Cryptographic Protocol Verifier
User Manual

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

October 22, 2003

1 Introduction

This manual describes the input syntax and output of our cryptographic protocol verifier. It does not describe the internal algorithms used in the system. These algorithms have been described in various research papers [3, 1, 4, 2, 5], that can be downloaded on


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 [3]. The second one is in the form of a process in an extension of the pi calculus, described in [1]. 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 or numbers. 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 and number of characters from the beginning of the file). Please use your text editor to find the position of the error.
\[ \text{rule} ::= [(\text{fact} \& \ )^* \text{fact} \rightarrow \text{fact}] \]
\[ \text{fact} ::= \text{ident} : \text{seq(\text{term})} \]
\[ | \text{term} \leftrightarrow \text{term} \]
\[ \text{term} ::= \text{ident} \text{seq(\text{term})} \]
\[ | \text{ident}[\text{seq(\text{term})}] \]
\[ | \text{seq(\text{term})} \]
\[ | \text{idem} \]
\[ \text{factformat} ::= \text{ident} : \text{seq(\text{termformat})} \]
\[ \text{termformat} ::= \text{ident} \text{seq(\text{termformat})} \]
\[ | \text{idem}[\text{seq(\text{termformat})}] \]
\[ | \text{seq(\text{termformat})} \]
\[ | \text{idem} \]
\[ | *\text{idem} \]

where seq(X) is a sequence of X: seq(X) = [(X),]^*(X)] = (X), \ldots, (X). (The sequence can be empty, it can be one element \langle X \rangle, or it can be several elements \langle X \rangle separated by commas.)

Figure 1: Grammar of facts and rules

3 Input as Horn clauses

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

\[ ./\text{analyzer} \text{ (filename)} \]

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 [3], this can be used to determine secrecy properties of protocols: if a certain fact cannot be derived from the rules, 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.

The keywords of this input system are anytrue, blocking, data, equation, fun, not, nounif, param, pred, query, reduc, and tupleinv. The keywords blocking, anytrue, and tupleinv are deprecated, please use pred instead.
The input file consists of a list of declarations, followed by the keyword `reduc` and a list of rules:

\[
\text{(declaration)}^* \text{ reduc (\text{rule});)}^* \text{ (rule)}.
\]

The syntax of facts and rules 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 \(\langle\) and \(\rangle\) represents non-terminals.

Declarations can be any of the following:

- **param** \(\langle\text{name}\rangle = \langle\text{value}\rangle\).
  
  This declaration sets the value of configuration parameters. The following cases are supported:

  - **param verbose = no.**
    
    \[\text{param verbose = rules.}\]
    
    Display the number of rules every 200 rule created during the solving process (\textbf{no}) or display each rule created during the solving process (\textbf{rules}).

  - **param maxDepth = none.**
    
    \[\text{param maxDepth = n.}\]
    
    Do not limit the depth of terms (\textbf{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.**
    
    \[\text{param maxHyp = n.}\]
    
    Do not limit the number of hypotheses of rules (\textbf{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 rules, 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 stopTerm = true.**
    
    \[\text{param stopTerm = false.}\]
    
    Display a warning and wait for user answer when the system thinks the solving process will not terminate (\textbf{true}), or go on as if nothing had happened (\textbf{false}).

  - **param selFun = TermMaxsize.**
    
    \[\text{param selFun = Term.}\]
    
    \[\text{param selFun = NounifsetMaxsize.}\]
    
    \[\text{param selFun = Nounifset.}\]

3
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 `redundancyElim` = simple.
  - param `redundancyElim` = no.
  - param `redundancyElim` = best.

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

- param `redundantHypElim` = false.
  - param `redundantHypElim` = true.

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

This replacement is done only when `redundantHypElim` = true, since 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.

In the above list, the default value is the first mentioned.

- 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, ..., x_n) ⇔ p : x_1 ∧ ... ∧ p : x_n` for each predicate `p` declared `tupleinv`.

- 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 very naive and preliminary. 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)`.  
  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.

- **pred** `(ident)/n seq(ident)`.  
  `pred p/n i₁,⋯,iₙ.` declares a new predicate `p`, of arity `n`, with special properties described by `i₁,⋯,iₙ`.

The following properties are allowed for `i₁,⋯,iₙ`:

  - **block**: Declares the predicate `p` as a blocking predicate. Blocking predicates should appear only in hypotheses of rules, not in conclusions. Instead of trying to prove facts containing these predicates (which is impossible since no rule implies such facts), the system collects hypotheses containing the blocking predicates necessary to prove the queries. This is useful in particular to prove authenticity [4].

  - **elimVar**: Tells the system to remove the hypothesis `p:x` where `x` is a variable that does not appear elsewhere in a rule, except possibly in inequality facts. Removing hypotheses is certainly correct. Removing such hypotheses attacker:`x` is also complete for proving secrecy or authenticity, because there always exists a value of `x` that makes true both the inequality facts, and attacker:`x`. (Note that `p` must be a unary predicate.)

  - **decompData**: Adds the following rules, where `p` is a unary predicate:

    ```
    p:(x₁,⋯,xₙ) ← (ident):xᵢ  
    p:x₁ & ⋯ & p:xₙ ← p:(x₁,⋯,xₙ)
    ```

    for all `n` and `i ∈ {1,⋯,n}`. Similar rules are added for each function symbol `f` declared data. These rules are treated in a specially optimized way, since they are used in most protocols.

  - **memberOptim**: This is intended to be used when `p` is defined by

    ```
    p:x,f(x,y);  
    p:x,y ← p:x,f(x',y).
    ```
where $f$ is a data constructor. It turns on the following optimization:
$p' : x \land p : M_1, x \land \ldots \land p : M_n, x$ where $p'$ is declared `decompData` and $p$
is declared `memberOptim` is replaced with $p' : x \land p' : M_1 \land \ldots \land p' : M_n$
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 $p' : M_1, \ldots, p' : M_n$, 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 $p_1(M, x) \land p_2(M', x)$ is never true: for it to be true, $x$ should be at the same
time $f_1(\omega)$ and $f_2(\omega)$.)

- `com`: Tells the system to remove rules containing $p : M, M'$ where $M$
is not a variable or a name. ($p$ must be a predicate of arity 2.) This is useful in particular for the predicate `mess(M, M')`, in which $M$
represents a channel: communications can occur only when channels are names.

- `not (fact)`. Adds a secrecy assumption, saying that (fact) cannot be proved from
the rules. Then the system can remove all rules that contain fact in
their hypotheses. (These rules 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 rules. 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: $f(M_1, \ldots, M_n)$. 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: $a[M_1, \ldots, M_n]$. 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 $f(M_1, \ldots, M_n)$, name
applications $a[M_1, \ldots, M_n]$, tuples $(M_1, \ldots, M_n)$, or identifiers $x$ that can
be used for variables or constructors without parameters.

Facts can be the application of a predicate to terms $p:M_1, \ldots, M_n$. (This
is written this way and not $p(M_1, \ldots, M_n)$ only for historical reasons.) They
can also be $M \leftrightarrow M'$, meaning $M$ is different from $M'$. Blocking facts and
inequalities are allowed only in hypotheses of rules, not in conclusions, queries
query (fact), and secrecy assumptions not (fact).

Rules can be $F_1 \land \ldots \land F_n \Rightarrow F$, meaning $F_1$ and \ldots and $F_n$ implies $F$.
They can also be simply $F$, meaning that $F$ is true, without any hypothesis.

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

4 Input as process in extension of the pi calculus

To give a pi calculus process as input to the analyzer, you have to add the
command line option `-in pi`. You can then run the analyzer by:

```
./analyzer -in pi (filename)
```

where (filename) references a file containing the process, in the format explained
below.

The keywords of this input system are among, and, authquery, begin,
channelquery, data, endquery, else, end, equation, free, fun, if, in, let,
new, noninterf, not, out, param, phase, private, process, query, reduc,
suchthat, and then.

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 Figure 2, using the same conven-
tions as in Section 3.

Declarations can be any of the following:

- param (name) = (value).

This declaration sets the value of configuration parameters. The cases
mentioned in Section 3 are supported, as well as the following ones:

- param attacker = active.
  param attacker = passive.
  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 ses-
  sion keys are not a priori compromised. Otherwise, it is assumed that
  some session keys are compromised (known by the adversary). Then
\[\langle \text{term} \rangle ::= \langle \text{ident} \rangle \langle \text{seq}(\text{term}) \rangle \]
\[| \quad \langle \text{seq}(\text{term}) \rangle \]
\[| \quad \langle \text{ident} \rangle \]

\[\langle \text{fact format} \rangle ::= \langle \text{ident} \rangle : \langle \text{seq}(\text{term format}) \rangle \]

\[\langle \text{term format} \rangle ::= \langle \text{ident} \rangle \langle \text{seq}(\text{term format}) \rangle \]
\[| \quad \langle \text{seq}(\text{term format}) \rangle \]
\[| \quad \langle \text{ident} \rangle \]
\[| \quad \ast(\text{ident}) \]

\[\langle \text{pattern} \rangle ::= \langle \text{ident} \rangle \]
\[| \quad \langle \text{seq}(\text{pattern}) \rangle \]
\[| \quad \langle \text{ident} \rangle \langle \text{seq}(\text{pattern}) \rangle \]
\[| \quad =\langle \text{term} \rangle \]

\[\langle \text{rule} \rangle ::= \langle \{ \langle \text{fact} \rangle \& \ast \langle \text{fact} \rangle \rightarrow \} \langle \text{fact} \rangle \rangle \]

\[\langle \text{fact} \rangle ::= \langle \text{ident} \rangle \langle \text{seq}(\text{term}) \rangle \]
\[| \quad \langle \text{term} \rangle \langle \text{term} \rangle \]
\[| \quad \langle \text{term} \rangle = \langle \text{term} \rangle \]

\[\langle \text{process} \rangle ::= \langle \{ \langle \text{process} \rangle \} \rangle \]
\[| \quad \langle \text{ident} \rangle \]
\[| \quad \langle \text{ident} \rangle \langle \text{process} \rangle \]
\[| \quad 0 \]
\[| \quad \text{new} \langle \text{ident} \rangle ; \langle \text{process} \rangle \]
\[| \quad \text{if} \langle \text{fact} \rangle \text{then} \langle \text{process} \rangle \langle \text{else} \langle \text{process} \rangle \rangle \]
\[| \quad \text{in}(\langle \text{term} \rangle, \langle \text{pattern} \rangle)[; \langle \text{process} \rangle] \]
\[| \quad \text{out}(\langle \text{term} \rangle, \langle \text{term} \rangle)[; \langle \text{process} \rangle] \]
\[| \quad \text{let} \langle \text{pattern} \rangle = \langle \text{term} \rangle \text{in} \langle \text{process} \rangle \langle \text{else} \langle \text{process} \rangle \rangle \]
\[| \quad \text{let} \text{seq}(\langle \text{ident} \rangle) \text{suchthat} \langle \text{fact} \rangle \text{in} \langle \text{process} \rangle \langle \text{else} \langle \text{process} \rangle \rangle \]
\[| \quad \langle \text{process} \rangle \langle \text{process} \rangle \]
\[| \quad \text{begin} \langle \text{term} \rangle [; \langle \text{process} \rangle] \]
\[| \quad \text{end} \langle \text{term} \rangle [; \langle \text{process} \rangle] \]
\[| \quad \text{phase} \langle \text{term} \rangle [; \langle \text{process} \rangle] \]

Figure 2: Grammar of processes
the system determines whether the secrets of other sessions can be obtained by the adversary. The modeling with `param keyCompromise = approx.` is more approximate than with `param keyCompromise = strict.`; that is, the chances of finding a false attack are greater with `param keyCompromise = approx.`.

- `param injectiveAg = false.`
  `param injectiveAg = true.`
  Sets whether the system should prove injective (`param injectiveAg = true.`) or non-injective correspondences (`param injectiveAg = false.`). When proving injective correspondences, some parameters are added to begin and end events to make it possible to detect in which sessions events are executed. The first parameter of end events is then a session identifier, and the second parameter of begin events is an environment. The proof of injectivity requires that the session identifier of the end event occurs in the environment of begin events (see [4] for the exact criterion). In particular, authenticity queries (see authquery) prove either injective or non-injective agreement depending on this parameter. Non-injective agreement means that if an event `end M` has been executed, then `begin M` has also been executed.Injective agreement means that if `end M` has been executed `n` times then `begin M` has been executed at least `n` times.

- `param movenew = true.`
  `param movenew = false.`
  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 channels = terms.`
  `param channels = names.`
  Sets whether communications are allowed on channels that are any term (`param channels = terms.`) or only names (`param channels = names.`). In the second case, rules that contain a fact corresponding to a communication on a channel that is not a variable or a name are removed.

- `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.

- `[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:

  \[
  \text{reduc } f(M_1, \ldots, M_n) = M_0; \\
  f(M'_1, \ldots, M'_n) = M'_0; \\
  \vdots \\
  f(M''_1, \ldots, M''_n) = M''_0.
  \]

  declares the destructor f, of arity n, with the given reduction 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 several rules can be applied, the process chooses one possibility non-deterministically (but the analysis considers all possibilities). When no rule can be applied, the destructor is not defined; the process blocks. The terms \( M_0, \ldots, M_n, M'_0, \ldots, M'_n, \ldots \) must contain only variables and constructors.

- **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 very naive and preliminary. 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 \texttt{block}, \texttt{elimVar}, \texttt{decompData}, and \texttt{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.

- **query (term) [phase n].**
  
  **query \( M \)** tells the system to determine whether \( M \) is secret or not. If \( M \) contains variables, it determines which instances of \( M \) are secret. When
$M$ contains names created by restrictions, it determines for which instances of the name $M$ is secret. Note that, to avoid ambiguities, several restrictions in the processes should not create the same name. (When several restrictions create the same name, the query in fact concerns one restriction, chosen randomly by the system...). $M$ must not contain destructors. When \textbf{phase} $n$ is present, it determines whether $M$ is secret in phase $n$. When \textbf{phase} $n$ is absent, it determines whether $M$ is secret in the last phase (so in all phases).

- \textbf{channelquery} $\langle \text{term} \rangle, \langle \text{term} \rangle [\text{phase} \ n]$.
  channelquery $M_1, M_2$ tells the system to determine whether $M_2$ may be sent on channel $M_1$ or not. If $M_1$ or $M_2$ contain variables, it determines which instances of $M_2$ may be sent on instances of $M_1$. When $M_1$ or $M_2$ contain names created by restrictions, it determines for which instances of the name the message $M_2$ may be sent on $M_1$. Note that, to avoid ambiguities, several restrictions in the processes should not create the same name. (When several restrictions create the same name, the query in fact concerns one restriction, chosen randomly by the system...). $M_1$ and $M_2$ must not contain destructors. When \textbf{phase} $n$ is present, it determines whether $M_2$ may be sent on channel $M_1$ in phase $n$. When \textbf{phase} $n$ is absent, it determines whether $M_2$ may be sent on channel $M_1$ in the last phase.

- \textbf{endquery} $\langle \text{term} \rangle$.
  endquery $M$ tells the system to determine whether the \textbf{end} event $M$ may be executed or not, and under which conditions, $M$ must be a function application. If $M$ contain variables, it determines which instances of the \textbf{end} event $M$ may be executed. When $M$ contains names created by restrictions, it determines for which instances of the name the event $M$ may executed. Note that, to avoid ambiguities, several restrictions in the processes should not create the same name. (When several restrictions create the same name, the query in fact concerns one restriction, chosen randomly by the system...). $M_1$ and $M_2$ must not contain destructors.

- \textbf{authquery} $\langle \text{ident} \rangle/n$.
  authquery $f/n$. tells the system to prove the agreement: if \textbf{end}(f(M_1, ..., M_n)) has been executed then begin(f(M_1, ..., M_n)) has been executed. $n$ must be the arity of $f$. The function symbol $f$ need not be declared before.

- \textbf{noninterf} seq(interfspec).
  where (interfspec) ::= (ident)[among (seq(term))]
  noninterf $n_1$ among $(S_1), ..., n_k$ among $(S_k)$. tells the system to prove strong secrecy for the secrets $n_1, ..., n_k$. That is, the system shows that several versions of the given process that differ by their values of $n_1, ..., n_k$
are bisimilar (therefore they are testing equivalent, observationally equivalent, ...). 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).

- **nounif** (factformat).

  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.

- **not** (term) [phase n].

  not \( M \) adds the secrecy assumption that \( M \) is secret (or all instances of \( M \) when \( M \) contains variables or names). This speeds up the solving process. At the end of the solving process, the system checks that \( M \) is indeed secret. If it is not, the proof fails with an error message. Note that, to avoid ambiguities, several restrictions in the processes should not create the same name. (When several restrictions create the same name, the secrecy assumption in fact concerns one restriction, chosen randomly by the system...) \( M \) must not contain destructors. 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.

- **[private] free** seq(ident).

  free \( i_1, \ldots, i_n \). declares the free names \( i_1, \ldots, i_n \). 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** ((rule);)* (rule).

  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.

- **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 \( f \text{seq}(\text{pattern}) \) matches terms of the form \( f(M_1, \ldots, M_n) \) 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 \( =\text{term} \) matches a term that is equal to the given \( \text{term} \). (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 \( \text{let } x = P \). 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 \( \text{new } a;P \) creates a new name \( a \), then executes \( P \).

- The test \( \text{if } f \text{ then } P \text{ else } Q \) executes \( P \) when the fact is true. Otherwise, it executes \( Q \). The process \( \text{if } f \text{ then } P \text{ is equivalent to } \text{if } f \text{ then } P \text{ else } 0 \). 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 \( \text{in}(c,p);P \) 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 \( \text{in}(c,x) \) is equivalent to \( \text{in}(c,x);0 \).

- The output \( \text{out}(c,M);P \) outputs the message \( M \) on the channel \( c \), then executes \( P \). (\( c \) can be any term.) The process \( \text{out}(c,M) \) is equivalent to \( \text{out}(c,M);0 \).
• 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 vari-
ables \( x_1, \ldots, x_n \), 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
\( M \leftrightarrow N \) 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 \( P_1 \mid P_2 \) executes \( P_1 \) and \( P_2 \) in parallel.

• The begin event begin \( M \); \( P \) emits the event begin(\( M \)), 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 begin event. Begin and end events are not really part of
the cryptographic protocol, but are used for authenticity specifications [4].
Such a begin event can be used to keep track of which steps are executed:
when the event begin \( M \) must be executed to obtain a certain fact, then
begin \( M \) is going to appear in the hypothesis of rules that prove this fact.

• The end event end \( M \); \( P \) emits the end event end(\( M \)), 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 end event.

• The phase separation command phase \( n \); \( P \) indicates the beginning of
phase \( n \). It can be understood as a global synchronization command. The
process first executes all instructions not under phase 1. Then it discards
all these instructions and executes the instructions under phase 1 but not
under phase 2, and so on. 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 between 1 and 10. Phases cannot be used with
key compromise, param keyCompromise = approx. or param keyCompromise
= strict., because key compromise introduces itself a 2-phase process.
5 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 [1]. 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 authenticity proofs. Some other predicates are used for modeling the
promise of session keys. The rules are numbered. These numbers are used
in the following of the output to reference the rules.

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

Third, it outputs the list of rules 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, channel,query,
or endquery. If the mentioned term is secret, or the mentioned fact is under-
viable, it outputs goal unreachable. Otherwise, it outputs goal reachable,
and a proof of the fact from the rules. The proof is built according to the
following format:

- When a part of the proof is done using one of the rules:
  
  rule n (fact proved by rule n)
  (proof of first hypothesis of rule n)
  ...
  (proof of last hypothesis of rule n)
• When reusing an already proved fact:

\[ \text{duplicate (fact)} \]

You should look for the proof 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 \texttt{tupleinv} \(p\). In the pi calculus input system, this happens with \(p = \text{attacker}\).):

\[
n\text{-th } p : M_n
\]

\[
\{\text{proof of } p : (M_1, \ldots, M_k)\}
\]

A similar situation happens when you have declared \texttt{data} \(f\):

\[
n\text{-th } p : M_n
\]

\[
\{\text{proof of } p : f(M_1, \ldots, M_k)\}
\]

• When building a tuple (In the Horn clauses input system, this happens when you have declared \texttt{tupleinv} \(p\). In the pi calculus input system, this happens with \(p = \text{attacker}\).):

\[
k\text{-tuple } p : (M_1, \ldots, M_k)
\]

\[
\{\text{proof of } p : M_1\}
\]

\[
\ldots
\]

\[
\{\text{proof of } p : M_k\}
\]

A similar situation happens when you have declared \texttt{data} \(f\):

\[
f\text{-tuple } p : f(M_1, \ldots, M_k)
\]

\[
\{\text{proof of } p : M_1\}
\]

\[
\ldots
\]

\[
\{\text{proof of } p : M_k\}
\]

• When proving \(p : x\) (in the Horn clauses input system, when you have declared \texttt{anytrue} \(p\); in the pi calculus input system, when \(p = \text{attacker}\).)

\[\text{any } p : x\]

Note that the proofs of blocking hypotheses and inequalities of terms are omitted.

The output is slightly different for \texttt{authquery}. In the last part of the output, for each \texttt{authquery} \(f/n\), it outputs -- \textit{Authenticity} \(f\), then the proof concerns rules of the form \(H \rightarrow \text{end}:f(\ldots)\). When such a rule is derivable, it is printed: * \texttt{goal}: \(H \rightarrow \text{end}:f(\ldots)\). If \(H\) contains a \texttt{begin} hypothesis
that matches the end: conclusion, the system then prints goal in fact not reachable, and continues. Otherwise, it prints the proof of the given rule. This proof corresponds to an attack against the protocol. (In rare cases, it can be a false attack.) At the end, it concludes with RESULT Agreement proof on $f$ failed or RESULT Agreement proof on $f$ succeeded.

References


