1`,`...`,`M_n`)`, name applications a`[`M_1`,`...`,`M_n`]`, tuples `(`M_1`,`...`,`M_n`)`, or identifiers x that can be used for variables or constructors without parameters. Note that the term (M) is different from M: (M) is a tuple with a single component containing M.

Facts can be the application of a predicate to terms p`:`M_1`,`...`,`M_n. (This is written this way and not p`(`M_1`,`...`,`M_n`)` only for historical reasons.) They can also be M`<>`M', meaning M is different from M'. Inequalities are allowed only in hypotheses of clauses, not in conclusions, queries `query` ⟨fact⟩`.`, and secrecy assumptions `not` ⟨fact⟩`.`.

Clauses can be F_1 `&` ... `&` F_n `->` F, meaning F_1 and ... and F_n implies F. They can also be simply F, meaning that F is true, without any hypothesis.

They can also be F_1 `&` ... `&` F_n `<->` F, meaning F_1 and ... and F_n is equivalent to F. This is allowed only when F_i do not contain inequality constraints, σF_i is smaller than σF for all σ, and no two facts F of different equivalence declarations unify. It then generates the clauses F_1 `&` ... `&` F_n `->` F, F `->` F_i, and furthermore enables the replacement of σF with σF_1 `&` ... `&` σF_n in all clauses.

The declaration F_1 `&` ... `&` F_n `<=>` F is a synonym for F_1 `&` ... `&` F_n `<->` F. It is kept only for backward compatibility.

The goal of the system is to determine whether the facts declared in queries can be derived from the given clauses.

5 Input as process in extension of the pi calculus

To give a pi calculus process as input to ProVerif, you have to add the command line option `-in pi`, or to use a filename that ends with `.pi`. You can then run ProVerif by:

`./proverif -in pi` ⟨filename⟩

where ⟨filename⟩ references a file containing the process, in the format explained below.

⟨term⟩ ::= ⟨ident⟩`(`seq⟨term⟩`)`
| `(`seq⟨term⟩`)`
| ⟨ident⟩
| `choice[`⟨term⟩`,`⟨term⟩`]`

⟨pattern⟩ ::= ⟨ident⟩
| `(`seq⟨pattern⟩`)`
| ⟨ident⟩`(`seq⟨pattern⟩`)`
| `=`⟨term⟩

⟨clause⟩ ::= [(⟨fact⟩ `&`)*⟨fact⟩ `->`] ⟨fact⟩ [`where` seq⟨ident⟩ `can fail`]
| (⟨fact⟩ `&`)*⟨fact⟩ `<->` ⟨fact⟩ [`where` seq⟨ident⟩ `can fail`]
| (⟨fact⟩ `&`)*⟨fact⟩ `<=>` ⟨fact⟩ [`where` seq⟨ident⟩ `can fail`]

⟨fact⟩ ::= ⟨ident⟩`:`seq⟨term⟩
| ⟨term⟩ `<>` ⟨term⟩
| ⟨term⟩ `=` ⟨term⟩

⟨process⟩ ::= `(` ⟨process⟩ `)`
| ⟨ident⟩
| `!` ⟨process⟩
| `0`
| `new` ⟨ident⟩`;` ⟨process⟩
| `if` ⟨fact⟩ `then` ⟨process⟩ [`else` ⟨process⟩]
| `in(`⟨term⟩`,` ⟨pattern⟩`)`[`;` ⟨process⟩]
| `out(`⟨term⟩`,` ⟨term⟩`)`[`;` ⟨process⟩]
| `let` ⟨pattern⟩ `=` ⟨term⟩ `in` ⟨process⟩ [`else` ⟨process⟩]
| `let` seq⟨ident⟩ `suchthat` ⟨fact⟩ `in` ⟨process⟩ [`else` ⟨process⟩]
| ⟨process⟩ `|` ⟨process⟩
| `event` ⟨term⟩ [`;` ⟨process⟩]
| `phase` n [`;` ⟨process⟩]
| `sync` n [`;` ⟨process⟩]

Figure 2: Grammar of processes

The keywords of this input system are `among`, `and`, `can`, `choice`, `clauses`, `data`, `diff`, `elimtrue`, `else`, `equation`, `event`, `fail`, `free`, `fun`, `if`, `in`, `let`, `new`, `noninterf`, `not`, `nounif`, `otherwise`, `out`, `param`, `phase`, `putbegin`, `pred`, `private`, `process`, `query`, `reduc`, `suchthat`, `sync`, `then`, `weaksecret`, and `where`.

The input file consists of a list of declarations, followed by the keyword `process` and a process:

⟨declaration⟩* `process` ⟨process⟩

The syntax of terms and processes is given in Figures 2 and 3, using the same conventions as in Section 4.

Declarations can be any of the following:

- `param` ⟨name⟩ `=` ⟨value⟩`.`

  This declaration sets the value of configuration parameters. The cases mentioned in Section 4 are supported, as well as the following ones:

  - `param attacker = active.`
    `param attacker = passive.`

⟨gterm⟩ ::= ⟨ident⟩`(`seq⟨gterm⟩`)`
| ⟨ident⟩`[`seq⟨gbinding⟩`]`
| `(`seq⟨gterm⟩`)`
| ⟨ident⟩

⟨gbinding⟩ ::= `!`n `=` ⟨gterm⟩
| ⟨ident⟩ `=` ⟨gterm⟩

⟨gfact⟩ ::= ⟨ident⟩`:`seq⟨gterm⟩ [`phase` n]
| ⟨gterm⟩ `<>` ⟨gterm⟩
| ⟨gterm⟩ `=` ⟨gterm⟩

⟨realquery⟩ ::= ⟨gfact⟩ `==>` ⟨hyp⟩

⟨hyp⟩ ::= ⟨hyp⟩ `|` ⟨hyp⟩
| ⟨hyp⟩ `&` ⟨hyp⟩
| ⟨gfact⟩
| `(`⟨hyp⟩`)`
| `(`⟨realquery⟩`)`

⟨query⟩ ::= `putbegin ev:`seq⟨ident⟩[`;` ⟨query⟩]
| `putbegin evinj:`seq⟨ident⟩[`;` ⟨query⟩]
| `let` ⟨ident⟩ `=` ⟨gterm⟩[`;` ⟨query⟩]
| ⟨gfact⟩[`;` ⟨query⟩]
| ⟨realquery⟩[`;` ⟨query⟩]

⟨gtermformat⟩ ::= *same as* ⟨gterm⟩ *with additional* `*`⟨ident⟩

⟨gfactformat⟩ ::= ⟨ident⟩`:`seq⟨gtermformat⟩ [`phase` n]

Figure 3: Grammar of queries and `nounif`

Indicates whether the attacker is active (`param attacker = active.`) or passive (`param attacker = passive.`). An active attacker can read messages, compute, and send messages. A passive attacker can read messages and compute but not send messages.

– `param keyCompromise = none.`
`param keyCompromise = approx.`
`param keyCompromise = strict.`

By default (`param keyCompromise = none.`), it is assumed that session keys are not a priori compromised. Otherwise, it is assumed that some session keys are compromised (known by the adversary). Then the system determines whether the secrets of other sessions can be obtained by the adversary. In this case, the names that occur in queries always refer to names of non-compromised sessions (the attacker has all names of compromised sessions), and the events that occur before an arrow `==>` in a query are executed only in non-compromised sessions. With `param keyCompromise = approx.`, the compromised sessions are considered as executing possibly in parallel with non-compromised ones. With `param keyCompromise = strict.`, the compromised sessions are finished before the non-compromised ones begin. The chances of finding an attack are greater with `param keyCompromise = approx.`. (It may be a false attack due to the approximations made in the verifier.)

– `param movenew = false.`
`param movenew = true.`

Sets whether the system should try to move restrictions under inputs, to have a more precise analysis (`param movenew = true.`), or leave them where the user has put them (`param movenew = false.`).

– `param predicatesImplementable = check.`
`param predicatesImplementable = nocheck.`

Sets whether the system should check that predicate calls are implementable. See the `clauses` declaration below for more details on this check. It is advised to leave the check turned on, as it is by default. Otherwise, the semantics of the processes may not be well-defined.

– `param verboseClauses = none.`
`param verboseClauses = explained.`
`param verboseClauses = short.`

When `verboseClauses = none`, ProVerif does not display the clauses it generates. When `verboseClauses = short`, it displays them. When `verboseClauses = explained`, it adds an English sentence after each clause it generates to explain where this clause comes from.

– `param explainDerivation = true.`
`param explainDerivation = false.`

When `explainDerivation = true`, ProVerif explains in English each step of the derivation (returned in case of failure of a proof). This explanation refers to program points in the given process. When `explainDerivation = false`, it displays the derivation by referring to the clauses generated initially.

– `param removeUselessClausesBeforeDisplay = true.`
`param removeUselessClausesBeforeDisplay = false.`

When `removeUselessClausesBeforeDisplay = true`, ProVerif removes subsumed clauses and tautologies from the initial clauses before displaying them, to avoid showing many useless clauses. When `removeUselessClausesBeforeDisplay = false`, all generated clauses are displayed.

– `param reconstructTrace = true.`
`param reconstructTrace = false.`

With `param reconstructTrace = true.`, when a query cannot be proved, the tool tries to build a pi calculus execution trace that is a counter-example to the query [5].

This feature is currently incompatible with key compromise (`param keyCompromise = approx.` or `param keyCompromise = strict.`).

Moreover, for `noninterf` and `choice`, it reconstructs a trace, but this trace may not always prove that the property is wrong: for `noninterf`, it reconstructs a trace until a program point at which the process behaves differently depending on the value of the secret (takes a different branch of a test, for instance), but this different behavior is not always observable by the adversary; similarly, for `choice`, it reconstructs a trace until a program point at which the process using the first argument of `choice` behaves differently from the process using the second argument of `choice`.

For injective queries, the trace reconstruction proceeds in two steps. In the first step, it reconstructs a trace that corresponds to the derivation found by resolution. This trace generally executes events once, so does not contradict injectivity. In a second step, it tries to reconstruct a trace that executes certain events twice while it executes other events once, in such a way that injectivity is really contradicted. This second step may fail even when the first one succeeds. For non-injective queries (including secrecy), when a trace is found, it is a counter-example to the query, which is then false.

– `param traceBacktracking = true.`
`param traceBacktracking = false.`

Allow or disable backtracking when reconstructing traces. In most cases, when traces can be found, they are found without backtracking. Disabling backtracking makes it possible to display the trace during its computation, and to forget previous states of the trace. This reduces memory consumption, which can be necessary for reconstructing very big traces.

– `param unifyDerivation = true.`
`param unifyDerivation = false.`

When set to `true`, activates a heuristic that increases the chances of finding a trace that corresponds to a derivation. This heuristic unifies messages received by the same input (same ocurrence and same session identifiers) in the derivation. Indeed, these messages must be equal if the derivation corresponds to a trace.

– `param traceDisplay = short.`
`param traceDisplay = long.`
`param traceDisplay = none.`

Choose the format in which the trace is displayed after trace reconstruction. By default (`param traceDisplay = short.`), outputs the labels of a labeled reduction. With `param traceDisplay = long.`, outputs the current state before each input and before and after each I/O reduction, as well as the list of all executed reductions. With `param traceDisplay = none.`, the trace is not displayed.

- [`private`] `fun` ⟨ident⟩`/`n`.`

  `fun` f`/`n`.` declares a function symbol f of arity n. This function symbol is a constructor. When `private` is not present, the function can be applied by the attacker. When `private` is present, the function cannot be applied by the attacker. This last case is useful to model tables of keys stored in a server, for instance. Only the server can use the table to get associations between host names and keys.

- `data` ⟨ident⟩`/`n`.`

  `data` f`/`n`.` declares a data function symbol f of arity n. Data function symbols are similar to tuples: the adversary can construct and decompose them.

- [`private`] `reduc` (⟨ident⟩`(`seq⟨term⟩`) = `⟨term⟩`;`)*
  ⟨ident⟩`(`seq⟨term⟩`) = `⟨term⟩`.`

  This declares destructors:

$$\begin{aligned}\texttt{reduc}\ & f\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_n\texttt{) = }M_0\texttt{;}\\ & f\texttt{(}M'_1\texttt{,}\ldots\texttt{,}M'_n\texttt{) = }M'_0\texttt{;}\\ & \ldots\\ & f\texttt{(}M''_1\texttt{,}\ldots\texttt{,}M''_n\texttt{) = }M''_0\texttt{.}\end{aligned}$$

declares the destructor f, of arity n, with the given rewrite rules. When a term $f(M_1, \ldots, M_n)$ is met, it is replaced by M_0, and similarly for the other rules. When no rule can be applied, the destructor is not defined; the process blocks. The destructor must be deterministic, that is, when several rules can be applied, they must all yield the same result. Otherwise, an error message is displayed. The terms $M_0, \ldots, M_n, M'_0, \ldots, M'_n, \ldots$ must contain only variables and constructors.

- [`private`] `reduc` (⟨ident⟩`(`seq⟨mfterm⟩`)` `=` ⟨mfterm⟩ [`where` seq⟨ident⟩ `can fail`] `otherwise`)*
  ⟨ident⟩`(`seq⟨mfterm⟩`)` `=` ⟨mfterm⟩ [`where` seq⟨ident⟩ `can fail`]`.`

  This is an extended declaration for destructors. The terms ⟨nfterm⟩ can be either the special constant `fail`, which represents the failure of a term, or terms ⟨term⟩ containing only variables and constructors.

  `reduc` f`(`M_1`,...,`M_n`)` `=` M_0 `where` $x_1, \ldots, x_k$ `can fail`
  `otherwise` f`(`M'_1`,...,`M'_n`)` `=` M'_0 `where` $x'_1, \ldots, x'_{k'}$ `can fail`
  `otherwise ...`
  `otherwise` f`(`M''_1`,...,`M''_n`)` `=` M''_0 `where` $x''_1, \ldots, x''_{k''}$ `can fail.`

  declares the destructor f with the given rewrite rules. When a term $f(N_1, \ldots, N_n)$ is met, if it is an instance of $f(M_1, \ldots, M_n)$, then it is replaced with the corresponding instance of M_0. Otherwise, if it is an instance of $f(M'_1, \ldots, M'_n)$, then it is replaced with the corresponding instance of M'_0, and so on. Each rewrite rule is applied only if the previous rewrite rules cannot be applied. Hence, the destructor is deterministic by construction.

  The variables $x_1, \ldots, x_k$ mentioned in "`where` $x_1, \ldots, x_k$ `can fail`" can be replaced with any term, and also with the constant `fail`. Such variables are allowed only at the root of terms $M_1, \ldots, M_n, M_0$. When there is no such variable, the indication "`where` $x_1, \ldots, x_k$ `can fail`" is omitted. Other variables can be replaced with any term but not with the constant `fail`; they can occur anywhere. The constant `fail` used in $N_i (i > 0)$ represents that the i-th argument of f failed; when it is used in the result M_0, it means that the evaluation of f itself fails in this case. Using this constant `fail`, we can, for instance, define destructors that do not fail even if some of their arguments fail.

  Examples include:

```
fun true/0.
fun false/0.
reduc equal(x,x) = true
      otherwise equal(x,y) = false.
```

  defines an equality destructor, which returns true when its two arguments are equal and false otherwise. This destructor fails when one of its arguments fails.

```
reduc test(true, x, y) = x where x, y can fail
      otherwise test(false, x, y) = y where x,y can fail.
```

  defines a test destructor that returns its second argument when its first argument is true, its third argument when its first argument is false, and fails otherwise. Notice that `x` and `y` are allowed to be `fail`, so that, when the first argument is true, the second argument is returned even if the third argument fails, and symmetrically in the other case.

- `equation` ⟨term⟩ `=` ⟨term⟩`.`

  `equation` M_1 `=` M_2 says that the terms M_1 and M_2 are in fact equal. The function symbols in the equation should be only already declared constructors. The treatment of equations is still rather limited. The equation $f(x, g(y)) = f(y, g(x))$, used for Diffie-Hellman key agreements, is known to work. The system may not terminate when more complex equations are entered.

- `pred` ⟨ident⟩`/`n seq⟨ident⟩`.`

  `pred` p`/`n $i_1, \ldots, i_n$`.` declares a new predicate p, of arity n, with special properties described by $i_1, \ldots, i_n$. Currently, the allowed elements for $i_1, \ldots, i_n$ are `block` and `memberOptim`. See the section on the Horn clause input for more details on the effect of these properties.

  All predicates must be declared by such a declaration before being used. The predicates `attacker`, `mess`, `ev`, and `evinj` are reserved and cannot be declared.

- `query` ⟨query⟩`.`

  This declaration tells the system which properties we want to prove. Its syntax is given in Figure 3.

  - A term M is a query can contain as usual constructor applications and variables, but also constructs $a[\ldots]$ or simply a that designate names created by the restriction `new` a. $a[]$ designates any name created by the restriction `new` a. $a[v_1 = M_1, \ldots, v_n = M_n]$ designates any name created by the restriction `new` a when the variables $v_1, \ldots, v_n$ have value $M_1, \ldots, M_n$ respectively. These variables must be in scope at the considered restriction. A special variable `!`n corresponds to the session identifier of the n-th replication, starting from the top of the process with $n = 1$ for the first replication. Session identifiers should only be bound to variables. By using the same variable several times, one can express that two names should be created in the same copy of the process.

    Note that, to avoid ambiguities, several restrictions in the processes should not create the same name. (When several restrictions create the same name and one tries to refer to this name in a query, an error message is displayed.)

    Also note that one should be careful of not mixing names and variables: An identifier that is defined by a restriction represents any name defined at that restriction (possibly different names for different occurrences), a free name of the process represents that name, while other identifiers are variables that represent any term (the same term for all occurrences of the same variable).

    A term M in a query must not contain destructors.

  - Facts in queries can be $p : M_1, \ldots, M_n$ [`phase` n], with the following meanings:
    - `attacker:`M means that the attacker may have M in some phase (M is not secret).
    - `attacker:`M `phase` n means that the attacker may have M in phase n.
    - `mess:`M, N means that the message N may be sent on channel M in the last phase.
    - `mess:`M, N `phase` n means that the message N may be sent on channel M in phase n.
    - `ev:`$f(M_1, \ldots, M_n)$ means that the event `event` $f(M_1, \ldots, M_n)$ may be executed. There must exist some `event` $f(M'_1, \ldots, M'_n)$ instruction with the same function symbol f in the process.
    - `evinj:`$f(M_1, \ldots, M_n)$ means that the event `event` $f(M_1, \ldots, M_n)$ may be executed, and that furthermore we want to prove injective correspondences for this event (see examples below).
    - $p : M_1, \ldots, M_n$, where p is a user-defined predicate (see the `pred` and `clauses` declarations), means that the corresponding fact is true.

    Note that the `phase` indication is not allowed for `ev`, `evinj`, and user-defined predicates. One can also use the facts M `=` N and M `<>` N.

  - The elementary query ⟨realquery⟩ can be F `==>` ϕ where ϕ is formed from conjunctions and disjunctions of facts. The query is true if and only if, when F is true, then ϕ is true. The query can also be a fact F (see ⟨query⟩): the answer can be that `not` F is true, i.e. the fact F is false, or that `not` F cannot be proved, when the system finds a (possibly false) attack that would make F true. (This query is a shorthand for F `==> false`.) Here are some examples:
    - `query attacker:`M determines whether the attacker may have M. `not attacker:`M is true when M is secret.
    - `query ev:`$f(M_1, \ldots, M_n)$ determines whether the event `event` $f(M_1, \ldots, M_n)$ may be executed. `not ev:`$f(M_1, \ldots, M_n)$ is true when the event `event` $f(M_1, \ldots, M_n)$ can never be executed.

* `query ev:`$f(x_1, \ldots, x_n)$ `==> ev:`$f'(x_1, \ldots, x_n)$ is non-injective agreement: it is true when, if the event $f(x_1, \ldots, x_n)$ has been executed, then the event $f'(x_1, \ldots, x_n)$ must have been executed (before the event $f(x_1, \ldots, x_n)$).
* `query evinj:`$f(x_1, \ldots, x_n)$ `==> evinj:`$f'(x_1, \ldots, x_n)$ is injective agreement: it is true when, for each executed event $f(x_1, \ldots, x_n)$, there exists a distinct executed event $f'(x_1, \ldots, x_n)$ (and $f'(x_1, \ldots, x_n)$ is executed before $f(x_1, \ldots, x_n)$).
* `query evinj:`$f(M_1)$ `==> evinj:`$f'(M_2)$ `&` `ev:`$f''(M_3)$ is true if and only if for each executed event $f(M_1)$ there exists a distinct executed event $f'(M_2)$ and an executed event $f''(M_3)$. (The event $f''(M_3)$ can be the same for several different events $f(M_1)$, since it is marked with `ev` and not `evinj`.)
* `query evinj:`$f(M_1)$ `==> evinj:`$f'(M_2)$ `|` `ev:`$f''(M_3)$ is true if and only if for each executed event $f(M_1)$ either there exists a distinct executed event $f'(M_2)$ or an event $f''(M_3)$ has been executed.

Note that using `evinj` or `ev` before the arrow `==>` does not change the meaning of the query. It is important only after the arrow.

We can also use *nested queries*: queries in which some of the events after the arrow `==>` are replaced with queries. For instance, F `==> (`F' `==>` F''`)`. For this query to be interesting, F' and F'' must be events. This query is true if and only if, when F is true, the event F' is executed, and F'' is executed *before* F'. (In contrast, the query F `==>` F' `&` F'' would not order F' and F''.) If F is also an event, F' is executed before F.

We can use more complex queries in this style, such as

$$F_0 \texttt{ ==> (}F_1 \texttt{ ==> (}F_2 \texttt{ ==> (}F_3 \texttt{ ==> } F_4\texttt{)))}$$

which is true if and only if, when F_0 is true, F_4, F_3, F_2, F_1 have been executed in that order, or

$$F_0 \texttt{ ==> (}F_1 \texttt{ ==> } F_2\texttt{) & (}F_3 \texttt{ ==> } F_4\texttt{)}$$

which is true if and only if, when F_0 is true, F_2 has been executed before F_1 and F_4 before F_3.

– The full query ⟨query⟩ consists of a list containing as elements queries of the form ⟨realquery⟩ or ⟨gfact⟩ as described above, as well as the following two elements:
  * `putbegin ev:`$f_1, \ldots, f_n$ or `putbegin evinj:`$f_1, \ldots, f_n$ instructs the system to consider active "begin" events the events $f_1(\ldots)$, …, $f_n(\ldots)$. This means that when such an event needs to be executed to trigger another action, a begin fact is going to appear in the hypothesis of the corresponding clause. This is useful when the exact events that should appear in a query are unknown. For instance, with the query `query putbegin ev:`f`; ev:`$f'(x)$, the system generates clauses that conclude `end:`$f'(M)$, and by manual inspection of the facts `begin:`$f(M')$ that occur in their hypothesis, one can infer the full query:

    $$\texttt{query ev:}f'(\ldots) \texttt{ ==> ev:}f(\ldots).$$

    When using `evinj:`, the activated begin events contain an environment that can be used to prove injective correspondences.
    (This way of writing queries simulates what happened in older versions of ProVerif, up to version 1.09.)
  * `let` $x = M$ binds the variable x to the term M. This is especially useful to designate several times the same name. For example, `query let x = a[]; attacker:f(x,x)` determines whether the attacker may have $f(x, x)$ where x is any name created by a restriction `new a`. In contrast, `query attacker:f(a[],a[])` determines whether the attacker may have $f(x, y)$ where x and y are (possibly different) names created by a restriction `new a`. A variable bound by `let` $x = M$ must be bound before being used in the following of the query.

All queries of the list included in a single `query` declaration are evaluated by building one set of clauses and performing resolution on it, while different `query` declarations are evaluated by rebuilding a new set of clauses from scratch. So the way queries are grouped influences the sharing of work between different queries, so the speed of the system. The main idea is that one should group queries that involve the same events, but separate queries that involve different events, because the more events appear in the query, the more complex the generated clauses are, which can slow down the system considerably, especially on complex examples. If one does not want to optimize, one can simply put a single query in each `query` declaration.

- `noninterf` seq⟨interfspec⟩`.`
  where ⟨interfspec⟩ ::= ⟨ident⟩ [`among (`seq⟨term⟩`)`)]

  `noninterf` n_1 `among` $(S_1), \ldots, n_k$ `among` (S_k)`.` tells the system to prove strong secrecy for the secrets $n_1, \ldots, n_k$. That is, the system shows that several versions of the given process that differ by their values of $n_1, \ldots, n_k$ are bisimilar (therefore they are testing equivalent, observationally equivalent, ... – see [8] for more details). When the `among` (S_i) indication is present, it means that n_i can take its values only inside S_i. When it is absent, n_i can take any value not containing bound names (or private free names).

  Note that the `let ... suchthat` construct is incompatible with the test of strong secrecy. What the solver does in this case cannot be done when the input is given under the form of Horn clauses (because the simplifications done by the system are sound only for particular clauses that correspond to those generated from a process; they do not make sense for general clauses).

- `weaksecret` ⟨ident⟩`.`

  `weaksecret` n tells the system to check whether an attacker guessing the value of n can verify its guess offline. This is useful when n is a weak secret, such as a password, that an attacker could guess by exhaustive enumeration.

- `nounif` ⟨gfactformat⟩[/n] [b]`.` where b = `;`⟨ident⟩ `=` ⟨gtermformat⟩`; ... ;`⟨ident⟩ `=` ⟨gtermformat⟩

  `nounif` F[/n] modifies the selection of facts to be resolved upon, to avoid resolving facts that match F. A fact F' matches F if and only if $F' = \sigma F$ for some substitution σ that maps variables always marked with a star `*` to any term and variables that occur at least once without star to a variable. The optional integer n indicates how much we should avoid resolution on facts that match F: the greater n, the more such resolutions will be avoided.

  The only supported facts F are `attacker:`M [`phase` n], `mess:`M, N [`phase` n], and $p : M_1, \ldots, M_n$ when p is a user-defined predicate.

  The fact F can contain as usual constructor applications and variables, but also constructs $a[\ldots]$, as in queries. (See the declaration `query` above.)

  `nounif` F [/n]`;`$x_1 = M_1$`; ... ;`$x_n = M_n$ corresponds to `nounif` $F\{M_n/x_n\} \ldots \{M_1/x_1\}$. ($x_j$ may occur in M_i only when $i > j$). This construct is especially useful to designate several times the same name: for example `nounif attacker:f(x,x);x = a[]` prevents resolution on `attacker` : `f(a[`$\mathtt{M}_1, \ldots, \mathtt{M}_m$`], a[`$\mathtt{M}_1, \ldots, \mathtt{M}_m$`])` while `nounif attacker:f(a[],a[])` prevents resolution on `attacker` : `f(a[`$\mathtt{M}_1, \ldots, \mathtt{M}_m$`], a[`$\mathtt{M}'_1, \ldots, \mathtt{M}'_m$`])`.

  In ⟨gfactformat⟩ and ⟨gtermformat⟩, an identifier without arguments a stands for $a()$ when the function a has been defined (it must then have arity 0), for $a[]$ when a restriction `new` a occurs in the process, and for a variable otherwise.

- `elimtrue` ⟨fact⟩ [`where` seq⟨ident⟩ `can fail`]`.`: same as for the Horn clause front-end (to be used with user-declared predicates).

  In addition, the fact is allowed to contain destructors, and the indication "`where` $x_1, \ldots, x_n$ `can fail`" specifies that the variables $x_1, \ldots, x_n$ can take the special value `fail`, which represents the failure of the evaluation of a destructor. When it is omitted, no variable can take the value `fail`. (See the definition of destructors by `reduc ... otherwise` for more details.)

- `not` ⟨gfact⟩ [*b*]`.` where *b* = `;`⟨ident⟩ `=` ⟨gterm⟩`;` ... `;`⟨ident⟩ `=` ⟨gterm⟩

  `not` *F* adds the assumption that *F* is not derivable (or all instances of *F* when *F* contains variables). This speeds up the solving process. At the end of the solving process, the system checks that *F* is indeed not derivable. If it is not, the proof fails with an error message. The only supported facts *F* are `attacker:`*M* [`phase` *n*], `mess:`*M*, *N* [`phase` *n*], and *p* : *M*$_1$, ..., *M*$_n$ when *p* is a user-defined predicate.

  For `not attacker:`*M* [`phase` *n*], when `phase` *n* is present, it means that *M* is secret in phases up to phase *n*. When `phase` *n* is absent, it means that *M* is secret in all phases. For backward compatibility, the syntax `not` *M* [`phase` *n*] [*b*]`.` is also supported as an abbreviation of `not attacker:`*M* [`phase` *n*] [*b*]`.`.

  The fact *F* can contain as usual constructor applications and variables, but also constructs *a*[...], as in queries. (See the declaration `query` above.)

  `not` *F*`;`*x*$_1$ = *M*$_1$`;` ... `;`*x*$_n$ = *M*$_n$ corresponds to `not` *F*{*M*$_n$/*x*$_n$} ... {*M*$_1$/*x*$_1$}. (*x*$_j$ may occur in *M*$_i$ only when *i* > *j*). This construct is especially useful to designate several times the same name: for example `not attacker:f(x,x);x = a[]` means that `f(x,x)` is secret when `x` is any name created by `new a`, while `not attacker:f(a[],a[])` means that `f(x,y)` is secret when `x` and `y` are any (possible different) names created by `new a`.

  In ⟨gfact⟩, an identifier without arguments *a* stands for *a*() when the function *a* has been defined (it must then have arity 0), for *a*[] when a restriction `new` *a* occurs in the process, and for a variable otherwise.

- [`private`] `free` seq⟨ident⟩`.`

  `free` *i*$_1$, ..., *i*$_n$`.` declares the free names *i*$_1$, ..., *i*$_n$. When the keyword `private` is present, the name is not known by the adversary, whereas by default, it is known by the adversary. When a name occurs free in the process, and is not declared by such a declaration, a warning is displayed. We strongly encourage you to declare all free names of your processes. Indeed, an unexpected free name corresponds in general to a typesetting error, and the warning might become an error in a future release.

- `clauses` (⟨clause⟩`;`)* ⟨clause⟩`.`

  This introduces clauses that define predicates. These predicates can be used in `let` ... `suchthat` processes and in tests `if` ... `then`. Note that there is an implementability condition. Essentially, for each predicate invocation, we bind variables in the conclusion of the clauses that define this predicate and whose position corresponds to bound arguments of the predicate invocation. Then, when evaluating hypotheses of clauses from left to right, all variables of predicates must get bound by the corresponding predicate call. Recursive definitions of predicates are allowed.

  The meaning of clauses is the same as for the Horn clauses input system. In addition, the clauses are allowed to contain destructors, and the indication "`where` *x*$_1$, ..., *x*$_n$ `can fail`" appended to clauses specifies that the variables *x*$_1$, ..., *x*$_n$ can take the special value `fail`, which represents the failure of the evaluation of a destructor. When it is omitted, no variable can take the value `fail`. (See the definition of destructors by `reduc` ... `otherwise` for more details.) The clauses apply only when the arguments of all facts in them do not fail: for facts *p*(*M*$_1$, ..., *M*$_k$), *M*$_1$, ..., *M*$_k$ do not fail, for equalities *M*$_1$ = *M*$_2$ and inequalities *M*$_1$ <> *M*$_2$, *M*$_1$ and *M*$_2$ do not fail.

- `let` ⟨ident⟩ `=` ⟨process⟩`.`

  Defines ⟨ident⟩ as the process ⟨process⟩. ⟨ident⟩ can be used inside the definition of processes. If the process contains free names or variables, they can be bound when ⟨ident⟩ is used. (So, this is a kind of macro-expansion rather than a real definition.)

In the syntax of processes,

- The pattern ⟨ident⟩ matches any term, and binds the given variable identifier to the matched term. The pattern (seq⟨pattern⟩) matches tuples (and each component of the tuple is recursively matched by the given patterns). The pattern *f*(seq⟨pattern⟩) matches terms of the form *f*(*M*$_1$, ..., *M*$_n$) and

the subterms M_i are recursively matched by the given patterns, where f is a data function symbol (see the `data` declaration). When f is not a data function symbol, such a construction is not allowed. The pattern `=`⟨term⟩ matches a term that is equal to the given ⟨term⟩. (This is equivalent to an equality test.)

- Parentheses are just used to clarify associativity of parallel compositions, and which processes are replicated.
- An identifier x must be defined by a previous declaration `let` x `=` P. It is then equivalent to having a copy of P instead of x.
- The replication `!` P executes an unbounded number of copies of P in parallel: $P \mid P \mid P \mid \ldots$.
- The nil process `0` does nothing.
- The restriction `new` a`;`P creates a new name a, then executes P.
- The test `if` f `then` P `else` Q executes P when the fact is true. Otherwise, it executes Q. The process `if` f `then` P is equivalent to `if` f `then` P `else 0`. Note that the predicate calls are subject to an implementability condition (see the `clauses` declaration above). Equality and inequality tests are always implementable.
- The input `in(`c`,`p`);`P inputs a message on channel c, and executes P after matching the input message with p, and binding the variables contained in p. When a message does not match p, it cannot be input by this construct. The channel c can be any term. The process `in(`c`,`x`)` is equivalent to `in(`c`,`x`);0`.
- The output `out(`c`,`M`);`P outputs the message M on the channel c, then executes P. (c can be any term.) The process `out(`c`,`M`)` is equivalent to `out(`c`,`M`);0`.
- The let binding `let` p `=` M `in` P [`else` Q] executes P after matching the term M with the pattern p, and binding the variables contained in p. If the term M does not match the pattern p, the process blocks, or executes Q when the `else` clause is present.
- The binding `let` $x_1, \ldots, x_n$ `such that` f `in` P [`else` Q] binds new variables $x_1, \ldots, x_n$, such that f is true, then executes P. If such a binding is impossible, it executes Q. Note that the predicate calls are subject to an implementability condition (see the `clauses` declaration above). Facts M `<>` N are not allowed in f, because of this implementability condition (they make sense only when all variables are already bound; in this case, using the `else` clause of a `if` is more appropriate).
- The parallel composition P_1 `|` P_2 executes P_1 and P_2 in parallel.
- The event `event` M`;`P emits the event `event`(M), then executes P. The term M must be a function application $f(M_1, \ldots, M_n)$. The function f need not be declared before. When the process P is absent, nothing is executed after the event. Events are not really part of the cryptographic protocol, but are used for authenticity specifications [7] and other properties of protocols. Events can be used to keep track of which steps of the protocol are executed.
- The phase separation command `phase` n`;`P indicates the beginning of phase n. Intuitively, we consider protocols split in several phases, and the instructions under `phase` n are active only during the n-th phase of the protocol. So the process first executes phase 0, that is, it executes all instructions not under `phase` i for $i \geq 1$. Then, when changing from phase 0 to phase 1, it discards all processes that have not reached a `phase` i instruction for $i \geq 1$ and executes the instructions under `phase` 1 but not under `phase` i for $i \geq 2$. More generally, when changing from phase n to phase $n+1$, all processes that have not reached a `phase` i instruction for $i \geq n+1$ are discarded and the instructions under `phase` $n+1$ but not under `phase` i for $i \geq n+2$ are executed. The adversary obviously keeps its knowledge when changing phases.

  Phases can be used to model scenarios in which temporality is important, such as when a long-term key is published after some sessions are executed and we want to determine whether the adversary

can then have the session secrets. (The long-term keys are then published in phase 1, while the rest of the protocol is in phase 0.) Similarly, phases can be used to model protocols that reveal a secret at the end of the session.

The phase number must be at least 1. Phases cannot be used with key compromise, `param keyCompromise = approx.` or `param keyCompromise = strict.`, because key compromise introduces itself a 2-phase process.

- The synchronization command `sync` n`;`P introduces a global synchronization, which has some similarity with phases. The global synchronizations must be executed in increasing order. The process waits until all `sync` n commands are reached before executing the synchronization n. More precisely, assuming n is the smallest synchronization number that occurs in the initial process and has not been executed yet, if the initial process contains k commands `sync` n, then the process waits until it reaches exactly k commands `sync` n, then it executes the synchronization n and continues after the `sync` n commands. So, in contrast to phases, processes are never discarded by synchronization, but the process may block in case some synchronizations cannot be reached or are discarded for instance by a test that fails above them.

  The synchronization number must be at least 1. Synchronizations `sync` n cannot occur under replications. Synchronizations cannot be used with phases or with key compromise. Synchronizations can help proving equivalences with `choice`, because they allow swapping data between processes at the synchronization points.

Two cases have to be distinguished:

- The terms in the process never contain `choice`. The process defines one pi calculus process, and we can ask the various queries (`query`, `noninterf`, `weaksecret`).
- The terms in the process contain `choice`. The process in fact defines two pi calculus processes: one process in which the first argument of `choice` is used, and one in which the second one is used. The verifier then tries to show the observational equivalence of these two processes [9]. The queries `query`, `noninterf`, `weaksecret`, and key compromise cannot be used. (The keyword `diff` is also accepted as a synonym for `choice`.)

6 Output of the system

The system gives an output of the following form:

```
Starting rules:
Rule 10: attacker:c[]
Rule 9: attacker:k -> attacker:host(k)
...
Completing...
Completed rules:
attacker:encrypt(secretB[],k[Kas[],Kbs[],Na[],Nb[host(Kas[]),Na[]]])
attacker:v147 & attacker:v148 -> attacker:encrypt(v148,k[Kas[],v147,Na[],v148])
...
ok, secrecy assumption verified: fact unreachable attacker:Kbs[]
...
goal unreachable: attacker:secretB[]
...
```

First, it displays the Horn clauses representing the protocol. If you use the Horn clauses input, these are the clauses you entered. If you use the pi calculus input, these clauses are the result of a translation of the process, described in [2]. This translation uses mainly two predicates `attacker:`M meaning that the adversary may have M, and `mess:`C`,`M meaning that the message M may be sent on channel C. It uses moreover a predicate `end` and a blocking predicate `begin` for proofs involving events. Some other predicates are used for modeling the compromise of session keys. The clauses are numbered. These numbers are used in the following of the output to reference the clauses.

Second, the system completes the clauses, using a resolution-based algorithm. Depending on the `verbose` parameter, it prints all clauses it creates, or only numbers of clauses every 200th clause created.

Third, it outputs the list of clauses obtained after completion (after the words `Completed rules`).

Fourth, if you have given secrecy assumptions, using the declaration `not`, the system checks them. In case the secrecy assumption is not satisfied, it stops immediately.

Fifth, the system checks each goal you have given using `query`. If the fact mentioned on the left of `==>` in the query is derivable, for each found clause that derives it, the system outputs `goal reachable`, and a derivation of the clause from the initial clauses. The derivation is built according to the following format:

- When a part of the derivation is done using one of the rules:

  `rule` n ⟨fact proved by rule n⟩
  ⟨derivation of first hypothesis of rule n⟩
  ...
  ⟨derivation of last hypothesis of rule n⟩

- When reusing an already proved fact:

  `duplicate` ⟨fact⟩

  You should look for the derivation of ⟨fact⟩ somewhere under the `duplicate` line.

- When taking the n-th element of a tuple (In the Horn clauses input system, this happens when you have declared the predicate p `decompData`. In the pi calculus input system, this happens with $p =$ `attacker`.):

  n-`th` $p : M_n$
  ⟨derivation of $p : (M_1, \ldots, M_k)$⟩

  A similar situation happens when you have declared `data` f:

  n-`th` $p : M_n$
  ⟨derivation of $p : f(M_1, \ldots, M_k)$⟩

- When building a tuple (In the Horn clauses input system, this happens when you have declared the predicate p `decompData`. In the pi calculus input system, this happens with $p =$ `attacker`.):

  k-`tuple` $p : (M_1, \ldots, M_k)$
  ⟨derivation of $p : M_1$⟩
  ...
  ⟨derivation of $p : M_k$⟩

  A similar situation happens when you have declared `data` f:

  f-`tuple` $p : f(M_1, \ldots, M_k)$
  ⟨derivation of $p : M_1$⟩
  ...
  ⟨derivation of $p : M_k$⟩

- When proving $p : x$ (in the Horn clauses input system, when you have declared the predicate p `elimVar`; in the pi calculus input system, when $p =$ `attacker`.)

  `any` $p : x$

Note that the derivations of inequalities of terms are omitted.

At the end, it concludes with `RESULT Query ...is true.` or `RESULT Query ...cannot be proved.`.

References

[1] M. Abadi and B. Blanchet. Computer-assisted verification of a protocol for certified email. In R. Cousot, editor, *Static Analysis, 10th International Symposium (SAS'03)*, volume 2694 of *Lecture Notes in Computer Science*, pages 316–335, San Diego, California, June 2003. Springer.

[2] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.

[3] M. Abadi and B. Blanchet. Computer-assisted verification of a protocol for certified email. *Science of Computer Programming*, 58(1–2):3–27, Oct. 2005. Special issue SAS'03.

[4] M. Abadi, B. Blanchet, and C. Fournet. Just Fast Keying in the pi calculus. In D. Schmidt, editor, *Programming Languages and Systems: 13th European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 340–354, Barcelona, Spain, Mar. 2004. Springer.

[5] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 140–154, Aix-en-Provence, France, June 2005. IEEE.

[6] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[7] B. Blanchet. From secrecy to authenticity in security protocols. In M. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359, Madrid, Spain, Sept. 2002. Springer.

[8] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[9] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, IL, June 2005. IEEE Computer Society.

[10] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science*, 333(1-2):67–90, Mar. 2005. Special issue FoSSaCS'03.