CryptoVerif
Computationally Sound, Automatic Cryptographic Protocol Verifier
User Manual

Bruno Blanchet and David Cadé
INRIA Paris-Rocquencourt, France

November 13, 2015

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 research papers [2, 1, 3, 4] that can be downloaded at

`http://prosecco.gforge.inria.fr/personal/bblanche/publications/index.html`.

The goal of our protocol verifier is to prove security properties of protocols in the computational model. The input file describes the considered security protocol, the hypotheses on the cryptographic primitives used in the protocol, and security properties to prove.

2 Command Line

The syntax of the command line is as follows:

`./cryptoverif` [*options*] ⟨filename⟩

where ⟨filename⟩ is the name of the input file. The options can be:

- `-in` ⟨frontend⟩: Chooses the frontend to use by CryptoVerif. ⟨frontend⟩ can be either `channels` (the default) or `oracles`. The `channels` frontend uses a calculus inspired by the pi calculus, described in Section 3 and in [2, 1]. The `oracles` frontend uses a calculus closer to cryptographic games, described in Section 4 and in [3, 4]. By default, CryptoVerif uses the `oracles` frontend when the input ⟨filename⟩ ends with `.ocv`, and otherwise it uses the `channels` frontend.

- `-lib` ⟨filename⟩: Sets the name of the library file (by default `default`) which is loaded by the system before reading the input file. In the `channels` front-end, the loaded file is ⟨filename⟩`.cvl`; in the `oracles` front-end, it is ⟨filename⟩`.ocvl`. The library file typically contains default declarations useful for all protocols.

- `-tex` ⟨filename⟩: Activates TeX output, and sets the output file name. In this mode, CryptoVerif outputs a TeX version of the proof, in the given file.

- `-impl`: Instead of proving the protocol, generate an implementation in OCaml corresponding to the modules defined in the input file.

- `-o` ⟨directory⟩: If the `-impl` option is given, outputs the implementation files in the given directory.

3 channels Front-end

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. Keywords cannot be used as ordinary identifiers. The keywords are: `channel`, `collision`, `const`, `define`, `defined`, `else`, `eps_find`, `eps_rand`, `equation`, `equiv`, `event`, `event_abort`, `expand`, `find`, `forall`, `fun`, `get`, `if`, `implementation`, `in`, `inj`, `insert`, `length`, `let`, `letfun`, `max`, `maxlength`, `new`, `newChannel`, `orfind`, `otheruses`, `out`, `param`, `Pcoll1rand`, `Pcoll2rand`, `proba`, `process`, `proof`, `query`, `secret`, `secret1`, `set`, `suchthat`, `table`, `then`, `time`, `type`, `yield`.

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`.)

The input file consists of a list of declarations followed by a process:

$$\langle\text{declaration}\rangle^* \texttt{ process } \langle\text{iprocess}\rangle$$

A library file (specified on the command-line by the `-lib` option) consists of a list of declarations. Various syntactic elements are described in Figures 1 and 2. The process describes the considered security protocol; the declarations specify in particular hypotheses on the cryptographic primitives and the security properties to prove.

Processes are described in a process calculus. In this calculus, terms represent computations on bitstrings. Simple terms consist of the following constructs:

- A term between parentheses `(`M`)` allows to disambiguate syntactic expressions.
- An identifier can be either a constant symbol f (declared by `const` or `fun` without argument) or a variable identifier.
- The function application f`(`$M_1, \dots, M_n$`)` applies function f to the result of $M_1, \dots, M_n$.
- The tuple application `(`$M_1, \dots, M_n$`)` builds a tuple from $M_1, \dots, M_n$ (corresponds to the concatenation of $M_1, \dots, M_n$ with length and type indications so that $M_1, \dots, M_n$ can be recovered without ambiguity). This is allowed only for $n \neq 1$, so that it is distinguished from parenthesing.
- The array access x`[`$M_1, \dots, M_n$`]` returns the cell of indexes $M_1, \dots, M_n$ of array x.
- `=`, `<>`, `||`, `&&` are function symbols that represent equality and inequality tests, disjunction and conjunction. They use the infix notation, but are otherwise considered as ordinary function symbols.

Terms contain further constructs `event_abort`, `if`, `find`, `let`, `new` which are similar to the corresponding constructs of output processes but return a bitstring instead of executing a process. They are not allowed to occur in `defined` conditions of `find` and in input channels. The constructs `new` and `event_abort` are not allowed to occur in conditions of `find` or `get`. The construct `find` is also not allowed in conditions of `get`. We refer the reader to the description of processes below for a fully detailed explanation.

- `new` x`:`T`;`M chooses a new random number in type T, stores it in x, and returns the result of M.
- `let` p `=` M `in` M' `else` M'' tries to decompose the term M according to pattern p. In case of success, returns the result of M', otherwise the result of M''.

  The pattern p can be:

  - x[`:`T] variable, possibly with its type. Matches any bitstring (in type T), and stores it in x.
  - f`(`$p_1, \dots, p_n$`)` where the function symbol f is declared `[compos]`. Matches bitstrings M equal to $f(M_1, \dots, M_n)$ for some $M_1, \dots, M_n$ that match $p_1, \dots, p_n$. (The poly-injectivity of f allows us to compute possible values $M_1, \dots, M_n$ of its arguments from the value of M, and to check whether M is equal to the resulting value of $f(M_1, \dots, M_n)$.)

```
⟨identbound⟩ ::= ⟨ident⟩ <= ⟨ident⟩
⟨vartype⟩ ::= ⟨ident⟩:⟨ident⟩
⟨simpleterm⟩ ::= ⟨ident⟩
               | ⟨ident⟩(seq⟨simpleterm⟩)
               | (seq⟨simpleterm⟩)
               | ⟨ident⟩[seq⟨simpleterm⟩]
               | ⟨simpleterm⟩ = ⟨simpleterm⟩
               | ⟨simpleterm⟩ <> ⟨simpleterm⟩
               | ⟨simpleterm⟩ || ⟨simpleterm⟩
               | ⟨simpleterm⟩ && ⟨simpleterm⟩
⟨term⟩ ::= ... (as in ⟨simpleterm⟩ with ⟨term⟩ instead of ⟨simpleterm⟩)
         | new ⟨vartype⟩;⟨term⟩
         | let ⟨pattern⟩ = ⟨term⟩ in ⟨term⟩ [else ⟨term⟩]
         | if ⟨cond⟩ then ⟨term⟩ else ⟨term⟩
         | find[[unique]] ⟨tfindbranch⟩ (orfind ⟨tfindbranch⟩)* else ⟨term⟩
         | event_abort ⟨ident⟩
⟨varref⟩ ::= ⟨ident⟩[seq⟨simpleterm⟩]
           | ⟨ident⟩
⟨cond⟩ ::= defined(seq+⟨varref⟩)  [&& ⟨term⟩]
         | ⟨term⟩
⟨tfindbranch⟩ ::= seq⟨identbound⟩ suchthat ⟨cond⟩ then ⟨term⟩
⟨pattern⟩ ::= ⟨ident⟩
            | ⟨vartype⟩
            | ⟨ident⟩(seq⟨pattern⟩)
            | (seq⟨pattern⟩)
            | =⟨term⟩
⟨event⟩ ::= [inj:]⟨ident⟩[(seq⟨simpleterm⟩)]
⟨queryterm⟩ ::= ⟨queryterm⟩ && ⟨queryterm⟩
              | ⟨queryterm⟩ || ⟨queryterm⟩
              | ⟨event⟩
              | ⟨simpleterm⟩
⟨query⟩ ::= secret ⟨ident⟩
          | secret1 ⟨ident⟩
          | [seq⟨vartype⟩;] event ⟨event⟩ (&& ⟨event⟩)* ==> ⟨queryterm⟩
```

where

- $[M]$ means that M is optional; $(M)^*$ means that M occurs 0 or any number of times.
- 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.)
- seq$^+$⟨X⟩ is a non-empty sequence of X: seq$^+$⟨X⟩ = (⟨X⟩,)*⟨X⟩ = ⟨X⟩, ..., ⟨X⟩. (It can be one or several elements of ⟨X⟩ separated by commas.)

Figure 1: Grammar for terms, patterns, and queries

⟨proba⟩ ::= (⟨proba⟩)
| ⟨proba⟩ + ⟨proba⟩
| ⟨proba⟩ - ⟨proba⟩
| ⟨proba⟩ * ⟨proba⟩
| ⟨proba⟩ / ⟨proba⟩
| max(seq$^+$⟨proba⟩)
| ⟨ident⟩[(seq⟨proba⟩)]
| |⟨ident⟩|
| maxlength(⟨simpleterm⟩)
| length(⟨ident⟩[, seq$^+$⟨proba⟩])
| length((seq⟨ident⟩)[, seq$^+$⟨proba⟩])
| n
| #⟨ident⟩
| eps_find
| eps_rand(T)
| Pcoll1rand(T)
| Pcoll2rand(T)
| time
| time(⟨ident⟩[, seq$^+$⟨proba⟩])
| time(let ⟨ident⟩[, seq$^+$⟨proba⟩])
| time((seq⟨ident⟩)[, seq$^+$⟨proba⟩])
| time(let (seq⟨ident⟩)[, seq$^+$⟨proba⟩])
| time(= ⟨ident⟩[, seq$^+$⟨proba⟩])
| time(!)
| time([n])
| time(&&)
| time(||)
| time(new ⟨ident⟩)
| time(newChannel)
| time(if)
| time(find n)
| time(out [[seq$^+$⟨ident⟩]]⟨ident⟩[, seq$^+$⟨proba⟩])
| time(in n)

⟨fungroup⟩ ::= ⟨ident⟩(seq⟨vartype⟩) [[n]] [[useful_change]] := ⟨term⟩
| ![⟨ident⟩ <=] ⟨ident⟩(new ⟨vartype⟩;)*⟨fungroup⟩
| ![⟨ident⟩ <=] ⟨ident⟩(new ⟨vartype⟩;)*(seq$^+$⟨fungroup⟩)
⟨funmode⟩ ::= ⟨fungroup⟩ [[exist]]
| ⟨fungroup⟩ [all]

⟨channel⟩ ::= ⟨ident⟩
⟨oprocess⟩ ::= ⟨ident⟩
| (⟨oprocess⟩)
| yield
| event ⟨ident⟩[(seq⟨term⟩)] [; ⟨oprocess⟩]
| event_abort ⟨ident⟩
| new ⟨vartype⟩[; ⟨oprocess⟩]
| let ⟨pattern⟩ = ⟨term⟩ [in ⟨oprocess⟩ [else ⟨oprocess⟩]]
| if ⟨cond⟩ then ⟨oprocess⟩ [else ⟨oprocess⟩]
| find[[unique]] ⟨findbranch⟩ (orfind ⟨findbranch⟩)* [else ⟨oprocess⟩]
| insert ⟨ident⟩(seq⟨term⟩) [; ⟨oprocess⟩]
| get ⟨ident⟩(seq⟨pattern⟩) [suchthat ⟨term⟩] in ⟨oprocess⟩ [else ⟨oprocess⟩]
| out(⟨channel⟩, ⟨term⟩)[; ⟨iprocess⟩]
⟨findbranch⟩ ::= seq⟨identbound⟩ suchthat ⟨cond⟩ then ⟨oprocess⟩
⟨iprocess⟩ ::= ⟨ident⟩
| (⟨iprocess⟩)
| 0
| ⟨iprocess⟩ | ⟨iprocess⟩
| ![⟨ident⟩ <=] ⟨ident⟩ ⟨iprocess⟩
| in(⟨channel⟩,⟨pattern⟩)[; ⟨oprocess⟩]

Figure 2: Grammar for probabilities, equivalences, and processes

- `(`$p_1, \ldots, p_n$`)` tuples, which are particular `[compos]` functions encoding unambiguously the values of $p_1, \ldots, p_n$ and their type.
- `=`M' matches a bitstring equal to M'.

When p is a variable, the `else` branch can be omitted (it cannot be executed).

- `if` *cond* `then` M `else` M' is syntactic sugar for `find suchthat` *cond* `then` M `else` M'. It returns the result of M if the condition *cond* evaluates to `true` and of M' if *cond* evaluates to `false`.

- `find` FB_1 `orfind` ... `orfind` FB_m `else` M where $FB_j = u_{j1}$`<=`$n_{j1}, \ldots, u_{jm_j}$`<=`n_{jm_j} `suchthat` $cond_j$ `then` M_j evaluates the conditions $cond_j$ for each j and each value of $u_{j1}, \ldots, u_{jm_j}$ in $[1, n_{j1}] \times \ldots \times [1, n_{jm_j}]$. If none of these conditions is `true`, it returns the result of M. Otherwise, it chooses randomly with (almost) uniform probability one j and one value of $u_{j1}, \ldots, u_{jm_j}$ such that the corresponding condition is `true`, and returns the result of M_j. See the explanation of the `find` process below for more details.

- `event_abort` e executes event e and aborts the game. It is intended to be used in the right-hand side of the definitions of some cryptographic primitives. (See also the `equiv` declaration; events in the right-hand side can be used when the simulation of left-hand side by the right-hand side fails. CryptoVerif is going bound the probability that the event is executed and include it in the probability of success of an attack.)

The calculus distinguishes two kinds of processes: input processes ⟨iprocess⟩ are ready to receive a message on a channel; output processes ⟨oprocess⟩ output a message on a channel after executing some internal computations. When an input or output process is an identifier, it is substituted with its value defined by a `let` declaration. Processes allow parenthesing for disambiguation.

Let us first describe input processes:

- `0` does nothing.

- Q `|` Q' is the parallel composition of Q and Q'.

- `!`i`<=`N Q represents N copies of Q in parallel each with a different value of $i \in [1, N]$. The identifier N must have been declared by `param` N. The identifier i cannot be referred to explicitly in the process; it is used only implicitly as array index of variables defined under the replication `!`i`<=`N. The replication `!`i`<=`N can be abbreviated `!`N.

  When a program point is under replications `!`i_1`<=`N_1, ..., `!`i_n`<=`N_n, the *current replication indexes* at that point are $i_1, \ldots, i_n$.

- The semantics of the input `in(`⟨channel⟩`,`⟨pattern⟩`);`⟨oprocess⟩ will be explained below together with the semantics of the output.

Note that the construct **newChannel** $c; Q$ used in research papers is absent from the implementation: this construct is useful in the proof of soundness of CryptoVerif, but not essential for encoding games that CryptoVerif manipulates.

Let us now describe output processes:

- `yield` yields control to another process, by outputting an empty message on channel *yield*. It can be understood as an abbreviation for `out(`*yield*`,());0`.

- `event` e`(`$M_1, \ldots, M_n$`);`P executes the event e`(`$M_1, \ldots, M_n$`)`, then executes P. Events serve in recording the execution of certain parts of the program for using them in queries. The symbol e must have been declared by an `event` declaration.

- `event_abort` e executes event e and terminates the game. (Nothing can be executed after this instruction, neither by the protocol nor by the adversary.) The symbol e must have been declared by an `event` declaration, without any argument.

- `new` x`:`T`;`P chooses a new random number in type T, stores it in x, and executes P. T must be declared with option `fixed`, `bounded`, or `nonuniform`. Each such type T comes with an associated default probability distribution D_T; the random number is chosen according to that distribution. The time for generated random numbers in that distribution is bounded by `time(new` T`)`.

  - When the type T is `nonuniform`, the default probability distribution D_T for type T may be non-uniform. It is left unspecified. (Notice that random bitstrings with non-uniform distributions can also be obtained by applying a function to a random bitstring choosen uniformly among a finite set of bitstrings, chosen in another type.)
  - When the type T is `fixed`, it consists of the set of all bitstrings of a certain length n. Probabilistic Turing machines can return uniformly distributed random numbers in such types, in bounded time. If T is not marked `nonuniform`, the default probability distribution D_T for T is the uniform distribution.
  - For other `bounded` types T, probabilistic bounded-time Turing machines can choose random numbers with a distribution as close as we wish to uniform, but may not be able to produce exactly a uniform distribution. If T is not marked `nonuniform`, the default probability distribution D_T is such that its distance to the uniform distribution is at most `eps_rand(`T`)`. The distance between two probability distributions D_1 and D_2 for type T is

    $$d(D_1, D_2) = \sum_{a \in T} |\Pr[X_1 = a] - \Pr[X_2 = a]|$$

    where X_i $(i = 1, 2)$ is a random variable of distribution D_i.

    For example, a possible algorithm to obtain a random integer in $[0, m-1]$ is to choose a random integer x' uniformly among $[0, 2^k - 1]$ for a certain k large enough and return $x' \bmod m$. By euclidian division, we have $2^k = qm + r$ with $r \in [0, m-1]$. With this algorithm

    $$\Pr[x = a] = \begin{cases} \frac{q+1}{2^k} & \text{if } a \in [0, r-1] \\ \frac{q}{2^k} & \text{if } a \in [r, m-1] \end{cases}$$

    so

    $$\left|\Pr[x = a] - \frac{1}{m}\right| = \begin{cases} \frac{q+1}{2^k} - \frac{1}{m} & \text{if } a \in [0, r-1] \\ \frac{1}{m} - \frac{q}{2^k} & \text{if } a \in [r, m-1] \end{cases}$$

    Therefore

    $$\begin{aligned} d(D_T, \mathit{uniform}) &= \sum_{a \in T} \left|\Pr[x = a] - \frac{1}{m}\right| = r\left(\frac{q+1}{2^k} - \frac{1}{m}\right) - (m - r)\left(\frac{1}{m} - \frac{q}{2^k}\right) \\ &= \frac{2r(m-r)}{m.2^k} \leq \frac{m}{2^k} \end{aligned}$$

    so we can take `eps_rand(`T`)` $= \frac{m}{2^k}$. A given precision of `eps_rand(`T`)` $= \frac{1}{2^{k'}}$ can be obtained by choosing $k = k' +$ number of bits of m random bits.

    When `ignoreSmallTimes` is set to a value greater than 0 (which is the default), the time for random number generations and the probability `eps_rand(`T`)` are ignored, to make probability formulas more readable.

- `let` p `=` M `in` P `else` P' tries to decompose the term M according to pattern p. In case of success, executes P, otherwise executes P'.

  The pattern p can be:

  - x[`:`T] variable, possibly with its type. Matches any bitstring (in type T), and stores it in x.
  - f`(`$p_1, \ldots, p_n$`)` where the function symbol f is declared `[compos]`. Matches bitstrings M equal to $f(M_1, \ldots, M_n)$ for some $M_1, \ldots, M_n$ that match $p_1, \ldots, p_n$. (The poly-injectivity of f allows us to compute possible values $M_1, \ldots, M_n$ of its arguments from the value of M, and to check whether M is equal to the resulting value of $f(M_1, \ldots, M_n)$.)

- ($p_1, \dots, p_n$) tuples, which are particular [compos] functions encoding unambiguously the values of $p_1, \dots, p_n$ and their type.
- =M' matches a bitstring equal to M'.

The else clause is never executed when the pattern is simply a variable. When else P' is omitted, it is equivalent to else yield. Similarly, when in P is omitted, it is equivalent to in yield.

- if *cond* then P else P' is syntactic sugar for find suchthat *cond* then P else P'. It executes P if the condition *cond* evaluates to true and P' if *cond* evaluates to false. When the else clause is omitted, it is implicitly else yield. (else 0 would not be syntactically correct.)

- Next, we explain the process find FB_1 orfind ... orfind FB_m else P where each branch FB_j is $FB_j = u_{j1}$<=$n_{j1}, \dots, u_{jm_j}$<=n_{jm_j} suchthat $cond_j$ then P_j.

  A simple example is the following: find u<=n suchthat defined($x[u]$) && $x[u] = a$ then P' else P tries to find an index u such that $x[u]$ is defined and $x[u] = a$, and when such a u is found, it executes P' with that value of u; otherwise, it executes P. In other words, this find construct looks for the value a in the array x, and when a is found, it stores in u an index such that $x[u] = a$. Therefore, the find construct allows us to access arrays, which is key for our purpose.

  More generally, find u_1<=$n_1, \dots, u_m$<=n_m suchthat defined($M_1, \dots, M_l$) && M then P' else P tries to find values of $u_1, \dots, u_m$ for which $M_1, \dots, M_l$ are defined and M is true. In case of success, it executes P'. In case of failure, it executes P.

  This is further generalized to m branches: find FB_1 orfind ... orfind FB_m else P where $FB_j = u_{j1}$<=$n_{j1}, \dots, u_{jm_j}$<=n_{jm_j} suchthat defined($M_{j1}, \dots, M_{jl_j}$) && M_j then P_j tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \dots, u_{jm_j}$ for which $M_{j1}, \dots, M_{jl_j}$ are defined and M_j is true. In case of success, it executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions $cond_j$ = defined($M_{j1}, \dots, M_{jl_j}$) && M_j for each j and each value of $u_{j1}, \dots, u_{jm_j}$ in $[1, n_{j1}] \times \dots \times [1, n_{jm_j}]$. If none of these conditions is true, it executes P. Otherwise, it chooses randomly with almost uniform probability[1] one j and one value of $u_{j1}, \dots, u_{jm_j}$ such that the corresponding condition is true, and executes P_j.

  In the general case, the conditions $cond_j$ are of the form defined($M_1, \dots, M_l$) [&& M] or simply M. The condition defined($M_1, \dots, M_l$) means that $M_1, \dots, M_l$ are defined. At least one of the two conditions defined or M must be present. Omitted defined conditions are considered empty; when M is omitted, it is considered true.

  Internally, CryptoVerif distinguishes two variables for each index of find, so that the syntax of a find branch becomes $FB_j = u_{j1} = u'_{j1}$<=$n_{j1}, \dots, u_{jm_j} = u'_{jm_j}$<=$n_{jm_j}$ suchthat defined($M_{j1}, \dots, M_{jl_j}$) && M_j then P_j. The variables $u'_{j1}, \dots, u'_{jm_j}$ are considered as replication indices, and are used in the defined condition and in M_j: they are temporary variables that are used as loop indices to look for indices that satisfy the desired conditions. Once suitable indices are found, their value is stored in $u_{j1}, \dots, u_{jm_j}$ and the then branch is executed using these variables. It is possible to make array accesses to $u_{j1}, \dots, u_{jm_j}$ (such as $u_{j1}[M_1, \dots, M_k]$) elsewhere in the game, which is not possible for $u'_{j1}, \dots, u'_{jm_j}$.

  A variant of find is find[unique]. Consider the process find[unique] FB_1 orfind ... orfind FB_m else P where $FB_j = u_{j1}$<=$n_{j1}, \dots, u_{jm_j}$<=n_{jm_j} suchthat defined($M_{j1}, \dots, M_{jl_j}$) && M_j then P_j. When there are several values of $j, u_{j1}, \dots, u_{jm_j}$ for which $M_{j1}, \dots, M_{jl_j}$ are defined and M_j is true, this process executes an event NonUnique and aborts the game. In all other cases, it behaves as find. Intuitively, find[unique] should be used when there is a negligible probability of finding several suitable values of $j, u_{j1}, \dots, u_{jm_j}$. The construct find[unique] is typically not used in the initial game. (One would have to prove manually that there is indeed a negligible probability of finding several suitable values of $j, u_{j1}, \dots, u_{jm_j}$. CryptoVerif displays a warning if find[unique] occurs in the initial game.) However, find[unique] is used in the specification of cryptographic primitives, in the right-hand of equivalences specified by equiv.

[1] Precisely, the distance between the distribution actually used for choosing $j, u_{j1}, \dots, u_{jm_j}$ and the uniform distribution is at most eps_find. See the explanation of new x:T for details on how to achieve this.

- `insert` tbl`(`$M_1, \dots, M_n$`);`P inserts the tuples $(M_1, \dots, M_n)$ in the table tbl, then executes P. The table tbl must have been declared with the appropriate types using the `table` declaration.

- `get` tbl`(`$p_1, \dots, p_n$`) suchthat` M `in` P `else` P' tries to find an element of the table tbl that matches the patterns $p_1, \dots, p_n$ and such that M is true. If it succeeds, it executes P with the variables of $p_1, \dots, p_n$ bound to that element of the table. If several elements match, one of them is chosen randomly with (almost) uniform probability. If no element matches, it executes P'.

  When `else` P' is omitted, it is equivalent to `else yield`. When `suchthat` M is omitted, it is equivalent to `suchthat` *true*. Internally, `get` is converted into `find` by CryptoVerif.

- Finally, let us explain the output `out(`c`[`$M_1, \dots, M_l$`],`N`);`Q. A channel c`[`$M_1, \dots, M_l$`]` consists of both a channel name c (declared by `channel` c) and a tuple of terms $M_1, \dots, M_l$. Terms $M_1, \dots, M_l$ are intuitively analogous to IP addresses and ports which are numbers that the adversary may guess. Two channels are equal when they have the same channel name and terms that evaluate to the same bitstrings. A semantic configuration always consists of a single output process (the process currently being executed) and several input processes. When the output process executes `out(`c`[`$M_1, \dots, M_l$`],`N`);`Q, one looks for an input on the same channel in the available input processes. If no such input process is found, the process blocks. Otherwise, one such input process `in(`c`[`$M'_1, \dots, M'_l$`],`p`);`P is chosen randomly with (almost) uniform probability. The communication is then executed: the output message N is evaluated, its result is truncated to the maximum length of bitstrings on channel c, the obtained bitstring is matched against pattern p. Finally, the output process P that follows the input is executed. The input process Q that follows the output is stored in the available input processes for future execution.

  Patterns p are as in the `let` process, except that variables in p that are not under a function symbol $f(\dots)$ must be declared with their type.

  In the game as given to CryptoVerif, the channel is just a channel name c, and it is an abbreviation for $c[i_1, \dots, i_n]$ where $i_1, \dots, i_n$ are the current replication indexes at the considered input or output. It is recommended to use as channel a different channel name for each input and output. Then the adversary has full control over the network: it can decide precisely from which copy of which input it receives a message and to which copy of which output it sends a message, by using the appropriate channel name and value of the replication indexes.

  Note that the syntax requires an output to be followed by an input process, as in [5]. If one needs to output several messages consecutively, one can simply insert fictitious inputs between the outputs. The adversary can then schedule the outputs by sending messages to these inputs.

In this calculus, all variables are implicitly arrays. When a variable x is defined (by `new`, `let`, `find`, `in`) under replications `!`i_1`<=`N_1, ..., `!`i_n`<=`N_n, x has implicitly indexes $i_1, \dots, i_n$: x stands for $x[i_1, \dots, i_n]$. Arrays allow us to have full access to the state of the process. Arrays can be read using `find`. Similarly, when x is used with $k < n$ indexes the missing $n - k$ indexes are implicit: $x[u_1, \dots, u_k]$ stands for $x[i_1, \dots, i_{n-k}, u_1, \dots, u_k]$ where $i_1, \dots, i_{n-k}$ must be the $n - k$ first replication indexes both at the creation of x and at the usage $x[u_1, \dots, u_k]$. (So the usage and creation of x must be under the same $n - k$ top-most replications.)

In the initial game, several variables may be defined with the same name, but they are immediately renamed to different names, so that after renaming, each variable is defined once. When several variables are defined with the same name, they can be referenced only under their definition without explicit array indexes, because for other references, we would not know which variable to reference after renaming.

In subsequent games created by CryptoVerif, a variable may be defined at several occurrences, but these occurrences must be in different branches of `if`, `find`, or `let`, so that they cannot be executed with the same value of the array indexes. This constraint guarantees that each array cell is defined at most once.

Each usage of x must be either:

- x without array index syntactically under its definition. (Then x is implicitly considered to have as indexes the current replication indexes at its definition.)

- x possibly with array indexes inside the `defined` condition of a find.

- $x[M_1, \dots, M_n]$ in M or P in a find branch … `suchthat defined(`$M'_1, \dots, M'_l$`) &&` M `then` P, such that $x[M_1, \dots, M_n]$ is a subterm of $M'_1, \dots, M'_l$.
- $x[M_1, \dots, M_n]$ in M or M'' in a find branch … `suchthat defined(`$M'_1, \dots, M'_l$`) &&` M `then` M'', such that $x[M_1, \dots, M_n]$ is a subterm of $M'_1, \dots, M'_l$.

These syntactic constraints guarantee that a variable is accessed only when it is defined. Moreover, the variables defined in conditions of `find` or in patterns or conditions of `get` must not have array accesses (that is, accesses corresponding to the last three cases above).

Finally, the calculus is equipped with a type system. To be able to use variables outside their scope (by `find`), the type checking algorithm works in two passes.

In the first pass, it collects the type of each variable: when a variable x is defined with type T under replications `!`N_1, …, `!`N_n, x has type $[1, N_1] \times \ldots \times [1, N_n] \rightarrow T$. When the type of x is not explicitly given in its declaration (in patterns in `let` or `in`), its type is left undefined in this pass, and x cannot be used outside its scope.

In the second pass, the type system checks the following requirements: In x`[`$M_1, \dots, M_m$`]`, $M_1, \dots, M_m$ must be of the suitable interval type, that is, a suffix of the types of replication indexes at the definition of x. In f`(`$M_1, \dots, M_m$`)`, if f has been declared by `fun` f`(`$T_1, \dots, T_m$`):`T, M_j must be of type T_j, and $f(M_1, \dots, M_m)$ is then of type T. In `(`$M_1, \dots, M_n$`)`, M_j can be of any bitstring type (that is, not an index type $[1, N]$), and the result is of type `bitstring`. In M_1 `=` M_2 and M_1 `<>` M_2, M_1 and M_2 must be of the same type, and the result is of type `bool`. In M_1 `||` M_2 and M_1 `&&` M_2, M_1 and M_2 must be of type `bool` and the result is of type `bool`. The type system requires each subterm to be well-typed. Furthermore, in `event` e`(`$M_1, \dots, M_n$`)`, if e has been declared by `event` e`(`$T_1, \dots, T_n$`)`, M_j must be of type T_j. In `new` x`:`T, T must be declared with option `bounded` (or `fixed`). In `if` M `then ... else ...`, M must be of type `bool`. Similarly, for

`find ... orfind ... suchthat defined(...) &&` M `then ...`

M must be of type `bool`. In `let` p `=` M `in ...`, M and p must be of the same type. For function application and tuple patterns, the typing rule is the same as for the corresponding terms. The pattern $x : T$ is of type T; the pattern x can be of any bitstring type, determined by the usage of x (when the pattern x is used as argument of a tuple pattern, its type is `bitstring`); the pattern `=`M is of the type of M. In `out(`c`[`$M_1, \dots, M_n$`],`M`)`, M must be of a bitstring type.

A declaration can be:

- `set` ⟨parameter⟩ `=` ⟨value⟩.

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

  - `set diffConstants = true.`
    `set diffConstants = false.`

    When `true`, different constant symbols are assumed to have a different value. When `false`, CryptoVerif does not make this assumption.
  - `set constantsNotTuple = true.`
    `set constantsNotTuple = false.`

    When `true`, constant symbols are assumed to be different from the result of applying a tuple function to any argument. When `false`, CryptoVerif does not make this assumption.
  - `set expandAssignXY = true.`
    `set expandAssignXY = false.`

    When `true`, CryptoVerif automatically removes assignments `let x = y` where `x` and `y` are variables by substituting `y` for `x` (in the transformation `remove_assign useless`) When `false`, this transformation is not performed as part of `remove_assign useless`.
  - `set minimalSimplifications = true.`
    `set minimalSimplifications = false.`

    When `true`, simplification replaces a term with a rewritten term only when the rewriting has used at least one rewriting rule given by the user, not when only equalities that come from let definitions and other instructions in the game have been used. When false, a term is replaced with its rewritten form in all cases. The latter configuration often leads to replacing a term with a more complex one, in particular expanding let definitions, thus duplicating their contents.

- set mergeBranches = true.
  set mergeBranches = false.

  When true, the transformation merge_branches is applied after simplification, to merge branches of if, let, and find when all branches execute the same code. This is useful in order to remove the test, which can remove a use of a secret. When false, this transformation is not performed. This is useful in particular when the test has been manually introduced in order to force CryptoVerif to distinguish cases.

- set mergeArrays = true.
  set mergeArrays = false.

  When true, merge_branches advises merge_arrays commands to make the merging of branches of if, find, let succeed more often. When false, this advice is not automatically given and the user should use the manual command merge_arrays (defined in Section 7) to perform the merging.

- set uniqueBranch = true.
  set uniqueBranch = false.

  When uniqueBranch = true, the following transformation is enabled as part of simplify: if a branch of a find[unique] is proved to succeed, then simplification removes all other branches of that find. When uniqueBranch = false, this transformation is not performed.

- set uniqueBranchReorganize = true.
  set uniqueBranchReorganize = false.

  When uniqueBranchReorganize = true, the following transformations are enabled as part of simplify:

  - If a find[unique] occurs in the then branch of a find[unique], we reorganize them.
  - If a find[unique] occurs in the condition of a find, we reorganize them.

  When uniqueBranchReorganize = false, these transformations are not performed.

- set autoSARename = true.
  set autoSARename = false.

  When true, and a variable is defined several times and used only in the scope of its definition with the current replication indexes at that definition, each definition of this variable is renamed to a different name, and the uses are renamed accordingly, by the transformation remove_assign. When false, such a renaming is not done automatically, but in manual proofs, it can be requested specifically for each variable by SArename x, where x is the name of the variable.

- set autoMove = true.
  set autoMove = false.

  When true, the transformation move all is automatically executed after each cryptographic transformation. This transformation moves random number generations new downwards as much as possible, duplicating them when crossing a if, let, or find. (A future SArename transformation may then enable us to distinguish cases depending on which of the duplicated random number generations a variable comes from.) It also moves assignments down in the syntax tree but without duplicating them, when the assignment can be moved under a if, let, or find, in which the assigned variable is used only in one branch. (In this case, the assigned term is computed in fewer cases thanks to this transformation.)

  When false, the transformation move all is never automatically executed.

- set optimizeVars = false.
  set optimizeVars = true.

When `true`, CryptoVerif tries to reduce the number of different intermediate variables introduced in cryptographic transformations. This can lead to distinguishing fewer cases, which unfortunately often leads to a failure of the proof. When `false`, different intermediate varaibles are used for each occurrence of the transformed expression.

- `set interactiveMode = false.`
  `set interactiveMode = true.`

  When `false`, CryptoVerif runs automatically. When `true`, CryptoVerif waits for instructions of the user on how to perform the proof. (See Section 7 for details on these instructions.) This setting is ignored when proof instructions are included in the input file using the `proof` command. In this case, the instructions given in the `proof` command are executed, without user interaction.

- `set autoAdvice = true.`
  `set autoAdvice = false.`

  In interactive mode, when `autoAdvice = true`, execute the advised transformations automatically. When `autoAdvice = false`, display the advised transformations, but do not execute them. The user may then give them as instructions if he wishes.

- `set noAdviceCrypto = false.`
  `set noAdviceCrypto = true.`

  When `noAdviceCrypto = true`, prevents the cryptographic transformations from generating advice. Useful mainly for debugging the proof strategy.

- `set noAdviceGlobalDepAnal = false.`
  `set noAdviceGlobalDepAnal = true.`

  When `noAdviceGlobalDepAnal = true`, prevents the global dependency analysis from generating advice. Useful when the global dependency analysis generates bad advice.

- `set simplifyAfterSARename = true.`
  `set simplifyAfterSARename = false.`

  When `simplifyAfterSARename = true`, apply simplification after each execution of the SArename transformation. This slows down the system, but enables it to succeed more often.

- `set backtrackOnCrypto = false.`
  `set backtrackOnCrypto = true.`

  When `backtrackOnCrypto = true`, use backtracking when the proof fails, to try other cryptographic transformations. This slows down the system considerably (so it is false by default), but enables it to succeed more often, in particular for public-key protocols that mix several primitives. One usage is to try first with the default setting and, if the proof fails although the property is believed to hold, try again with backtracking.

- `set useKnownEqualitiesInCryptoTransform = true.`
  `set useKnownEqualitiesInCryptoTransform = false.`

  When `useKnownEqualitiesInCryptoTransform = true`, CryptoVerif relies on known equalities between terms to replace variables with their values in the cryptographic transformations. When it is false, CryptoVerif just uses the variables as their appear in the game, and relies only on advice to replace variables with their values.

- `set ignoreSmallTimes =` ⟨n⟩. (default 3)

  When `0`, the evaluation of the runtime is very precise, but the formulas are often too complicated to read.

  When `1`, the system ignores many small values when computing the runtime of the games. It considers only function applications and pattern matching.

  When `2`, the system ignores even more details, including application of boolean operations (`&&`, `||`, `not`), constants generated by the system, `()` and matching on `()`. It ignores the creation and decomposition of tuples in inputs and outputs.

  When `3`, the system additionally ignores the time of equality tests between values of bounded length, as well as the time of all constants.

- `set maxIterSimplif = ⟨n⟩.` (default 2)

  Sets the maximum number of repetitions of the simplification transformation for each `simplify` instruction. A greater value slows down the system but may enable it to obtain simpler games, and therefore increase its chances of success. When $n \leq 0$, repeats simplification until a fixpoint is reached.

- `set maxAddFactDepth = ⟨n⟩.` (default 1000)

  Sets the maximum depth of recursion in the addition and simplification of known facts. When $n \leq 0$, puts no limit on this depth of recursion. Putting a limit avoids an infinite loop in some rare cases.

- `set maxIterRemoveUselessAssign = ⟨n⟩.` (default 10)

  Sets the maximum number of repetitions of the removal of useless assignments for each `remove_assign useless` instruction. A greater value slows down the system but may enable it to obtain simpler games, and therefore increase its chances of success. When $n \leq 0$, repeats removal of useless assignments until a fixpoint is reached.

- `set minAutoCollElim = ⟨s⟩.` (default `size15`)

  Sets the minimum size of a type for which elimination of collisions is possible automatically. The size argument ⟨s⟩ can be `large`, `password`, or `size`n (see the `type` declaration for their meaning).

- `set maxAdvicePossibilitiesBeginning =` n_1. (default `50`)
  `set maxAdvicePossibilitiesEnd =` n_2. (default `10`)

  In cryptographic transformations, when CryptoVerif can transform many terms in several ways of different priority, these various ways combine, yielding a very large number of advice possibilities. These two options allow to limit the number of considered advice possibilities by keeping the n_1 first possibilities (with highest priority) and the n_2 last possibilities (with lowest priority but fewer advised transformations). When n_1 or n_2 are not positive, all advice possibilities are kept, but that may yield a very slow execution.

- `set elsefindFactsInReplace = true.`
  `set elsefindFactsInReplace = false.`

  When `elsefindFactsInReplace = true`, CryptoVerif will try to infer more facts when doing a `replace` operation: when it encounters a `find` branch in the process, it considers a variable $x[M_1, \dots, M_l]$, which is guaranteed to be defined by this `find`. If x is defined in the `else` part of another `find` construct, then at the definition of x, we know that the conditions of the `then` branches of this `find` are not satisfied:

  $$\forall u_1, \dots, u_k, \mathtt{not}(\mathtt{defined}(y_1[M_{11}, \dots, M_{1l_1}], \dots, y_k[M_{k1}, \dots, M_{kl_k}]) \wedge t)$$

  We try to infer $\mathtt{not}(t)$ from this fact.

  ∗ if each variable $y_j[M_{j1}, \dots, M_{jl_j}]$ is defined before $x[M_1, \dots, M_l]$, then $\mathtt{not}(t)$ indeed holds by the fact above;

  ∗ for each $y_j[M_{j1}, \dots, M_{jl_j}]$, we assume that $y_j[M_{j1}, \dots, M_{jl_j}]$ is defined after $x[M_1, \dots, M_l]$ and try to prove $\mathtt{not}(t)$.
  It this proof succeeds, we can infer that $\mathtt{not}(t)$ holds at the current program point.

- `set elsefindFactsInSimplify = true.`
  `set elsefindFactsInSimplify = false.`

  Similar to `elsefindFactsInReplace`, but applies in `simplify` operations.

- `set maxReplaceDepth =` n`.` (default 20)

  Sets the maximum number of rewriting steps that are allowed to prove that the new term is equal to the old one in a `replace` transformation.

The default value is the first mentioned, except when explicitly specified. In most cases, the default values should be left as they are, except for `interactiveMode`, which allows to perform interactive proofs.

- `param` seq$^+$⟨ident⟩ `[[noninteractive] | [passive] | [size`n`]]`.

  `param` $n_1, \dots, n_m$`.` declares parameters $n_1, \dots, n_m$. Parameters are used to represent the number of copies of replicated processes (that is, the maximum number of calls to each query). In asymptotic analyses, they are polynomial in the security parameter. In exact security analyses, they appear in the formulas that express the probability of an attack.

  The options `[noninteractive]`, `[passive]`, or `[size`n`]` indicate to CryptoVerif an order of magnitude of the size of the parameter. The option `[size`n`]` (where n is a constant integer) indicates that the considered parameter has "size n": the larger the n, the larger the parameter is likely to be. CryptoVerif uses this information to optimize the computed probability bounds: when several bounds are correct, it chooses the smallest one.

  The option `[noninteractive]` means that the queries bounded by the considered parameters can be made by the adversary without interacting with the tested protocol, so the number of such queries is likely to be large. Parameters with option `[noninteractive]` are typically used for bounding the number of calls to random oracles. `[noninteractive]` is equivalent to `[size20]`.

  The option `[passive]` means that the queries bounded by the considered parameters correspond to the adversary passively listening to sessions of the protocol that run as expected. Therefore, for such runs, the adversary is undetected. This number of runs is therefore likely to be larger than runs in which the adversary actively interacts with the honest participants, when these participants stop after a certain number of failed attempts. `[passive]` is equivalent to `[size10]`.

- `proba` ⟨ident⟩`.`

  `proba` p`.` declares a probability p. (Probabilities may be used as functions of other arguments, without explicit checking of these arguments.)

- `type` ⟨ident⟩ `[[`seq$^+$⟨option⟩`]]`.

  `type` T`.` declares a type T. Types correspond to sets of bitstrings or a special symbol $\perp$ (used for failed decryptions, for instance). Optionally, the declaration of a type may be followed by options between brackets. These options can be:

  - `bounded` means that the type is a set of bitstrings of bounded length or perhaps $\perp$. In other words, the type is a finite subset of bitstrings plus $\perp$.
  - `fixed` means that the type is the set of all bitstrings of a certain length n. In particular, the type is a finite set, so `fixed` implies `bounded`.
  - `nonuniform` means that random numbers may be chosen in the type with a non-uniform distribution. (When `nonuniform` is absent, random numbers are chosen using a uniform distribution for `fixed` types, an almost uniform distribution for `bounded` types, and random values cannot be chosen among other types.)
  - `large`, `password`, and `size`n indicate the order of magnitude of the probability of collision `Pcoll1rand`(T) between a random element chosen according to the default probability distribution D_T for the considered type T, and an independent element of type T. When the default distribution is uniform or almost uniform (`fixed` and `bounded` types), `Pcoll1rand`$(T) = \frac{1}{|T|}$, so these parameters give an order of magnitude of the cardinal of the type.

    The option `size`n (where n is a constant integer) indicates that the considered type has "size n": the larger the n, the smaller the probability of collision `Pcoll1rand`(T). When no size option is present, the type has size 0. CryptoVerif uses this information to determine whether collisions with random elements of the considered type T should be eliminated. For collisions to be eliminated, two conditions must be satisfied:

    1. the size of the type must be at least `minAutoCollElim` (which is set by `set minAutoCollElim =` n; the default is 15), or the size of the type must be at least 1 and elimination of collisions on this data has been manually requested by the command `simplify coll_elim` ....
    2. the probability of collision is at most one of the formulas specified by the command `allowed_collisions` (used inside a `proof` environment). By default, all collisions are eliminated for types of size at least 20, and collisions are eliminated for types of size at

least 10 when the collision is repeated at most N times, where N is a parameter of size 0. See the command `allowed_collisions` for more details.

`large` means that the type T is large enough so that all collisions with random elements of T can be eliminated. (In asymptotic analyses, `Pcoll1rand`(T) is negligible. In exact security analyses, the probability of a collision is correctly expressed by the system.) `large` is equivalent to `size20`.

`password` is intended for passwords in password-based security protocols. These passwords are taken in a dictionary whose size is much smaller than the size of a nonce for instance, so the probability of collisions among passwords is larger than among data of `large` types. `password` is equivalent to `size10`.

- `fun` ⟨ident⟩`(`seq⟨ident⟩`):`⟨ident⟩ `[`[seq⁺⟨option⟩]`]``.`

  `fun` f`(`$T_1, \dots, T_n$`):`T`.` declares a function that takes n arguments, of types $T_1, \dots, T_n$, and returns a result of type T. Optionally, the declaration of a function may be followed by options between brackets. These options can be:

  – `compos` means that f is injective and that its inverses can be computed in polynomial time: $f(x_1, \dots, x_m) = y$ implies for $i \in \{1, \dots, m\}$, $x_i = f_i^{-1}(y)$ for some functions f_i^{-1}. (In the vocabulary of [1], f is poly-injective.) f can then be used for pattern matching.

  – `decompos` means that f is an inverse of a poly-injective function. f must be unary. (Thanks to the pattern matching construct, one can in general avoid completely the declaration of `decompos` functions, by just declaring the corresponding poly-injective function `compos`.)

  – `uniform` means that f maps the default distribution of its argument into the default distribution of its result. f must be unary; the argument and the result of f must be of types marked `fixed`, `bounded`, or `nonuniform`.

- `letfun` ⟨ident⟩[`(`seq⟨vartype⟩`)`]`=`⟨term⟩`.`

  `letfun` f`(`x_1`:`$T_1, \dots, x_n$`:`T_n`)=`M`.` declares a function f that takes n arguments named $x_1, \dots, x_n$ of types $T_1, \dots, T_n$, respectively. The subsequent calls to this function are replaced by the term M in which we replace $x_1, \dots, x_n$ with the arguments given by the caller.

  The term M must not contain `find` constructs.

- `const` seq⁺⟨ident⟩`:`⟨ident⟩`.`

  `const` $c_1, \dots, c_n$`:`T`.` declares constants $c_1, \dots, c_n$ of type T. Different constants are assumed to correspond to different bitstrings (except when the instruction `set diffConstants = false.` is given).

- `table` ⟨ident⟩`(`seq⁺⟨ident⟩`).`

  `table` tbl`(`$T_1, \dots, T_n$`).` declares the table tbl, whose elements are tuples of type $T_1, \dots, T_n$. Elements can be inserted in the table by `insert` tbl`(`$M_1, \dots, M_n$`)` and the table can be read using `get`.

- `channel` seq⁺⟨ident⟩`.`

  `channel` $c_1, \dots, c_n$`.` declares communication channels $c_1, \dots, c_n$.

- `event` ⟨ident⟩[`(`seq⟨ident⟩`)`]`.`

  `event` e`(`$T_1, \dots, T_n$`).` declares an event e that takes arguments of types $T_1, \dots, T_n$. When there are no arguments, we can simply declare `event` e`.`

- `let` ⟨ident⟩ `=` ⟨oprocess⟩`.`
  `let` ⟨ident⟩ `=` ⟨iprocess⟩`.`

  `let` x `=` P`.` says that x represents the process P. When parsing a process, x will be replaced with P.

- `forall` seq⟨vartype⟩`;`⟨simpleterm⟩`.`

  `forall` $x_1 : T_1, \dots, x_n : T_n$`;`$M$`.` says that for all values of $x_1, \dots, x_n$ in types $T_1, \dots, T_n$ respectively, M is true. The term M must be a simple term without array accesses. When M is an equality M_1`=`M_2, CryptoVerif uses this information for rewriting M_1 into M_2, so one must be careful of the orientation of the equality, in particular for termination. When M is an inequality, M_1`<>`M_2, CryptoVerif rewrites M_1`=`M_2 to false and M_1`<>`M_2 to true. Otherwise, it rewrites M to true.

- `equation` ⟨eq_name⟩`(`seq$^+$⟨ident⟩`).`

  This declaration declares the equational theories satisfied by function symbols. The following equational theories are supported:

  - `equation commut(`f`).` indicates that the function f is commutative, that is, $f(x, y) = f(y, x)$ for all x, y. In this case, the function f must be a binary function with both arguments of the same type. (The equation $f(x, y) = f(y, x)$ cannot be given by the `forall` declaration because CryptoVerif interprets such declarations as rewrite rules, and the rewrite rule $f(x, y) \to f(y, x)$ does not terminate.)
  - `equation assoc(`f`).` indicates that the function f is associative, that is, $f(x, f(y, z)) = f(f(x, y), z)$ for all x, y, z. In this case, the function f must be a binary function with both arguments and the result of the same type.
  - `equation AC(`f`).` indicates that the function f is associative and commutative. In this case, the function f must be a binary function with both arguments and the result of the same type.
  - `equation assocU(`f`, `n`).` indicates that the function f is associative, and that n is a neutral element for f, that $f(x, n) = f(n, x) = x$ for all x. In this case, the function f must be a binary function with both arguments and the result of the same type as the type of the constant n.
  - `equation ACU(`f`, `n`).` indicates that the function f is associative and commutative, and that n is a neutral element for f. In this case, the function f must be a binary function with both arguments and the result of the same type as the type of the constant n.
  - `equation ACUN(`f`, `n`).` indicates that the function f is associative and commutative, that n is a neutral element for f, and that f satisfies the cancellation equation $f(x, x) = n$. In this case, the function f must be a binary function with both arguments and the result of the same type as the type of the constant n.
  - `equation group(`f`, `inv`, `n`).` indicates that f forms group with inverse inv and neutral element n, that is, the function f is associative, n is a neutral element for f, and $inv(x)$ is the inverse of x, that is, $f(inv(x), x) = f(x, inv(x)) = n$. In this case, the function f must be a binary function with both arguments and the result of the same type T, inv must be a unary function that takes and returns a value of type T, and n must be a constant of type T.
  - `equation commut_group(`f`, `inv`, `n`).` indicates that f forma commutative group with inverse inv and neutral element n, that is, the function f is associative and commutative, n is a neutral element for f, and $inv(x)$ is the inverse of x. In this case, the function f must be a binary function with both arguments and the result of the same type T, inv must be a unary function that takes and returns a value of type T, and n must be a constant of type T.

- `collision (new` ⟨vartype⟩`;)`*`[forall` seq⟨vartype⟩`;]`⟨simpleterm⟩ `<=(`⟨proba⟩`)=>` ⟨simpleterm⟩`.`

  `collision new` x_1`:`T_1`;`…`new` x_n`:`T_n`;forall` $y_1 : T'_1, \dots, y_m : T'_m$`;`$M_1$ `<=(`p`)=>` M_2`.` means that when $x_1, \dots, x_n$ are chosen randomly and independently in $T_1, \dots, T_n$ respectively (with the default probability distributions for these types), a Turing machine running in time `time` has probability at most p of finding $y_1, \dots, y_m$ in $T'_1, \dots, T'_m$ such that $M_1 \neq M_2$. The terms M_1 and M_2 must be simple terms without array accesses. See below for the syntax of probability formulas.

  This allows CryptoVerif to rewrite M_1 into M_2 with probability loss p, when $x_1, \dots, x_n$ are created by independent random number generations of types $T_1, \dots, T_n$ respectively. One should be careful of the orientation of the equivalence, in particular for termination.

- `equiv` ⟨name⟩ seq$^+$⟨funmode⟩ `<=(`⟨proba⟩`)=>` [`[manual]`|`[computational]`] seq$^+$⟨fungroup⟩`.`

  `equiv` *name* L `<=(`p`)=>` R`.` means that the probability that a probabilistic Turing machine that runs in time `time` distinguishes L from R is at most p. The name *name* is used to designate the equivalence in the `crypto` command used in manual proofs (see Section 7). This name can be either an identifier *id*, or *id*(f), where *id* is an identifier and f a second identifier. Names of the form *id*(f) are most useful when the equivalence is defined inside a macro definition (`define`). In this case, the identifier *id* is kept unchanged and the identifier f is renamed during macro expansion; if f is a parameter of the macro, it is then replaced with its value at macro expansion, so that one can always designate precisely the desired equivalence even when a macro is expanded several times.

  L and R define sets of functions. (They can be translated into processes as explained in [1].)

  - O`(`x_1 `:` $T_1, \dots, x_n$ `:` T_n`)` `:=` M represents a function O that takes arguments $x_1, \dots, x_n$ of types $T_1, \dots, T_n$ respectively, and returns the result M.
  - Optionally, in the left-hand side, an integer between brackets `[`n`]` ($n \geq 0$) can be added in the previous functions, which become `O(`x_1 `:` $T_1, \dots, x_n$ `:` T_n`)` `[`n`]` `:=` M This integer does not change the semantics of the function, but is used for the proof strategy: CryptoVerif uses preferably the functions with the smallest integers n when several functions can be used for representing the same expression. When no integer is mentioned, $n = 0$ is assumed, so the function has the highest priority.
  - Optionally, in the left-hand side, the indication `[useful_change]` can also be added in the previous functions, which become `O(`x_1 `:` $T_1, \dots, x_n$ `:` T_n`)` `[useful_change]` `:=` M This indication is also used for the proof strategy: if at least one `[useful_change]` indication is present, CryptoVerif applies the transformation defined by the equivalence only when at least one `[useful_change]` function is called in the game.
  - `!`i`<=`N `new` y_1`:`T'_1`;`...`new` y_m`:`T'_m`;(`$FG_1, \dots, FG_n$`)` represents N copies of a process that picks fresh random numbers y_1, ..., y_m of types $T'_1, \dots, T'_m$ respectively, and makes available the functions described in $FG_1, \dots, FG_n$. Each copy has a different value of $i \in [1, N]$. The identifier i cannot be referred to explicitly in the process; it is used only implicitly as array index of variables defined under `!`i`<=`N. The replication `!`i`<=`N can be abbreviated `!`N.

  CryptoVerif uses such equivalences to transform processes that call functions of L into processes that call functions of R.

  L may contain mode indications to guide the rewriting: the mode `[all]` means that all occurrences of the root function symbol of functions in the considered group must be transformed; the mode `[exist]` means that at least one occurrence of a function in this group must be transformed. (`[exist]` is the default; when a function group contains no random number generation, it must be in mode `[all]`.)

  The `[manual]` indication, when it is present in the equivalence, prevents the automatic application of the transformation. The transformation is then applied only using the manual `crypto` command.

  The `[computational]` indication, when it is present in the equivalence, means that the transformation relies on a computational assumption (by opposition to decisional assumptions). This indication allows one to mark some random number generations of the right-hand side of the equivalence with `[unchanged]`, which means that the random value is preserved by the transformation. The transformation is then allowed even if the random value occurs as argument of events. (This argument will be unchanged.) The mark `[unchanged]` is forbidden when the equivalence is not marked `[computational]`. Indeed, decisional assumptions may alter any random values.

  L and R must satisfy certain syntactic constraints:

  - L and R must be well-typed, satisfy the constraints on array accesses (see the description of processes above), and the type of the results of corresponding functions in L and R must be the same.
  - L cannot contain `find`, `let`, `if`.

  - L and R must have the same structure: same replications, same number of functions, same function names in the same order, same number of arguments with the same types for each function.
  - Under a replication with no random number generation in L, one can have only a single function.
  - Replications in L (resp. R) must have pairwise distinct bounds. Functions in L (resp. R) must have pairwise distinct names.
  - Finds in R are of the form

    `find[[unique]]` ...
    `orfind` u_1 `<=` $N_1, \dots, u_m$ `<=` N_m `suchthat defined(`$z_1[\widetilde{u_1}], \dots, z_l[\widetilde{u_l}]$`) &&` M `then` FP
    ... `else` FP'

    where $\widetilde{u_k}$ is a non-empty prefix of $u_1, \dots, u_m$, at least one $\widetilde{u_k}$ for $1 \leq k \leq l$ is the whole sequence $u_1, \dots, u_m$, and the implicit prefix of the current array indexes is the same for all $z_1, \dots, z_l$. (When z is defined under replications `!`N_1, ..., `!`N_n, z is always an array with n dimensions, so it expects n indexes, but the first $n' < n$ indexes are left implicit when they are equal to the current indexes of the top-most n' replications above the usage of z—which must also be the top-most n' replications above the definition of z. We require the implicit indexes to be the same for all variables $z_1, \dots, z_l$.) Furthermore, there must exist $k \in \{1, \dots, l_j\}$ such that for all $k' \neq k$, $z_{k'}$ is defined syntactically above all definitions of z_k and $\widetilde{u_{k'}}$ is a prefix of $\widetilde{u_k}$. Finally, variables z_k must not be defined by a `find` in R.

  This is the key declaration for defining the security properties of cryptographic primitives. Since such declarations are delicate to design, we recommend using predefined primitives listed in Section 6, or copy-pasting declarations from examples.

- `query` seq$^+\langle$query$\rangle$`.`

  The `query` declaration indicates which security properties we would like to prove. The available queries are as follows:

  - `secret1` x: show that any element of the array x cannot be distinguished from a random number (by a single test query). In the vocabulary of [1], this is one-session secrecy.
  - `secret` x: show that the array x is indistinguishable from an array of independent random numbers (by several test queries). In the vocabulary of [1], this is secrecy.
  - $x_1 : T_1, \dots, x_n : T_n$`;event` M `==>` M'. First, we declare the types of all variables $x_1, \dots, x_n$ that occur in M or M'. The system shows that, for all values of variables that occur in M, if M is true then there exist values of variables of M' that do not occur in M such that M' is true.

    M must be a conjunction of terms [`inj:`]e or [`inj:`]$e(M_1, \dots, M_n)$ where e is an event declared by `event` and the M_i are simple terms without array accesses (not containing events).

    M' must be formed by conjunctions and disjunctions of terms [`inj:`]e, [`inj:`]$e(M_1, \dots, M_n)$, or simple terms without array accesses (not containing events).

    When `inj:` is present, the system proves an injective correspondence, that is, it shows that several different events marked `inj:` before `==>` imply the execution of several different events marked `inj:` after `==>`. More precisely, `inj:`$e_1(M_{11}, \dots, M_{1m_1})$ `&&` ... `&&` `inj:`$e_n(M_{n1}, \dots, M_{nm_n})$ `&&` ... `==>` M' means that for each tuple of executed events $e_1(M_{11}, \dots, M_{1m_1})$ (executed N_1 times), ..., $e_n(M_{n1}, \dots, M_{nm_n})$ (executed N_n times), M' holds, considering that an event `inj:`$e'(M_1, \dots, M_m)$ in M' holds when it has been executed at least $N_1 \times \dots \times N_n$ times. When e is preceded by `inj:` in a query, e must occur at most once in each branch of `if`, `find`, `let`, and all occurrences of the same e must be under replications of the same types. The `inj:` marker must occur either both before and after `==>` or not at all. (Otherwise, the query would be equivalent to a non-injective correspondence.)

- `proof` $\{\langle\text{command}\rangle$`;`$\ldots$`;`$\langle\text{command}\rangle\}$

  Allows the user to include in the CryptoVerif input file the commands that must be executed by CryptoVerif in order to prove the protocol. The allowed commands are those described in Section 7, except that `help` and `?` are not allowed and that the `crypto` command must be fully specified (so that no user interaction is required). If the command contains a string that is not a valid identifier, `*`, or `.`, then this string must be put between quotes `"`. This is useful in particular for variable names introduced internally by CryptoVerif and that contain `@` (so that they cannot be confused with variables introduced by the user), for example `"@2_r1"`.

- `define` $\langle\text{ident}\rangle$`(`seq$\langle\text{ident}\rangle$`)` $\{$seq$\langle\text{decl}\rangle\}$

  `define` m`(`$x_1, \ldots, x_n$`)` $\{d_1, \ldots, d_k\}$ defines a macro named m, with arguments $x_1, \ldots, x_n$. This macro expands to the declarations $d_1, \ldots, d_k$, which can be any of the declarations listed in this manual, except `define` itself. The macro is expanded by the `expand` declaration described below. When the `expand` declaration appears inside a `define` declaration, the expanded macro must have been defined before the `define` declaration (which prevents recursive macros, whose expansion would loop). Macros are used in particular to define a library of standard cryptographic primitives that can be reused by the user without entering their full definition. These primitives are presented in Section 6.

- `expand` $\langle\text{ident}\rangle$`(`seq$\langle\text{ident}\rangle$`).`

  `expand` m`(`$y_1, \ldots, y_n$`).` expands the macro m by applying it to the arguments $y_1, \ldots, y_n$. If the definition of the macro m is `define` m`(`$x_1, \ldots, x_n$`)` $\{d_1, \ldots, d_k\}$, then it generates $d_1, \ldots, d_k$ in which $y_1, \ldots, y_n$ are substituted for $x_1, \ldots, x_n$ and the other identifiers that were not already defined at the `define` declaration are renamed to fresh identifiers.

The following identifiers are predefined:

- The type `bitstring` is the type of all bitstrings. It is large.
- The type `bitstringbot` is the type that contains all bitstrings and $\perp$. It is also large.
- The type `bool` is the type of boolean values, which consists of two constant bitstrings `true` and `false`. It is declared `fixed`.
- The function `not` is the boolean negation, from `bool` to `bool`.
- The constant `bottom` represents $\perp$. (The special element of `bitstringbot` that is not a bitstring.)

The syntax of probability formulas allows parenthesing and the usual algebraic operations `+`, `-`, `*`, `/`. (`*` and `/` have higher priority than `+` and `-`, as usual.), as well as the maximum, denoted `max(`p_1`,` $\ldots$ `,`p_n`)`. They may also contain

- P or $P(p_1, \ldots, p_n)$ where P has been declared by `proba` P and $p_1, \ldots, p_n$ are probability formulas; this formula represents an unspecified probability depending on $p_1, \ldots, p_n$.
- N, where N has been declared by `param` N, designates the number of copies of a replication.
- $\#O$, where O is a function, designates the number of different calls to the function O.
- $|T|$, where T has been declared by `type` T and is `fixed` or `bounded`, designates the cardinal of T.
- `maxlength(`M`)` is the maximum length of term M (M must be a simple term without array access, and must be of a non-bounded type).
- `length(`$f, p_1, \ldots, p_n$`)` designates the maximal length of the result of a call to f, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of f (p_i must be built from `max`, `maxlength(`M`)`, and `length(`$f', \ldots$`)`, where M is a term of the type of the corresponding argument of f and the result of f' is of the type of the corresponding argument of f).
- `length`(T) designates the maximal length of a bitstring of type T, where T is a bounded type.

- `length(`$(T_1, \dots, T_n)$`,`$p_1, \dots, p_n$`)` designates the maximal length of the result of the tuple function from $T_1 \times \dots \times T_m$ to `bitstring`, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of this function.

- n is an integer constant.

- `eps_find` is the maximum distance between the uniform probability distribution and the probability distribution used for choosing elements in `find`.

- `eps_rand(`T`)` is the maximum distance between the uniform probability distribution and the default probability distribution D_T for type T (when T is `bounded`).

- `Pcoll1rand(`T`)` is the maximum probability of collision between a random value X of type T chosen according to the default distribution D_T for type T and an element of type T that does not depend on it (when T is `nonuniform`). This is also the maximum probability of choosing any given element of T in the default distribution for that type:

  $$\texttt{Pcoll1rand(}T\texttt{)} = \max_{a \in T} \Pr[X = a]$$

  where X is chosen according to distribution D_T.

- `Pcoll2rand(`T`)` is the maximum probability of collision between two independent random values of type T chosen according to the default distribution D_T for type T (when T is `nonuniform`). We have

  $$\texttt{Pcoll2rand(}T\texttt{)} = \sum_{a \in T} \Pr[X = a]^2 \leq \texttt{Pcoll1rand(}T\texttt{)}$$

  where X is chosen according to the default distribution D_T.

- `time` designates the runtime of the environment (attacker).

Finally, `time(...)` designates the runtime time of each elementary action of a game:

- `time(`$f, p_1, \dots, p_n$`)` designates the maximal runtime of one call to function symbol f, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of f.

- `time(let` $f, p_1, \dots, p_n$`)` designates the maximal runtime of one pattern matching operation with function symbol f, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of f.

- `time(`$(T_1, \dots, T_m)$`,`$p_1, \dots, p_n$`)` designates the maximal runtime of one call to the tuple function from $T_1 \times \dots \times T_m$ to `bitstring`, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of this function.

- `time(let(`$T_1, \dots, T_m$`),`$p_1, \dots, p_n$`)` designates the maximal runtime of one pattern matching with the tuple function from $T_1 \times \dots \times T_m$ to `bitstring`, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of this function.

- `time(=`$T[, p_1, p_2]$`)` designates the maximal runtime of one call to bitstring comparison function for bitstrings of type T, where p_1, p_2 represent the maximum length of the arguments of this function when T is non-bounded.

- `time(!)` is the maximum time of an access to a replication index.

- `time([`n`])` is the maximum time of an array access with n indexes.

- `time(&&)` is the maximum time of a boolean and.

- `time(||)` is the maximum time of a boolean or.

- `time(new` T`)` is the maximum time needed to choose a random number of type T according to the default distribution for type T.

- `time(newChannel)` is the maximum time to create a new private channel.
- `time(if)` is the maximum time to perform a boolean test.
- `time(find` n`)` is the maximum time to perform one condition test of a find with n indexes to choose. (Essentially, the time to store the values of the indexes in a list and part of the time needed to randomly choose an element of that list.)
- `time(out [`$T_1, \dots, T_m$`]`$T, p_1, \dots, p_n$`)` represents the time of an output in which the channel indexes are of types $T_1, \dots, T_m$, the output bitstring is of type T, and the maximum length of the channel indexes and the output bitstring is represented by $p_1, \dots, p_n$ when they are non-bounded.
- `time(in` n`)` is the maximum time to store an input in which the channel has n indexes in the list of available inputs.

CryptoVerif checks the dimension of probability formulas.

4 `oracles` Front-end

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. Keywords cannot be used as ordinary identifiers. The keywords are: `collision`, `const`, `define`, `defined`, `do`, `else`, `end`, `eps_find`, `eps_rand`, `equation`, `equiv`, `event`, `event_abort`, `expand`, `find`, `forall`, `foreach`, `fun`, `get`, `if`, `implementation`, `in`, `inj`, `insert`, `length`, `let`, `letfun`, `max`, `maxlength`, `newOracle`, `orfind`, `otheruses`, `param`, `Pcoll1rand`, `Pcoll2rand`, `proba`, `process`, `proof`, `query`, `return`, `secret`, `secret1`, `set`, `suchthat`, `table`, `then`, `time`, `type`.

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`.)

The input file consists of a list of declarations followed by an oracle definition:

$$\langle\text{declaration}\rangle^* \ \texttt{process}\ \langle\text{odef}\rangle$$

A library file (specified on the command-line by the `-lib` option) consists of a list of declarations. Various syntactic elements are described in Figures 3 and 4. The oracle definition describes the considered security protocol; the declarations specify in particular hypotheses on the cryptographic primitives and the security properties to prove.

Oracle definitions are described in a process calculus. In this calculus, terms represent computations on bitstrings. Simple terms consist of the following constructs:

- A term between parentheses `(`M`)` allows to disambiguate syntactic expressions.
- An identifier can be either a constant symbol f (declared by `const` or `fun` without argument) or a variable identifier.
- The function application f`(`$M_1, \dots, M_n$`)` applies function f to the result of $M_1, \dots, M_n$.
- The tuple application `(`$M_1, \dots, M_n$`)` builds a tuple from $M_1, \dots, M_n$ (corresponds to the concatenation of $M_1, \dots, M_n$ with length and type indications so that $M_1, \dots, M_n$ can be recovered without ambiguity). This is allowed only for $n \neq 1$, so that it is distinguished from parenthesing.
- The array access x`[`$M_1, \dots, M_n$`]` returns the cell of indexes $M_1, \dots, M_n$ of array x.
- `=`, `<>`, `||`, `&&` are function symbols that represent equality and inequality tests, disjunction and conjunction. They use the infix notation, but are otherwise considered as ordinary function symbols.

```
⟨identbound⟩ ::= ⟨ident⟩ <= ⟨ident⟩
⟨vartype⟩ ::= ⟨ident⟩:⟨ident⟩
⟨simpleterm⟩ ::= ⟨ident⟩
               | ⟨ident⟩(seq⟨simpleterm⟩)
               | (seq⟨simpleterm⟩)
               | ⟨ident⟩[seq⟨simpleterm⟩]
               | ⟨simpleterm⟩ = ⟨simpleterm⟩
               | ⟨simpleterm⟩ <> ⟨simpleterm⟩
               | ⟨simpleterm⟩ || ⟨simpleterm⟩
               | ⟨simpleterm⟩ && ⟨simpleterm⟩
⟨term⟩ ::= . . . (as in ⟨simpleterm⟩ with ⟨term⟩ instead of ⟨simpleterm⟩)
         | ⟨ident⟩ <-R ⟨ident⟩;⟨term⟩
         | ⟨ident⟩[:⟨ident⟩] <- ⟨term⟩
         | let ⟨pattern⟩ = ⟨term⟩ in ⟨term⟩ [else ⟨term⟩]
         | if ⟨cond⟩ then ⟨term⟩ else ⟨term⟩
         | find[[unique]] ⟨tfindbranch⟩ (orfind ⟨tfindbranch⟩)* else ⟨term⟩
         | event_abort ⟨ident⟩
⟨varref⟩ ::= ⟨ident⟩[seq⟨simpleterm⟩]
           | ⟨ident⟩
⟨cond⟩ ::= defined(seq+⟨varref⟩)  [&& ⟨term⟩]
         | ⟨term⟩
⟨tfindbranch⟩ ::= seq⟨identbound⟩ suchthat ⟨cond⟩ then ⟨term⟩
⟨pattern⟩ ::= ⟨ident⟩
            | ⟨vartype⟩
            | ⟨ident⟩(seq⟨pattern⟩)
            | (seq⟨pattern⟩)
            | =⟨term⟩
⟨event⟩ ::= [inj:]⟨ident⟩[(seq⟨simpleterm⟩)]
⟨queryterm⟩ ::= ⟨queryterm⟩ && ⟨queryterm⟩
              | ⟨queryterm⟩ || ⟨queryterm⟩
              | ⟨event⟩
              | ⟨simpleterm⟩
⟨query⟩ ::= secret ⟨ident⟩
          | secret1 ⟨ident⟩
          | [seq⟨vartype⟩;] event ⟨event⟩ (&& ⟨event⟩)* ==> ⟨queryterm⟩
```

where

- [M] means that M is optional; $(M)^*$ means that M occurs 0 or any number of times.
- 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.)
- seq$^+$⟨X⟩ is a non-empty sequence of X: seq$^+$⟨X⟩ = (⟨X⟩,)*⟨X⟩ = ⟨X⟩, . . . ,⟨X⟩. (It can be one or several elements of ⟨X⟩ separated by commas.)

Figure 3: Grammar for terms, patterns, and queries

```
⟨proba⟩ ::= (⟨proba⟩)
          | ⟨proba⟩ + ⟨proba⟩
          | ⟨proba⟩ - ⟨proba⟩
          | ⟨proba⟩ * ⟨proba⟩
          | ⟨proba⟩ / ⟨proba⟩
          | max(seq+⟨proba⟩)
          | ⟨ident⟩[(seq⟨proba⟩)]
          | |⟨ident⟩|
          | maxlength(⟨simpleterm⟩)
          | length(⟨ident⟩[, seq+⟨proba⟩])
          | length((seq⟨ident⟩)[, seq+⟨proba⟩])
          | n
          | #⟨ident⟩
          | eps_find
          | eps_rand(T)
          | Pcoll1rand(T)
          | Pcoll2rand(T)
          | time
          | time(⟨ident⟩[, seq+⟨proba⟩])
          | time(let ⟨ident⟩[, seq+⟨proba⟩])
          | time((seq⟨ident⟩)[, seq+⟨proba⟩])
          | time(let (seq⟨ident⟩)[, seq+⟨proba⟩])
          | time(= ⟨ident⟩[, seq+⟨proba⟩])
          | time(!)
          | time([n])
          | time(&&)
          | time(||)
          | time(<-R ⟨ident⟩)
          | time(newOracle)
          | time(if)
          | time(find n)

⟨ogroup⟩ ::= ⟨ident⟩(seq⟨vartype⟩) [[n]] [[useful_change]] := ⟨obody⟩
           | foreach ⟨ident⟩ <= ⟨ident⟩ do (⟨ident⟩ <-R ⟨ident⟩;)*⟨ogroup⟩
           | foreach ⟨ident⟩ <= ⟨ident⟩ do (⟨ident⟩ <-R ⟨ident⟩;)*(⟨ogroup⟩ | ... | ⟨ogroup⟩)
⟨omode⟩ ::= ⟨ogroup⟩ [[exist]]
          | ⟨ogroup⟩ [all]

⟨channel⟩ ::= ⟨ident⟩
⟨obody⟩ ::= ⟨ident⟩
          | (⟨obody⟩)
          | end
          | event ⟨ident⟩[(seq⟨term⟩)] [; ⟨obody⟩]
          | event_abort ⟨ident⟩
          | ⟨ident⟩ <-R ⟨ident⟩[; ⟨obody⟩]
          | ⟨ident⟩[:⟨ident⟩] <- ⟨term⟩[; ⟨obody⟩]
          | let ⟨pattern⟩ = ⟨term⟩ [in ⟨obody⟩ [else ⟨obody⟩]]
          | if ⟨cond⟩ then ⟨obody⟩ [else ⟨obody⟩]
          | find[[unique]] ⟨findbranch⟩ (orfind ⟨findbranch⟩)* [else ⟨obody⟩]
          | insert ⟨ident⟩(seq⟨term⟩) [; ⟨obody⟩]
          | get ⟨ident⟩(seq⟨pattern⟩) [suchthat ⟨term⟩] in ⟨obody⟩ [else ⟨obody⟩]
          | return(seq⟨term⟩)[; ⟨odef⟩]
⟨findbranch⟩ ::= seq⟨identbound⟩ suchthat ⟨cond⟩ then ⟨obody⟩
⟨odef⟩ ::= ⟨ident⟩
         | (⟨odef⟩)
         | 0
         | ⟨odef⟩ | ⟨odef⟩
         | foreach ⟨ident⟩ <= ⟨ident⟩ do ⟨odef⟩
         | ⟨ident⟩(seq⟨pattern⟩) := ⟨obody⟩
```

Figure 4: Grammar for probabilities, equivalences, and processes

Terms contain further constructs `<-R`, `<-`, `event_abort`, `let`, `if`, and `find` which are similar to the corresponding constructs of oracle bodies. They are not allowed to occur in `defined` conditions of `find`. The constructs `<-R` and `event_abort` are not allowed to occur in conditions of `find` or `get`. The construct `find` is also not allowed in conditions of `get`. We refer the reader to the description of oracle bodies below for a fully detailed explanation.

- x `<-R` T`;`M chooses a new random number in type T, stores it in x, and returns the result of M.
- $x[: T]$ `<-` M`;`M' stores the result of M in x and returns the result of M'. This is equivalent to the construct `let` $x[: T]$ `=` M `in` M' below.
- `let` p `=` M `in` M' `else` M'' tries to decompose the term M according to pattern p. In case of success, returns the result of M', otherwise the result of M''.

  The pattern p can be:

  - x[`:`T] variable, possibly with its type. Matches any bitstring (in type T), and stores it in x.
  - f`(`$p_1, \dots, p_n$`)` where the function symbol f is declared `[compos]`. Matches bitstrings M equal to $f(M_1, \dots, M_n)$ for some $M_1, \dots, M_n$ that match $p_1, \dots, p_n$. (The poly-injectivity of f allows us to compute possible values $M_1, \dots, M_n$ of its arguments from the value of M, and to check whether M is equal to the resulting value of $f(M_1, \dots, M_n)$.)
  - `(`$p_1, \dots, p_n$`)` tuples, which are particular `[compos]` functions encoding unambiguously the values of $p_1, \dots, p_n$ and their type.
  - `=`M' matches a bitstring equal to M'.

  When p is a variable, the `else` branch can be omitted (it cannot be executed).

- `if` *cond* `then` M `else` M' is syntactic sugar for `find suchthat` *cond* `then` M `else` M'. It returns the result of M if the condition *cond* evaluates to `true` and of M' if *cond* evaluates to `false`.
- `find` FB_1 `orfind ... orfind` FB_m `else` M where $FB_j = u_{j1}$`<=`$n_{j1}, \dots, u_{jm_j}$`<=`n_{jm_j} `suchthat` *cond*$_j$ `then` M_j evaluates the conditions *cond*$_j$ for each j and each value of $u_{j1}, \dots, u_{jm_j}$ in $[1, n_{j1}] \times \dots \times [1, n_{jm_j}]$. If none of these conditions is `true`, it returns the result of M. Otherwise, it chooses randomly with (almost) uniform probability one j and one value of $u_{j1}, \dots, u_{jm_j}$ such that the corresponding condition is `true`, and returns the result of M_j. See the explanation of the `find` process below for more details.
- `event_abort` e executes event e and aborts the game. It is intended to be used in the right-hand side of the definitions of some cryptographic primitives. (See also the `equiv` declaration; events in the right-hand side can be used when the simulation of left-hand side by the right-hand side fails. CryptoVerif is going bound the probability that the event is executed and include it in the probability of success of an attack.)

The calculus distinguishes two kinds of processes: oracle definitions ⟨odef⟩ define new oracles; oracle bodies ⟨obody⟩ return a result after executing some internal computations. When a process (oracle definition or oracle body) is an identifier, it is substituted with its value defined by a `let` declaration. Processes allow parenthesing for disambiguation.

Let us first describe oracle definitions:

- `0` does nothing.
- Q `|` Q' is the parallel composition of Q and Q'.
- `foreach` i`<=`N `do` Q represents N copies of Q in parallel each with a different value of $i \in [1, N]$; it means that the oracles defined in Q are available N times. The identifier N must have been declared by `param` N. The identifier i cannot be referred to explicitly in the process; it is used only implicitly as array index of variables defined under the replication `foreach` i`<=`N.

  When a program point is under replications `foreach` i_1`<=`N_1, ..., `foreach` i_n`<=`N_n, the *current replication indexes* at that point are $i_1, \dots, i_n$.

- `O(`$p_1, \dots, p_n$`) :=` P defines an oracle O taking arguments $p_1, \dots, p_n$, and returning the result of the oracle body P. The patterns $p_1, \dots, p_n$ are as in the `let` construct above, except that variables in p that are not under a function symbol $f(\dots)$ must be declared with their type.

Note that the construct **newOracle** $c; Q$ used in research papers is absent from the implementation: this construct is useful in the proof of soundness of CryptoVerif, but not essential for encoding games that CryptoVerif manipulates.

Let us now describe output processes:

- `end` terminates the oracle, returning control to the caller.
- `event` e`(`$M_1, \dots, M_n$`);`P executes the event e`(`$M_1, \dots, M_n$`)`, then executes P. Events serve in recording the execution of certain parts of the program for using them in queries. The symbol e must have been declared by an `event` declaration.
- `event_abort` e executes event e and terminates the game. (Nothing can be executed after this instruction, neither by the protocol nor by the adversary.) The symbol e must have been declared by an `event` declaration, without any argument.
- x `<-R` T`;`P chooses a new random number in type T, stores it in x, and executes P. T must be declared with option `fixed`, `bounded`, or `nonuniform`. Each such type T comes with an associated default probability distribution D_T; the random number is chosen according to that distribution. The time for generated random numbers in that distribution is bounded by `time(<-R` T`)`.
  - When the type T is `nonuniform`, the default probability distribution D_T for type T may be non-uniform. It is left unspecified. (Notice that random bitstrings with non-uniform distributions can also be obtained by applying a function to a random bitstring choosen uniformly among a finite set of bitstrings, chosen in another type.)
  - When the type T is `fixed`, it consists of the set of all bitstrings of a certain length n. Probabilistic Turing machines can return uniformly distributed random numbers in such types, in bounded time. If T is not marked `nonuniform`, the default probability distribution D_T for T is the uniform distribution.
  - For other `bounded` types T, probabilistic bounded-time Turing machines can choose random numbers with a distribution as close as we wish to uniform, but may not be able to produce exactly a uniform distribution. If T is not marked `nonuniform`, the default probability distribution D_T is such that its distance to the uniform distribution is at most `eps_rand(`T`)`. The distance between two probability distributions D_1 and D_2 for type T is

    $$d(D_1, D_2) = \sum_{a \in T} |\Pr[X_1 = a] - \Pr[X_2 = a]|$$

    where X_i $(i = 1, 2)$ is a random variable of distribution D_i.
    For example, a possible algorithm to obtain a random integer in $[0, m-1]$ is to choose a random integer x' uniformly among $[0, 2^k - 1]$ for a certain k large enough and return $x' \bmod m$. By euclidian division, we have $2^k = qm + r$ with $r \in [0, m-1]$. With this algorithm

    $$\Pr[x = a] = \begin{cases} \frac{q+1}{2^k} & \text{if } a \in [0, r-1] \\ \frac{q}{2^k} & \text{if } a \in [r, m-1] \end{cases}$$

    so

    $$\left|\Pr[x = a] - \frac{1}{m}\right| = \begin{cases} \frac{q+1}{2^k} - \frac{1}{m} & \text{if } a \in [0, r-1] \\ \frac{1}{m} - \frac{q}{2^k} & \text{if } a \in [r, m-1] \end{cases}$$

    Therefore

    $$\begin{aligned} d(D_T, \mathit{uniform}) &= \sum_{a \in T} \left|\Pr[x = a] - \frac{1}{m}\right| = r\left(\frac{q+1}{2^k} - \frac{1}{m}\right) - (m - r)\left(\frac{1}{m} - \frac{q}{2^k}\right) \\ &= \frac{2r(m-r)}{m.2^k} \leq \frac{m}{2^k} \end{aligned}$$

so we can take $\texttt{eps\_rand(}T\texttt{)} = \frac{m}{2^k}$. A given precision of $\texttt{eps\_rand(}T\texttt{)} = \frac{1}{2^{k'}}$ can be obtained by choosing $k = k' +$ number of bits of m random bits.

When `ignoreSmallTimes` is set to a value greater than 0 (which is the default), the time for random number generations and the probability $\texttt{eps\_rand(}T\texttt{)}$ are ignored, to make probability formulas more readable.

- $x[\texttt{:}T]$ `<-` $M\texttt{;}P$ stores the result of term M in x and executes P. M must be of type T when T is mentioned. This is equivalent to the construct `let` $x[\texttt{:}T]$ `=` M `in` P below.

- `let` p `=` M `in` P `else` P' tries to decompose the term M according to pattern p. In case of success, executes P, otherwise executes P'.

  The pattern p can be:

  - $x[\texttt{:}T]$ variable, possibly with its type. Matches any bitstring (in type T), and stores it in x.
  - $f\texttt{(}p_1, \dots, p_n\texttt{)}$ where the function symbol f is declared `[compos]`. Matches bitstrings M equal to $f(M_1, \dots, M_n)$ for some $M_1, \dots, M_n$ that match $p_1, \dots, p_n$. (The poly-injectivity of f allows us to compute possible values $M_1, \dots, M_n$ of its arguments from the value of M, and to check whether M is equal to the resulting value of $f(M_1, \dots, M_n)$.)
  - $\texttt{(}p_1, \dots, p_n\texttt{)}$ tuples, which are particular `[compos]` functions encoding unambiguously the values of $p_1, \dots, p_n$ and their type.
  - `=`M' matches a bitstring equal to M'.

  The `else` clause is never executed when the pattern is simply a variable. When `else` P' is omitted, it is equivalent to `else end`. Similarly, when `in` P is omitted, it is equivalent to `in end`.

- `if` *cond* `then` P `else` P' is syntactic sugar for `find suchthat` *cond* `then` P `else` P'. It executes P if the condition *cond* evaluates to `true` and P' if *cond* evaluates to `false`. When the `else` clause is omitted, it is implicitly `else end`.

- Next, we explain the process `find` FB_1 `orfind` $\dots$ `orfind` FB_m `else` P where each branch FB_j is $FB_j = u_{j1}\texttt{<=}n_{j1}, \dots, u_{jm_j}\texttt{<=}n_{jm_j}$ `suchthat` $cond_j$ `then` P_j.

  A simple example is the following: `find` $u\texttt{<=}n$ `suchthat` $\texttt{defined(}x[u]\texttt{)}$ `&&` $x[u] = a$ `then` P' `else` P tries to find an index u such that $x[u]$ is defined and $x[u] = a$, and when such a u is found, it executes P' with that value of u; otherwise, it executes P. In other words, this `find` construct looks for the value a in the array x, and when a is found, it stores in u an index such that $x[u] = a$. Therefore, the `find` construct allows us to access arrays, which is key for our purpose.

  More generally, `find` $u_1\texttt{<=}n_1, \dots, u_m\texttt{<=}n_m$ `suchthat` $\texttt{defined(}M_1, \dots, M_l\texttt{)}$ `&&` M `then` P' `else` P tries to find values of $u_1, \dots, u_m$ for which $M_1, \dots, M_l$ are defined and M is true. In case of success, it executes P'. In case of failure, it executes P.

  This is further generalized to m branches: `find` FB_1 `orfind` $\dots$ `orfind` FB_m `else` P where $FB_j = u_{j1}\texttt{<=}n_{j1}, \dots, u_{jm_j}\texttt{<=}n_{jm_j}$ `suchthat` $\texttt{defined(}M_{j1}, \dots, M_{jl_j}\texttt{)}$ `&&` M_j `then` P_j tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \dots, u_{jm_j}$ for which $M_{j1}, \dots, M_{jl_j}$ are defined and M_j is true. In case of success, it executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions $cond_j = \texttt{defined(}M_{j1}, \dots, M_{jl_j}\texttt{)}$ `&&` M_j for each j and each value of $u_{j1}, \dots, u_{jm_j}$ in $[1, n_{j1}] \times \dots \times [1, n_{jm_j}]$. If none of these conditions is `true`, it executes P. Otherwise, it chooses randomly with almost uniform probability[2] one j and one value of $u_{j1}, \dots, u_{jm_j}$ such that the corresponding condition is `true`, and executes P_j.

  In the general case, the conditions $cond_j$ are of the form $\texttt{defined(}M_1, \dots, M_l\texttt{)}$ [`&&` M] or simply M. The condition $\texttt{defined(}M_1, \dots, M_l\texttt{)}$ means that $M_1, \dots, M_l$ are defined. At least one of the two conditions `defined` or M must be present. Omitted `defined` conditions are considered empty; when M is omitted, it is considered `true`.

[2] Precisely, the distance between the distribution actually used for choosing $j, u_{j1}, \dots, u_{jm_j}$ and the uniform distribution is at most `eps_find`. See the explanation of x `<-R` T for details on how to achieve this.

Internally, CryptoVerif distinguishes two variables for each index of find, so that the syntax of a find branch becomes $FB_j = u_{j1} = u'_{j1}$<=$n_{j1}, \dots, u_{jm_j} = u'_{jm_j}$<=$n_{jm_j}$ suchthat defined($M_{j1}, \dots, M_{jl_j}$) && M_j then P_j. The variables $u'_{j1}, \dots, u'_{jm_j}$ are considered as replication indices, and are used in the defined condition and in M_j: they are temporary variables that are used as loop indices to look for indices that satisfy the desired conditions. Once suitable indices are found, their value is stored in $u_{j1}, \dots, u_{jm_j}$ and the then branch is executed using these variables. It is possible to make array accesses to $u_{j1}, \dots, u_{jm_j}$ (such as $u_{j1}[M_1, \dots, M_k]$) elsewhere in the game, which is not possible for $u'_{j1}, \dots, u'_{jm_j}$.

A variant of find is find[unique]. Consider the process find[unique] FB_1 orfind ... orfind FB_m else P where $FB_j = u_{j1}$<=$n_{j1}, \dots, u_{jm_j}$<=n_{jm_j} suchthat defined($M_{j1}, \dots, M_{jl_j}$) && M_j then P_j. When there are several values of $j, u_{j1}, \dots, u_{jm_j}$ for which $M_{j1}, \dots, M_{jl_j}$ are defined and M_j is true, this process executes an event NonUnique and aborts the game. In all other cases, it behaves as find. Intuitively, find[unique] should be used when there is a negligible probability of finding several suitable values of $j, u_{j1}, \dots, u_{jm_j}$. The construct find[unique] is typically not used in the initial game. (One would have to prove manually that there is indeed a negligible probability of finding several suitable values of $j, u_{j1}, \dots, u_{jm_j}$. CryptoVerif displays a warning if find[unique] occurs in the initial game.) However, find[unique] is used in the specification of cryptographic primitives, in the right-hand of equivalences specified by equiv.

- insert *tbl*($M_1, \dots, M_n$); P inserts the tuples $(M_1, \dots, M_n)$ in the table *tbl*, then executes P. The table *tbl* must have been declared with the appropriate types using the table declaration.

- get *tbl*($p_1, \dots, p_n$) suchthat M in P else P' tries to find an element of the table *tbl* that matches the patterns $p_1, \dots, p_n$ and such that M is true. If it succeeds, it executes P with the variables of $p_1, \dots, p_n$ bound to that element of the table. If several elements match, one of them is chosen randomly with (almost) uniform probability. If no element matches, it executes P'.

  When else P' is omitted, it is equivalent to else end. When suchthat M is omitted, it is equivalent to suchthat *true*. Internally, get is converted into find by CryptoVerif.

- return($N_1, \dots, N_l$);Q terminates the oracle, returning the result of the terms $N_1, \dots, N_l$. Then, it makes available the oracles defined in Q.

In this calculus, all variables are implicitly arrays. When a variable x is defined (by <-R, <-, let, find, and oracle definitions) under replications foreach i_1<=N_1, ..., foreach i_n<=N_n, x has implicitly indexes $i_1, \dots, i_n$: x stands for $x[i_1, \dots, i_n]$. Arrays allow us to have full access to the state of the process. Arrays can be read using find. Similarly, when x is used with $k < n$ indexes the missing $n - k$ indexes are implicit: $x[u_1, \dots, u_k]$ stands for $x[i_1, \dots, i_{n-k}, u_1, \dots, u_k]$ where $i_1, \dots, i_{n-k}$ must be the $n-k$ first replication indexes both at the creation of x and at the usage $x[u_1, \dots, u_k]$. (So the usage and creation of x must be under the same $n - k$ top-most replications.) When an oracle O is defined under foreach i_1<=N_1, ..., foreach i_n<=N_n, it also implicitly defines $O[i_1, \dots, i_n]$.

In the initial game, several variables may be defined with the same name, but they are immediately renamed to different names, so that after renaming, each variable is defined once. When several variables are defined with the same name, they can be referenced only under their definition without explicit array indexes, because for other references, we would not know which variable to reference after renaming.

In subsequent games created by CryptoVerif, a variable may be defined at several occurrences, but these occurrences must be in different branches of if, find, or let, so that they cannot be executed with the same value of the array indexes. This constraint guarantees that each array cell is defined at most once.

Each usage of x must be either:

- x without array index syntactically under its definition. (Then x is implicitly considered to have as indexes the current replication indexes at its definition.)

- x possibly with array indexes inside the defined condition of a find.

- $x[M_1, \dots, M_n]$ in M or P in a find branch ... suchthat defined($M'_1, \dots, M'_l$) && M then P, such that $x[M_1, \dots, M_n]$ is a subterm of $M'_1, \dots, M'_l$.

- $x[M_1, \dots, M_n]$ in M or M'' in a find branch ... `suchthat defined(`$M'_1, \dots, M'_l$`) &&` M `then` M'', such that $x[M_1, \dots, M_n]$ is a subterm of $M'_1, \dots, M'_l$.

These syntactic constraints guarantee that a variable is accessed only when it is defined. Moreover, the variables defined in conditions of `find` or in patterns or conditions of `get` must not have array accesses (that is, accesses corresponding to the last three cases above).

Finally, the calculus is equipped with a type system. To be able to use variables outside their scope (by `find`), the type checking algorithm works in two passes.

In the first pass, it collects the type of each variable: when a variable x is defined with type T under `foreach` i_1`<=`N_1, ..., `foreach` i_n`<=`N_n, x has type $[1, N_1] \times \ldots \times [1, N_n] \rightarrow T$. When the type of x is not explicitly given in its declaration (in `<-` or in patterns in `let` or oracle definitions), its type is left undefined in this pass, and x cannot be used outside its scope.

In the second pass, the type system checks the following requirements: In x`[`$M_1, \dots, M_m$`]`, $M_1, \dots, M_m$ must be of the suitable interval type, that is, a suffix of the types of replication indexes at the definition of x. In f`(`$M_1, \dots, M_m$`)`, if f has been declared by `fun` f`(`$T_1, \dots, T_m$`):`T, M_j must be of type T_j, and f`(`$M_1, \dots, M_m$`)` is then of type T. In `(`$M_1, \dots, M_n$`)`, M_j can be of any bitstring type (that is, not an index type $[1, N]$), and the result is of type `bitstring`. In M_1 `=` M_2 and M_1 `<>` M_2, M_1 and M_2 must be of the same type, and the result is of type `bool`. In M_1 `||` M_2 and M_1 `&&` M_2, M_1 and M_2 must be of type `bool` and the result is of type `bool`. The type system requires each subterm to be well-typed. Furthermore, in `event` e`(`$M_1, \dots, M_n$`)`, if e has been declared by `event` e`(`$T_1, \dots, T_n$`)`, M_j must be of type T_j. In x `<-R` T, T must be declared with option `bounded` (or `fixed`). In `if` M `then ... else ...`, M must be of type `bool`. Similarly, for

`find ... orfind ... suchthat defined(...) &&` M `then ...`

M must be of type `bool`. In `let` p `=` M `in ...`, M and p must be of the same type. For function application and tuple patterns, the typing rule is the same as for the corresponding terms. The pattern $x : T$ is of type T; the pattern x can be of any bitstring type, determined by the usage of x (when the pattern x is used as argument of a tuple pattern, its type is `bitstring`); the pattern `=`M is of the type of M. In `return(`$M_1, \dots, M_n$`)`, M_j must be of a bitstring type T_j for all $j \leq n$ and that return instruction is said to be of type $T_1 \times \ldots \times T_n$. All return instructions in an oracle body P (excluding return instructions that occur in oracle definitions Q in processes of the form `return(`$M_1, \dots, M_n$`);`Q) must be of the same type, and that type is said to be the type of the oracle body P. For each oracle definition O`(`$p_1, \dots, p_m$`) :=` P under `foreach` i_1`<=`N_1, ..., `foreach` i_n`<=`N_n, the oracle O is said to be of type $[1, N_1] \times \ldots \times [1, N_n] \rightarrow T'_1 \times \ldots \times T'_m \rightarrow T_1 \times \ldots \times T_n$ where p_j is of type T'_j for all $j \leq m$ and P is of type $T_1 \times \ldots \times T_n$. When an oracle has several definitions, it must be of the same type for all its definitions. Furthermore, definitions of the same oracle O must not occur on both sides of a parallel composition Q`|`Q' (so that several definitions of the same oracle cannot be simultaneously available).

A declaration can be:

- `set` ⟨parameter⟩ `=` ⟨value⟩.

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

  - `set diffConstants = true.`
    `set diffConstants = false.`
    When `true`, different constant symbols are assumed to have a different value. When `false`, CryptoVerif does not make this assumption.
  - `set constantsNotTuple = true.`
    `set constantsNotTuple = false.`
    When `true`, constant symbols are assumed to be different from the result of applying a tuple function to any argument. When `false`, CryptoVerif does not make this assumption.
  - `set expandAssignXY = true.`
    `set expandAssignXY = false.`
    When `true`, CryptoVerif automatically removes assignments `x <- y` where `x` and `y` are variables by substituting `y` for `x` (in the transformation `remove_assign useless`) When `false`, this transformation is not performed as part of `remove_assign useless`.

- `set minimalSimplifications = true.`
  `set minimalSimplifications = false.`

  When `true`, simplification replaces a term with a rewritten term only when the rewriting has used at least one rewriting rule given by the user, not when only equalities that come from `let` definitions and other instructions in the game have been used. When `false`, a term is replaced with its rewritten form in all cases. The latter configuration often leads to replacing a term with a more complex one, in particular expanding `let` definitions, thus duplicating their contents.

- `set mergeBranches = true.`
  `set mergeBranches = false.`

  When `true`, the transformation `merge_branches` is applied after simplification, to merge branches of `if`, `let`, and `find` when all branches execute the same code. This is useful in order to remove the test, which can remove a use of a secret. When `false`, this transformation is not performed. This is useful in particular when the test has been manually introduced in order to force CryptoVerif to distinguish cases.

- `set mergeArrays = true.`
  `set mergeArrays = false.`

  When `true`, `merge_branches` advises `merge_arrays` commands to make the merging of branches of `if`, `find`, `let` succeed more often. When `false`, this advice is not automatically given and the user should use the manual command `merge_arrays` (defined in Section 7) to perform the merging.

- `set uniqueBranch = true.`
  `set uniqueBranch = false.`

  When `uniqueBranch = true`, the following transformation is enabled as part of `simplify`: if a branch of a `find[unique]` is proved to succeed, then simplification removes all other branches of that `find`. When `uniqueBranch = false`, this transformation is not performed.

- `set uniqueBranchReorganize = true.`
  `set uniqueBranchReorganize = false.`

  When `uniqueBranchReorganize = true`, the following transformations are enabled as part of `simplify`:

  ∗ If a `find[unique]` occurs in the `then` branch of a `find[unique]`, we reorganize them.

  ∗ If a `find[unique]` occurs in the condition of a `find`, we reorganize them.

  When `uniqueBranchReorganize = false`, these transformations are not performed.

- `set autoSARename = true.`
  `set autoSARename = false.`

  When `true`, and a variable is defined several times and used only in the scope of its definition with the current replication indexes at that definition, each definition of this variable is renamed to a different name, and the uses are renamed accordingly, by the transformation `remove_assign`. When `false`, such a renaming is not done automatically, but in manual proofs, it can be requested specifically for each variable by `SArename x`, where `x` is the name of the variable.

- `set autoMove = true.`
  `set autoMove = false.`

  When `true`, the transformation `move all` is automatically executed after each cryptographic transformation. This transformation moves random number generations `<-R` downwards as much as possible, duplicating them when crossing a `if`, `let`, or `find`. (A future `SArename` transformation may then enable us to distinguish cases depending on which of the duplicated random number generations a variable comes from.) It also moves assignments down in the syntax tree but without duplicating them, when the assignment can be moved under a `if`, `let`, or `find`, in which the assigned variable is used only in one branch. (In this case, the assigned term is computed in fewer cases thanks to this transformation.)

  When `false`, the transformation `move all` is never automatically executed.

- `set optimizeVars = false.`
  `set optimizeVars = true.`

  When `true`, CryptoVerif tries to reduce the number of different intermediate variables introduced in cryptographic transformations. This can lead to distinguishing fewer cases, which unfortunately often leads to a failure of the proof. When `false`, different intermediate varaibles are used for each occurrence of the transformed expression.

- `set interactiveMode = false.`
  `set interactiveMode = true.`

  When `false`, CryptoVerif runs automatically. When `true`, CryptoVerif waits for instructions of the user on how to perform the proof. (See Section 7 for details on these instructions.) This setting is ignored when proof instructions are included in the input file using the `proof` command. In this case, the instructions given in the `proof` command are executed, without user interaction.

- `set autoAdvice = true.`
  `set autoAdvice = false.`

  In interactive mode, when `autoAdvice = true`, execute the advised transformations automatically. When `autoAdvice = false`, display the advised transformations, but do not execute them. The user may then give them as instructions if he wishes.

- `set noAdviceCrypto = false.`
  `set noAdviceCrypto = true.`

  When `noAdviceCrypto = true`, prevents the cryptographic transformations from generating advice. Useful mainly for debugging the proof strategy.

- `set noAdviceGlobalDepAnal = false.`
  `set noAdviceGlobalDepAnal = true.`

  When `noAdviceGlobalDepAnal = true`, prevents the global dependency analysis from generating advice. Useful when the global dependency analysis generates bad advice.

- `set simplifyAfterSARename = true.`
  `set simplifyAfterSARename = false.`

  When `simplifyAfterSARename = true`, apply simplification after each execution of the SArename transformation. This slows down the system, but enables it to succeed more often.

- `set backtrackOnCrypto = false.`
  `set backtrackOnCrypto = true.`

  When `backtrackOnCrypto = true`, use backtracking when the proof fails, to try other cryptographic transformations. This slows down the system considerably (so it is false by default), but enables it to succeed more often, in particular for public-key protocols that mix several primitives. One usage is to try first with the default setting and, if the proof fails although the property is believed to hold, try again with backtracking.

- `set useKnownEqualitiesInCryptoTransform = true.`
  `set useKnownEqualitiesInCryptoTransform = false.`

  When `useKnownEqualitiesInCryptoTransform = true`, CryptoVerif relies on known equalities between terms to replace variables with their values in the cryptographic transformations. When it is false, CryptoVerif just uses the variables as their appear in the game, and relies only on advice to replace variables with their values.

- `set ignoreSmallTimes =` ⟨n⟩. (default 3)

  When `0`, the evaluation of the runtime is very precise, but the formulas are often too complicated to read.

  When `1`, the system ignores many small values when computing the runtime of the games. It considers only function applications and pattern matching.

  When `2`, the system ignores even more details, including application of boolean operations (`&&`, `||`, `not`), constants generated by the system, `()` and matching on `()`. It ignores the creation and decomposition of tuples in oracle calls and returns.

When `3`, the system additionally ignores the time of equality tests between values of bounded length, as well as the time of all constants.

- `set maxIterSimplif =` ⟨n⟩`.` (default 2)

  Sets the maximum number of repetitions of the simplification transformation for each `simplify` instruction. A greater value slows down the system but may enable it to obtain simpler games, and therefore increase its chances of success. When $n \leq 0$, repeats simplification until a fixpoint is reached.

- `set maxAddFactDepth =` ⟨n⟩`.` (default 1000)

  Sets the maximum depth of recursion in the addition and simplification of known facts. When $n \leq 0$, puts no limit on this depth of recursion. Putting a limit avoids an infinite loop in some rare cases.

- `set maxIterRemoveUselessAssign =` ⟨n⟩`.` (default 10)

  Sets the maximum number of repetitions of the removal of useless assignments for each `remove_assign useless` instruction. A greater value slows down the system but may enable it to obtain simpler games, and therefore increase its chances of success. When $n \leq 0$, repeats removal of useless assignments until a fixpoint is reached.

- `set minAutoCollElim =` ⟨s⟩`.` (default `size15`)

  Sets the minimum size of a type for which elimination of collisions is possible automatically. The size argument ⟨s⟩ can be `large`, `password`, or `size`n (see the `type` declaration for their meaning).

- `set maxAdvicePossibilitiesBeginning =` n_1`.` (default `50`)
  `set maxAdvicePossibilitiesEnd =` n_2`.` (default `10`)

  In cryptographic transformations, when CryptoVerif can transform many terms in several ways of different priority, these various ways combine, yielding a very large number of advice possibilities. These two options allow to limit the number of considered advice possibilities by keeping the n_1 first possibilities (with highest priority) and the n_2 last possibilities (with lowest priority but fewer advised transformations). When n_1 or n_2 are not positive, all advice possibilities are kept, but that may yield a very slow execution.

- `set elsefindFactsInReplace = true.`
  `set elsefindFactsInReplace = false.`

  When `elsefindFactsInReplace = true`, CryptoVerif will try to infer more facts when doing a `replace` operation: when it encounters a `find` branch in the process, it considers a variable $x[M_1, \dots, M_l]$, which is guaranteed to be defined by this `find`. If x is defined in the `else` part of another `find` construct, then at the definition of x, we know that the conditions of the `then` branches of this `find` are not satisfied:

  $$\forall u_1, \dots, u_k, \mathtt{not}(\mathtt{defined}(y_1[M_{11}, \dots, M_{1l_1}], \dots, y_k[M_{k1}, \dots, M_{kl_k}]) \wedge t)$$

  We try to infer `not`(t) from this fact.

  * if each variable $y_j[M_{j1}, \dots, M_{jl_j}]$ is defined before $x[M_1, \dots, M_l]$, then `not`(t) indeed holds by the fact above;
  * for each $y_j[M_{j1}, \dots, M_{jl_j}]$, we assume that $y_j[M_{j1}, \dots, M_{jl_j}]$ is defined after $x[M_1, \dots, M_l]$ and try to prove `not`(t).
    It this proof succeeds, we can infer that `not`(t) holds at the current program point.

- `set elsefindFactsInSimplify = true.`
  `set elsefindFactsInSimplify = false.`

  Similar to `elsefindFactsInReplace`, but applies in `simplify` operations.

- `set maxReplaceDepth =` n`.` (default 20)

  Sets the maximum number of rewriting steps that are allowed to prove that the new term is equal to the old one in a `replace` transformation.

The default value is the first mentioned, except when explicitly specified. In most cases, the default values should be left as they are, except for `interactiveMode`, which allows to perform interactive proofs.

- `param` seq$^+$⟨ident⟩ [`[noninteractive]` | `[passive]` | `[size`n`]`]`.`

  `param` $n_1, \dots, n_m$`.` declares parameters $n_1, \dots, n_m$. Parameters are used to represent the number of copies of replicated processes (that is, the maximum number of calls to each query). In asymptotic analyses, they are polynomial in the security parameter. In exact security analyses, they appear in the formulas that express the probability of an attack.

  The options `[noninteractive]`, `[passive]`, or `[size`n`]` indicate to CryptoVerif an order of magnitude of the size of the parameter. The option `[size`n`]` (where n is a constant integer) indicates that the considered parameter has "size n": the larger the n, the larger the parameter is likely to be. CryptoVerif uses this information to optimize the computed probability bounds: when several bounds are correct, it chooses the smallest one.

  The option `[noninteractive]` means that the queries bounded by the considered parameters can be made by the adversary without interacting with the tested protocol, so the number of such queries is likely to be large. Parameters with option `[noninteractive]` are typically used for bounding the number of calls to random oracles. `[noninteractive]` is equivalent to `[size20]`.

  The option `[passive]` means that the queries bounded by the considered parameters correspond to the adversary passively listening to sessions of the protocol that run as expected. Therefore, for such runs, the adversary is undetected. This number of runs is therefore likely to be larger than runs in which the adversary actively interacts with the honest participants, when these participants stop after a certain number of failed attempts. `[passive]` is equivalent to `[size10]`.

- `proba` ⟨ident⟩`.`

  `proba` p`.` declares a probability p. (Probabilities may be used as functions of other arguments, without explicit checking of these arguments.)

- `type` ⟨ident⟩ [`[`seq$^+$⟨option⟩`]`]`.`

  `type` T`.` declares a type T. Types correspond to sets of bitstrings or a special symbol $\perp$ (used for failed decryptions, for instance). Optionally, the declaration of a type may be followed by options between brackets. These options can be:

  – `bounded` means that the type is a set of bitstrings of bounded length or perhaps $\perp$. In other words, the type is a finite subset of bitstrings plus $\perp$.

  – `fixed` means that the type is the set of all bitstrings of a certain length n. In particular, the type is a finite set, so `fixed` implies `bounded`.

  – `nonuniform` means that random numbers may be chosen in the type with a non-uniform distribution. (When `nonuniform` is absent, random numbers are chosen using a uniform distribution for `fixed` types, an almost uniform distribution for `bounded` types, and random values cannot be chosen among other types.)

  – `large`, `password`, and `size`n indicate the order of magnitude of the probability of collision `Pcoll1rand`(T) between a random element chosen according to the default probability distribution D_T for the considered type T, and an independent element of type T. When the default distribution is uniform or almost uniform (`fixed` and `bounded` types), `Pcoll1rand`$(T) = \frac{1}{|T|}$, so these parameters give an order of magnitude of the cardinal of the type.

    The option `size`n (where n is a constant integer) indicates that the considered type has "size n": the larger the n, the smaller the probability of collision `Pcoll1rand`(T). When no size option is present, the type has size 0. CryptoVerif uses this information to determine whether collisions with random elements of the considered type T should be eliminated. For collisions to be eliminated, two conditions must be satisfied:

    1. the size of the type must be at least `minAutoCollElim` (which is set by `set minAutoCollElim =` n; the default is 15), or the size of the type must be at least 1 and elimination of collisions on this data has been manually requested by the command `simplify coll_elim` ....

2. the probability of collision is at most one of the formulas specified by the command `allowed_collisions` (used inside a `proof` environment). By default, all collisions are eliminated for types of size at least 20, and collisions are eliminated for types of size at least 10 when the collision is repeated at most N times, where N is a parameter of size 0. See the command `allowed_collisions` for more details.

`large` means that the type T is large enough so that all collisions with random elements of T can be eliminated. (In asymptotic analyses, `Pcoll1rand`(T) is negligible. In exact security analyses, the probability of a collision is correctly expressed by the system.) `large` is equivalent to `size20`.

`password` is intended for passwords in password-based security protocols. These passwords are taken in a dictionary whose size is much smaller than the size of a nonce for instance, so the probability of collisions among passwords is larger than among data of `large` types. `password` is equivalent to `size10`.

- `fun` ⟨ident⟩`(`seq⟨ident⟩`)`:⟨ident⟩ `[[`seq$^+$⟨option⟩`]]`.

  `fun` f`(`$T_1, \dots, T_n$`)`:T. declares a function that takes n arguments, of types $T_1, \dots, T_n$, and returns a result of type T. Optionally, the declaration of a function may be followed by options between brackets. These options can be:

  - `compos` means that f is injective and that its inverses can be computed in polynomial time: $f(x_1, \dots, x_m) = y$ implies for $i \in \{1, \dots, m\}$, $x_i = f_i^{-1}(y)$ for some functions f_i^{-1}. (In the vocabulary of [1], f is poly-injective.) f can then be used for pattern matching.
  - `decompos` means that f is an inverse of a poly-injective function. f must be unary. (Thanks to the pattern matching construct, one can in general avoid completely the declaration of `decompos` functions, by just declaring the corresponding poly-injective function `compos`.)
  - `uniform` means that f maps the default distribution of its argument into the default distribution of its result. f must be unary; the argument and the result of f must be of types marked `fixed`, `bounded`, or `nonuniform`.

- `letfun` ⟨ident⟩`[(`seq⟨vartype⟩`)]=`⟨term⟩`.`

  `letfun` f`(`x_1`:`$T_1, \dots, x_n$`:`T_n`)=`M. declares a function f that takes n arguments named $x_1, \dots, x_n$ of types $T_1, \dots, T_n$, respectively. The subsequent calls to this function are replaced by the term M in which we replace $x_1, \dots, x_n$ with the arguments given by the caller.

  The term M must not contain `find` constructs.

- `const` seq$^+$⟨ident⟩`:`⟨ident⟩`.`

  `const` $c_1, \dots, c_n$`:`T. declares constants $c_1, \dots, c_n$ of type T. Different constants are assumed to correspond to different bitstrings (except when the instruction `set diffConstants = false.` is given).

- `table` ⟨ident⟩`(`seq$^+$⟨ident⟩`).`

  `table` tbl`(`$T_1, \dots, T_n$`)`. declares the table tbl, whose elements are tuples of type $T_1, \dots, T_n$. Elements can be inserted in the table by `insert` tbl`(`$M_1, \dots, M_n$`)` and the table can be read using `get`.

- `event` ⟨ident⟩`[(`seq⟨ident⟩`)]`.

  `event` e`(`$T_1, \dots, T_n$`)`. declares an event e that takes arguments of types $T_1, \dots, T_n$. When there are no arguments, we can simply declare `event` e.

- `let` ⟨ident⟩ `=` ⟨obody⟩`.`
  `let` ⟨ident⟩ `=` ⟨odef⟩`.`

  `let` x `=` P. says that x represents the process P. When parsing a process, x will be replaced with P.

- `forall` seq⟨vartype⟩`;`⟨simpleterm⟩`.`

  `forall` $x_1 : T_1, \dots, x_n : T_n$`;`$M$`.` says that for all values of $x_1, \dots, x_n$ in types $T_1, \dots, T_n$ respectively, M is true. The term M must be a simple term without array accesses. When M is an equality M_1`=`M_2, CryptoVerif uses this information for rewriting M_1 into M_2, so one must be careful of the orientation of the equality, in particular for termination. When M is an inequality, M_1`<>`M_2, CryptoVerif rewrites M_1`=`M_2 to false and M_1`<>`M_2 to true. Otherwise, it rewrites M to true.

- `equation` ⟨eq_name⟩`(`seq$^+$⟨ident⟩`).`

  This declaration declares the equational theories satisfied by function symbols. The following equational theories are supported:

  – `equation commut(`f`).` indicates that the function f is commutative, that is, $f(x, y) = f(y, x)$ for all x, y. In this case, the function f must be a binary function with both arguments of the same type. (The equation $f(x, y) = f(y, x)$ cannot be given by the `forall` declaration because CryptoVerif interprets such declarations as rewrite rules, and the rewrite rule $f(x, y) \to f(y, x)$ does not terminate.)

  – `equation assoc(`f`).` indicates that the function f is associative, that is, $f(x, f(y, z)) = f(f(x, y), z)$ for all x, y, z. In this case, the function f must be a binary function with both arguments and the result of the same type.

  – `equation AC(`f`).` indicates that the function f is associative and commutative. In this case, the function f must be a binary function with both arguments and the result of the same type.

  – `equation assocU(`f`, `n`).` indicates that the function f is associative, and that n is a neutral element for f, that $f(x, n) = f(n, x) = x$ for all x. In this case, the function f must be a binary function with both arguments and the result of the same type as the type of the constant n.

  – `equation ACU(`f`, `n`).` indicates that the function f is associative and commutative, and that n is a neutral element for f. In this case, the function f must be a binary function with both arguments and the result of the same type as the type of the constant n.

  – `equation ACUN(`f`, `n`).` indicates that the function f is associative and commutative, that n is a neutral element for f, and that f satisfies the cancellation equation $f(x, x) = n$. In this case, the function f must be a binary function with both arguments and the result of the same type as the type of the constant n.

  – `equation group(`f`, `inv`, `n`).` indicates that f forms group with inverse inv and neutral element n, that is, the function f is associative, n is a neutral element for f, and $inv(x)$ is the inverse of x, that is, $f(inv(x), x) = f(x, inv(x)) = n$. In this case, the function f must be a binary function with both arguments and the result of the same type T, inv must be a unary function that takes and returns a value of type T, and n must be a constant of type T.

  – `equation commut_group(`f`, `inv`, `n`).` indicates that f forma commutative group with inverse inv and neutral element n, that is, the function f is associative and commutative, n is a neutral element for f, and $inv(x)$ is the inverse of x. In this case, the function f must be a binary function with both arguments and the result of the same type T, inv must be a unary function that takes and returns a value of type T, and n must be a constant of type T.

- `collision` `(`⟨ident⟩ `<-R` ⟨ident⟩`;)`*`[forall` seq⟨vartype⟩`;]`
  `return(`⟨simpleterm⟩`) <=(`⟨proba⟩`)=> return(`⟨simpleterm⟩`).`

  `collision` x_1 `<-R` T_1`;`$\dots x_n$ `<-R` T_n`;forall` $y_1 : T'_1, \dots, y_m : T'_m$`; return(`$M_1$`) <=(`$p$`)=> return(`$M_2$`).` means that when $x_1, \dots, x_n$ are chosen randomly and independently in $T_1, \dots, T_n$ respectively (with the default probability distributions for these types), a Turing machine running in time `time` has probability at most p of finding $y_1, \dots, y_m$ in $T'_1, \dots, T'_m$ such that $M_1 \neq M_2$. The terms M_1 and M_2 must be simple terms without array accesses. See below for the syntax of probability formulas.

  This allows CryptoVerif to rewrite M_1 into M_2 with probability loss p, when $x_1, \dots, x_n$ are created by independent random number generations of types $T_1, \dots, T_n$ respectively. One should be careful of the orientation of the equivalence, in particular for termination.

- `equiv` ⟨omode⟩ [`|` ... `|`⟨omode⟩] `<=(`⟨proba⟩`)=>` [`[manual]`] ⟨ogroup⟩ [`|` ... `|`⟨ogroup⟩]`.`

  `equiv` L `<=(`p`)=>` R`.` means that the probability that a probabilistic Turing machine that runs in time `time` distinguishes L from R is at most p.

  L and R define sets of oracles. (In these definitions, `foreach` i`<=`N `do` x_1 `<-R` $T_1; \dots x_m$ `<-R` $T_m; Q$ in fact stands for `foreach` i`<=`N `do` O`()` `:=` x_1 `<-R` $T_1; \dots x_m$ `<-R` T_m; `return`; Q, where O is a fresh oracle name. The same oracle names are used in both sides of the equivalence.)

  In the left-hand side, an optional integer between brackets `[`n`]` ($n \geq 0$) can be added in the definition of an oracle, which becomes O`(`x_1 `:` $T_1, \dots x_n$ `:` T_n`)` `[`n`]` `:=` P. This integer does not change the semantics of the oracle, but is used for the proof strategy: CryptoVerif uses preferably the oracles with the smallest integers n when several oracles can be used for representing the same expression. When no integer is mentioned, $n = 0$ is assumed, so the oracle has the highest priority.

  In the left-hand side, the optional indication `[useful_change]` can also be added in the definition of an oracle, which becomes O`(`x_1 `:` $T_1, \dots x_n$ `:` T_n`)` `[useful_change]` `:=` P. This indication is also used for the proof strategy: if at least one `[useful_change]` indication is present, CryptoVerif applies the transformation defined by the equivalence only when at least one `[useful_change]` function is called in the game.

  CryptoVerif uses such equivalences to transform processes that call oracles of L into processes that call oracles of R.

  L may contain mode indications to guide the rewriting: the mode `[all]` means that all occurrences of the root function symbol of oracles in the considered group must be transformed; the mode `[exist]` means that at least one occurrence of an oracle in this group must be transformed. (`[exist]` is the default; there must be at most one oracle group with mode `[exist]`; when an oracle group contains no random number generation, it must be in mode `[all]`.)

  The `[manual]` indication, when it is present in the equivalence, prevents the automatic application of the transformation. The transformation is then applied only using the manual `crypto` command.

  L and R must satisfy certain syntactic constraints:

  - L and R must be well-typed, satisfy the constraints on array accesses (see the description of processes above), and the type of the results of corresponding oracles in L and R must be the same.
  - All oracle definitions in L are of the form O`(...)` `:=` `return(`M`)` where M is a simple term. Oracle definitions in R cannot contain `end`, `event`, and their return instructions must be of the form `return(`M`)`. (They return a single term and they have no further oracle definitions under the return.)
  - L and R must have the same structure: same replications, same number of oracles, same oracle names in the same order, same number of arguments with the same types for each oracle.
  - Under a replication with no random number generation in L, one can have only a single oracle.
  - Replications in L (resp. R) must have pairwise distinct bounds. Oracles in L (resp. R) must have pairwise distinct names.
  - Finds in R are of the form

    `find` ... `orfind` u_1 `<=` $N_1, \dots, u_m$ `<=` N_m `suchthat` `defined(`$z_1[\widetilde{u_1}], \dots, z_l[\widetilde{u_l}]$`)` `&&` M `then` $FP \dots$ `else` FP'

    where $\widetilde{u_k}$ is a non-empty prefix of $u_1, \dots, u_m$, at least one $\widetilde{u_k}$ for $1 \leq k \leq l$ is the whole sequence $u_1, \dots, u_m$, and the implicit prefix of the current array indexes is the same for all $z_1, \dots, z_l$. (When z is defined under replications `!`N_1, ..., `!`N_n, z is always an array with n dimensions, so it expects n indexes, but the first $n' < n$ indexes are left implicit when they are equal to the current indexes of the top-most n' replications above the usage of z—which must also be the top-most n' replications above the definition of z. We require the implicit indexes to be the same for all variables $z_1, \dots, z_l$.) Furthermore, there must exist $k \in \{1, \dots, l_j\}$ such that for all $k' \neq k$, $z_{k'}$ is defined syntactically above all definitions of z_k and $\widetilde{u_{k'}}$ is a prefix of $\widetilde{u_k}$. Finally, variables z_k must not be defined by a `find` in R.

  This is the key declaration for defining the security properties of cryptographic primitives. Since such declarations are delicate to design, we recommend using predefined primitives listed in Section 6, or copy-pasting declarations from examples.

- `query` seq$^+$⟨query⟩ `.`

  The `query` declaration indicates which security properties we would like to prove. The available queries are as follows:

  - `secret1` x: show that any element of the array x cannot be distinguished from a random number (by a single test query). In the vocabulary of [1], this is one-session secrecy.
  - `secret` x: show that the array x is indistinguishable from an array of independent random numbers (by several test queries). In the vocabulary of [1], this is secrecy.
  - $x_1 : T_1, \ldots, x_n : T_n$`;event` M `==>` M'. First, we declare the types of all variables $x_1, \ldots, x_n$ that occur in M or M'. The system shows that, for all values of variables that occur in M, if M is true then there exist values of variables of M' that do not occur in M such that M' is true.

    M must be a conjunction of terms [`inj:`]e or [`inj:`]$e(M_1, \ldots, M_n)$ where e is an event declared by `event` and the M_i are simple terms without array accesses (not containing events).

    M' must be formed by conjunctions and disjunctions of terms [`inj:`]e, [`inj:`]$e(M_1, \ldots, M_n)$, or simple terms without array accesses (not containing events).

    When `inj:` is present, the system proves an injective correspondence, that is, it shows that several different events marked `inj:` before `==>` imply the execution of several different events marked `inj:` after `==>`. More precisely, `inj:`$e_1(M_{11}, \ldots, M_{1m_1})$ `&& ... && inj:`$e_n(M_{n1}, \ldots, M_{nm_n})$ `&& ... ==>` M' means that for each tuple of executed events $e_1(M_{11}, \ldots, M_{1m_1})$ (executed N_1 times), …, $e_n(M_{n1}, \ldots, M_{nm_n})$ (executed N_n times), M' holds, considering that an event `inj:`$e'(M_1, \ldots, M_m)$ in M' holds when it has been executed at least $N_1 \times \ldots \times N_n$ times. When e is preceded by `inj:` in a query, e must occur at most once in each branch of `if`, `find`, `let`, and all occurrences of the same e must be under replications of the same types. The `inj:` marker must occur either both before and after `==>` or not at all. (Otherwise, the query would be equivalent to a non-injective correspondence.)

- `proof {`⟨command⟩`;`…`;`⟨command⟩`}`

  Allows the user to include in the CryptoVerif input file the commands that must be executed by CryptoVerif in order to prove the protocol. The allowed commands are those described in Section 7, except that `help` and `?` are not allowed and that the `crypto` command must be fully specified (so that no user interaction is required). If the command contains a string that is not a valid identifier, `*`, or `.`, then this string must be put between quotes `"`. This is useful in particular for variable names introduced internally by CryptoVerif and that contain `@` (so that they cannot be confused with variables introduced by the user), for example `"@2_r1"`.

- `define` ⟨ident⟩`(`seq⟨ident⟩`) {`seq⟨decl⟩`}`

  `define` $m(x_1, \ldots, x_n)$ $\{d_1, \ldots, d_k\}$ defines a macro named m, with arguments $x_1, \ldots, x_n$. This macro expands to the declarations $d_1, \ldots, d_k$, which can be any of the declarations listed in this manual, except `define` itself. The macro is expanded by the `expand` declaration described below. When the `expand` declaration appears inside a `define` declaration, the expanded macro must have been defined before the `define` declaration (which prevents recursive macros, whose expansion would loop). Macros are used in particular to define a library of standard cryptographic primitives that can be reused by the user without entering their full definition. These primitives are presented in Section 6.

- `expand` ⟨ident⟩`(`seq⟨ident⟩`).`

`expand` $m(y_1, \dots, y_n)$`.` expands the macro m by applying it to the arguments $y_1, \dots, y_n$. If the definition of the macro m is `define` $m(x_1, \dots, x_n)$ $\{d_1, \dots, d_k\}$, then it generates $d_1, \dots, d_k$ in which $y_1, \dots, y_n$ are substituted for $x_1, \dots, x_n$ and the other identifiers that were not already defined at the `define` declaration are renamed to fresh identifiers.

The following identifiers are predefined:

- The type `bitstring` is the type of all bitstrings. It is large.
- The type `bitstringbot` is the type that contains all bitstrings and $\perp$. It is also large.
- The type `bool` is the type of boolean values, which consists of two constant bitstrings `true` and `false`. It is declared `fixed`.
- The function `not` is the boolean negation, from `bool` to `bool`.
- The constant `bottom` represents $\perp$. (The special element of `bitstringbot` that is not a bitstring.)

The syntax of probability formulas allows parenthesing and the usual algebraic operations `+`, `-`, `*`, `/`. (`*` and `/` have higher priority than `+` and `-`, as usual.), as well as the maximum, denoted `max(`p_1`,` $\dots$ `,`p_n`)`. They may also contain

- P or $P(p_1, \dots, p_n)$ where P has been declared by `proba` P and $p_1, \dots, p_n$ are probability formulas; this formula represents an unspecified probability depending on $p_1, \dots, p_n$.
- N, where N has been declared by `param` N, designates the number of copies of a replication.
- $\#O$, where O is an oracle, designates the number of different calls to the oracle O.
- $|T|$, where T has been declared by `type` T and is `fixed` or `bounded`, designates the cardinal of T.
- `maxlength(`M`)` is the maximum length of term M (M must be a simple term without array access, and must be of a non-bounded type).
- `length(`$f, p_1, \dots, p_n$`)` designates the maximal length of the result of a call to f, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of f (p_i must be built from `max`, `maxlength(`M`)`, and `length(`$f', \dots$`)`, where M is a term of the type of the corresponding argument of f and the result of f' is of the type of the corresponding argument of f).
- `length`(T) designates the maximal length of a bitstring of type T, where T is a bounded type.
- `length((`$T_1, \dots, T_n$`)`$, p_1, \dots, p_n$`)` designates the maximal length of the result of the tuple function from $T_1 \times \dots \times T_m$ to `bitstring`, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of this function.
- n is an integer constant.
- `eps_find` is the maximum distance between the uniform probability distribution and the probability distribution used for choosing elements in `find`.
- `eps_rand(`T`)` is the maximum distance between the uniform probability distribution and the default probability distribution D_T for type T (when T is `bounded`).
- `Pcoll1rand(`T`)` is the maximum probability of collision between a random value X of type T chosen according to the default distribution D_T for type T and an element of type T that does not depend on it (when T is `nonuniform`). This is also the maximum probability of choosing any given element of T in the default distribution for that type:

  $$\texttt{Pcoll1rand(}T\texttt{)} = \max_{a \in T} \Pr[X = a]$$

  where X is chosen according to distribution D_T.

- `Pcoll2rand(`T`)` is the maximum probability of collision between two independent random values of type T chosen according to the default distribution D_T for type T (when T is `nonuniform`). We have

$$\texttt{Pcoll2rand(}T\texttt{)} = \sum_{a \in T} \Pr[X = a]^2 \leq \texttt{Pcoll1rand(}T\texttt{)}$$

  where X is chosen according to the default distribution D_T.

- `time` designates the runtime of the environment (attacker).

Finally, `time(...)` designates the runtime time of each elementary action of a game:

- `time(`$f, p_1, \dots, p_n$`)` designates the maximal runtime of one call to function symbol f, where $p_1, \dots,$ p_n represent the maximum length of the non-bounded arguments of f.
- `time(let` $f, p_1, \dots, p_n$`)` designates the maximal runtime of one pattern matching operation with function symbol f, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of f.
- `time((`$T_1, \dots, T_m$`),`$p_1, \dots, p_n$`)` designates the maximal runtime of one call to the tuple function from $T_1 \times \dots \times T_m$ to `bitstring`, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of this function.
- `time(let(`$T_1, \dots, T_m$`),`$p_1, \dots, p_n$`)` designates the maximal runtime of one pattern matching with the tuple function from $T_1 \times \dots \times T_m$ to `bitstring`, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of this function.
- `time(=`$T[, p_1, p_2]$`)` designates the maximal runtime of one call to bitstring comparison function for bitstrings of type T, where p_1, p_2 represent the maximum length of the arguments of this function when T is non-bounded.
- `time(foreach)` is the maximum time of an access to an index i of an instruction `foreach` i`<=`N.
- `time([`n`])` is the maximum time of an array access with n indexes.
- `time(&&)` is the maximum time of a boolean and.
- `time(||)` is the maximum time of a boolean or.
- `time(<-R` T`)` is the maximum time needed to choose a random number of type T according to the default distribution for type T.
- `time(newOracle)` is the maximum time to create a new private oracle.
- `time(if)` is the maximum time to perform a boolean test.
- `time(find` n`)` is the maximum time to perform one condition test of a find with n indexes to choose. (Essentially, the time to store the values of the indexes in a list and part of the time needed to randomly choose an element of that list.)

CryptoVerif checks the dimension of probability formulas.

5 Summary of the Main Differences between the two Front-ends

The main difference between the two front-ends is that the `oracles` front-end uses oracles while the `channels` front-end uses channels. So we have essentially the following correspondence:

`channels`	`oracles`
input process	oracle definition
output process	oracle body
`newChannel` c	`newOracle` O
`in(`c`, (`$x_1 : T_1, \dots, x_l : T_l$`));`$P$	O`(`$x_1 : T_1, \dots, x_l : T_l$`) :=` P
`out(`c`, (`$M_1, \dots, M_l$`));`Q	`return(`$M_1, \dots, M_l$`);`Q
`yield`	`end`

The `newChannel` or `newOracle` instruction does not appear in processes, but appears in the evaluation time of contexts. In the `channels` front-end, channels must be declared by a `channel` declaration. There is no such declaration in the `oracles` front-end.

In equivalences that define security assumptions, functions of the `channels` front-end are also replaced with oracle definitions in the `oracles` front-end:

`channels`	`oracles`
function	oracle definition
($x_1 : T_1, \ldots, x_l : T_l$) `->` M	O($x_1 : T_1, \ldots, x_l : T_l$) `:= return(`$M$`)`
($x_1 : T_1, \ldots, x_l : T_l$) N `->` M	`foreach` i`<=`N `do` O($x_1 : T_1, \ldots, x_l : T_l$) `:= return(`$M$`)`

Finally, some constructs use a different syntax in the `oracles` front-end, to be closer to the syntax of cryptographic games:

`channels`	`oracles`
`!`i`<=`N Q	`foreach` i`<=`N `do` Q
`new` x`:`T`;` P	x `<-R` T`;` P
`let` x`:`T `=` M `in` P	x`:`T `<-` M`;` P

The `let` instruction is still available in the `oracles` front-end. Indeed, the assignment x`:`T `<-` M can be used only for directly assigning a variable; when a pattern occurs instead of the variable x, one has to use the `let` instruction.

6 Predefined cryptographic primitives

A number of standard cryptographic primitives are predefined in CryptoVerif. The definitions of these primitives are given as macros in the library file `default.cvl` (or `default.ocvl` for the `oracles` front-end) that is automatically loaded at startup. The user does not need to redefine these primitives, he can just expand the corresponding macro. The examples contained in the library can be used as a basis in order to build definitions of new primitives, by copying and modifying them as desired. Here is a list of the predefined primitives.

- `expand IND_CPA_sym_enc(`*keyseed*, *key*, *cleartext*, *ciphertext*, *seed*, *kgen*, *enc*, *dec*, *injbot*, Z, *Penc*`).` defines a IND-CPA (indistinguishable under chosen plaintext attacks) probabilistic symmetric encryption scheme.

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  key is the type of keys, must be `bounded`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  seed is the type of random seeds for encryption, must be `bounded`, typically `fixed`.

  kgen(*keyseed*) : *key* is the key generation function.

  enc(*cleartext*, *key*, *seed*) : *ciphertext* is the encryption function.

  dec(*ciphertext*, *key*) : `bitstringbot` is the decryption function; it returns `bottom` when decryption fails.

  injbot(*cleartext*) : `bitstringbot` is the natural injection from *cleartext* to `bitstringbot`.

  Z(*cleartext*) : *cleartext* is the function that returns for each cleartext a cleartext of the same length consisting only of zeroes.

  Penc(t, N, l) is the probability of breaking the IND-CPA property in time t for one key and N encryption queries with cleartexts of length at most l.

  The types *keyseed*, *key*, *cleartext*, *ciphertext*, *seed* and the probability *Penc* must be declared before this macro is expanded. The functions *kgen*, *enc*, *dec*, *injbot*, and Z are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

This macro defines the equivalence named `ind_cpa`(*enc*) for use in the `crypto` command in interactive proofs (see Section 7).

- `expand IND_CPA_INT_CTXT_sym_enc(`*keyseed*, *key*, *cleartext*, *ciphertext*, *seed*, *kgen*, *enc*, *dec*, *injbot*, *Z*, *Penc*, *Pencctxt*`).` defines a IND-CPA (indistinguishable under chosen plaintext attacks) and INT-CTXT (ciphertext integrity) probabilistic symmetric encryption scheme.

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  key is the type of keys, must be `bounded`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  seed is the type of random seeds for encryption, must be `bounded`, typically `fixed`.

  $kgen(keyseed)$: *key* is the key generation function.

  $enc(cleartext, key, seed)$: *ciphertext* is the encryption function.

  $dec(ciphertext, key)$: `bitstringbot` is the decryption function; it returns `bottom` when decryption fails.

  $injbot(cleartext)$: `bitstringbot` is the natural injection from *cleartext* to `bitstringbot`.

  $Z(cleartext)$: *cleartext* is the function that returns for each cleartext a cleartext of the same length consisting only of zeroes.

  $Penc(t, N, l)$ is the probability of breaking the IND-CPA property in time t for one key and N encryption queries with cleartexts of length at most l.

  $Pencctxt(t, N, N', l, l')$ is the probability of breaking the INT-CTXT property in time t for one key, N encryption queries, N' decryption queries with cleartexts of length at most l and ciphertexts of length at most l'.

  The types *keyseed*, *key*, *cleartext*, *ciphertext*, *seed* and the probabilities *Penc* and *Pencctxt* must be declared before this macro is expanded. The functions *kgen*, *enc*, *dec*, *injbot*, and *Z* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalences named `ind_cpa`(*enc*) and `int_ctxt`(*enc*) corresponding respectively to the IND-CPA and INT-CTXT properties, for use in the `crypto` command (see Section 7).

- `expand IND_CCA2_INT_PTXT_sym_enc(`*keyseed*, *key*, *cleartext*, *ciphertext*, *seed*, *kgen*, *enc*, *dec*, *injbot*, *Z*, *Penc*, *Pencptxt*`).` defines a IND-CCA2 (indistinguishable under adaptive chosen ciphertext attacks) and INT-PTXT (plaintext integrity) probabilistic symmetric encryption scheme.

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  key is the type of keys, must be `bounded`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  seed is the type of random seeds for encryption, must be `bounded`, typically `fixed`.

  $kgen(keyseed)$: *key* is the key generation function.

  $enc(cleartext, key, seed)$: *ciphertext* is the encryption function.

  $dec(ciphertext, key)$: `bitstringbot` is the decryption function; it returns `bottom` when decryption fails.

  $injbot(cleartext)$: `bitstringbot` is the natural injection from *cleartext* to `bitstringbot`.

  $Z(cleartext)$: *cleartext* is the function that returns for each cleartext a cleartext of the same length consisting only of zeroes.

Penc(t, N, N', l, l') is the probability of breaking the IND-CCA2 property in time t for one key, N encryption queries, N' decryption queries with cleartexts of length at most l and ciphertexts of length at most l'.

Pencptxt(t, N, N', l, l') is the probability of breaking the INT-PTXT property in time t for one key, N encryption queries, N' decryption queries with cleartexts of length at most l and ciphertexts of length at most l'.

The types *keyseed*, *key*, *cleartext*, *ciphertext*, *seed* and the probabilities *Penc* and *Pencptxt* must be declared before this macro is expanded. The functions *kgen*, *enc*, *dec*, *injbot*, and *Z* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

This macro defines the equivalences named `ind_cca2`(*enc*) and `int_ptxt`(*enc*) corresponding respectively to the IND-CCA2 and INT-PTXT properties, for use in the `crypto` command (see Section 7).

- `expand SPRP_cipher(`*keyseed*, *key*, *blocksize*, *kgen*, *enc*, *dec*, *Penc*`).` defines a SPRP (super-pseudo-random permutation) deterministic symmetric encryption scheme.

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  key is the type of keys, must be `bounded`.

  blocksize is the type of cleartexts and ciphertexts, must be `fixed` and `large`. (The modeling of SPRP block ciphers is not perfect in that, in order to encrypt a new message, one chooses a fresh random number, not necessarily different from previously generated random numbers. Then CryptoVerif needs to eliminate collisions between those random numbers, so *blocksize* must really be `large`.)

  kgen(*keyseed*) : *key* is the key generation function.

  enc(*blocksize*, *key*) : *blocksize* is the encryption function.

  dec(*blocksize*, *key*) : *blocksize* is the decryption function.

  Penc(t, N, N') is the probability of breaking the SPRP property in time t for one key, N encryption queries, and N' decryption queries.

  The types *keyseed*, *key*, *blocksize* and the probability *Penc* must be declared before this macro is expanded. The functions *kgen*, *enc*, and *dec* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalence named `sprp`(*enc*) for use in the `crypto` command (see Section 7).

- `expand PRP_cipher(`*keyseed*, *key*, *blocksize*, *kgen*, *enc*, *dec*, *Penc*`).` defines a PRP (pseudo-random permutation) deterministic symmetric encryption scheme.

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  key is the type of keys, must be `bounded`.

  blocksize is the type of cleartexts and ciphertexts, must be `fixed` and `large`. (The modeling of PRP block ciphers is not perfect in that, in order to encrypt a new message, one chooses a fresh random number, not necessarily different from previously generated random numbers. In other words, we model a PRF rather than a PRP, and apply the PRF/PRP switching lemma to make sure that this is sound. Then CryptoVerif needs to eliminate collisions between those random numbers, so *blocksize* must really be `large`.)

  kgen(*keyseed*) : *key* is the key generation function.

  enc(*blocksize*, *key*) : *blocksize* is the encryption function.

  dec(*blocksize*, *key*) : *blocksize* is the decryption function.

  Penc(t, N) is the probability of breaking the PRP property in time t for one key and N encryption queries.

The types *keyseed*, *key*, *blocksize* and the probability *Penc* must be declared before this macro is expanded. The functions *kgen*, *enc*, and *dec* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

This macro defines the equivalence named `prp(`*enc*`)` for use in the `crypto` command (see Section 7).

- `expand ICM_cipher(`*cipherkey*, *key*, *blocksize*, *enc*, *dec*`).` defines a block cipher in the ideal cipher model.

  cipherkey is the type of keys that correspond to the choice of the scheme, must be `bounded`, typically `fixed`.

  key is the type of keys (typically `large`).

  blocksize is type of the input and output of the cipher, must be `fixed` and `large`. (The modeling of the ideal cipher model is not perfect in that, in order to encrypt a new message, one chooses a fresh random number, not necessarily different from previously generated random numbers. Then CryptoVerif needs to eliminate collisions between those random numbers, so blocksize must really be `large`.)

  $enc(blocksize, key) : blocksize$ is the encryption function.

  $dec(blocksize, key) : blocksize$ is the decryption function.

  WARNING: the encryption and decryption functions take 2 keys as input: the key of type cipherkey that corresponds to the choice of the scheme, and the normal encryption/decryption key. The cipherkey must be chosen once and for all at the beginning of the game and the encryption and decryption oracles must be made available to the adversary, by including a process such as

```
   (! qE in(c1, (x:blocksize, ke:key)); out(c2, enc(ck,x,ke)))
 | (! qD in(c3, (m:blocksize, kd:key)); out(c4, dec(ck,m,kd)))
```

  where `c1`, `c2`, `c3`, `c4` are channels, `qE` the number of requests to the encryption oracle, `qD` the number of requests to the decryption oracle, `ck` the cipherkey (or similar oracles if you use the oracles front-end).

  The types *cipherkey*, *key*, *blocksize* must be declared before this macro is expanded. The functions *enc*, *dec* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalence named `icm(`*enc*`)` for use in the `crypto` command (see Section 7).

- `expand UF_CMA_mac(`*mkeyseed*, *mkey*, *macinput*, *macres*, *mkgen*, *mac*, *check*, *Pmac*`).` defines a UF-CMA (unforgeable under chosen message attacks) MAC (message authentication code).

  mkeyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  mkey is the type of keys, must be `bounded`.

  macinput is the type of inputs of MACs

  macres is the type of MACs.

  $mkgen(mkeyseed) : mkey$ is the key generation function.

  $mac(macinput, mkey) : macres$ is the MAC function.

  $check(macinput, mkey, macres)$: `bool` is the verification function.

  $Pmac(t, N, N', l)$ is the probability of breaking the UF-CMA property in time t for one key, N MAC queries, N' verification queries for messages of length at most l.

  The types *mkeyseed*, *mkey*, *macinput*, *macres* and the probability *Pmac* must be declared before this macro is expanded. The functions *mkgen*, *mac*, *check* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalences named `uf_cma(`*mac*`)` and `uf_cma_corrupt(`*mac*`)`, for use in the `crypto` command (see Section 7). Both equivalences correspond to the UF-CMA property, but the former does not allow corruption of the secret keys while latter allows it. The latter equivalence is applied only manually, because its automatic application can sometimes be done too early, when other transformations should first be done in order to eliminate uses of the secret keys.

- `expand SUF_CMA_mac(`*mkeyseed*, *mkey*, *macinput*, *macres*, *mkgen*, *mac*, *check*, *Pmac*`).` defines a SUF-CMA (strongly unforgeable under chosen message attacks) MAC (message authentication code). The difference between a UF-CMA MAC and a SUF-CMA MAC is that, for a UF-CMA MAC, the adversary may easily forge a new MAC for a message for which he has already seen a MAC. Such a forgery is guaranteed to be hard for a SUF-CMA MAC. The arguments are the same as for the previous macro. This macro defines the equivalences named `suf_cma`(*mac*) and `suf_cma_corrupt`(*mac*), for use in the `crypto` command (see Section 7).

- `expand IND_CCA2_public_key_enc(`*keyseed*, *pkey*, *skey*, *cleartext*, *ciphertext*, *seed*, *skgen*, *pkgen*, *enc*, *dec*, *injbot*, *Z*, *Penc*, *Penccoll*`).` defines a IND-CCA2 (indistinguishable under adaptive chosen ciphertext attacks) probabilistic public-key encryption scheme.

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  pkey is the type of public keys, must be `bounded`.

  skey is the type of secret keys, must be `bounded`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  seed is the type of random seeds for encryption, must be `bounded`, typically `fixed`.

  skgen(*keyseed*) : *skey* is the secret key generation function.

  pkgen(*keyseed*) : *pkey* is the public key generation function.

  enc(*cleartext*, *pkey*, *seed*) : *ciphertext* is the encryption function.

  dec(*ciphertext*, *skey*) : `bitstringbot` is the decryption function; it returns `bottom` when decryption fails.

  injbot(*cleartext*) : `bitstringbot` is the natural injection from *cleartext* to `bitstringbot`.

  Z : *cleartext* is a constant cleartext. The encryption scheme is assumed to encrypt a block, so that the length of the cleartext is not leaked.

  Penc(t, N) is the probability of breaking the IND-CCA2 property in time t for one key and N decryption queries.

  Penccoll is the probability of collision between independently generated keys.

  The types *keyseed*, *pkey*, *skey*, *cleartext*, *ciphertext*, *seed* and the probabilities *Penc*, *Penccoll* must be declared before this macro is expanded. The functions *skgen*, *pkgen*, *enc*, *dec*, *injbot*, and *Z* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalence named `ind_cca2`(*enc*) for use in the `crypto` command (see Section 7).

- `expand UF_CMA_signature(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *seed*, *skgen*, *pkgen*, *sign*, *check*, *Psign*, *Psigncoll*`).` defines a UF-CMA (unforgeable under chosen message attacks) probabilistic signature scheme.

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  pkey is the type of public keys, must be `bounded`.

  skey is the type of secret keys, must be `bounded`.

  signinput is the type of signature inputs.

  signature is the type of signatures.

seed is the type of random seeds for signatures, must be `bounded`, typically `fixed`.

skgen(*keyseed*) : *skey* is the secret key generation function.

pkgen(*keyseed*) : *pkey* is the public key generation function.

sign(*signinput*, *skey*, *seed*) : *signature* is the signature function.

check(*signinput*, *pkey*, *signature*) : `bool` is the verification function.

Psign(t, N, l) is the probability of breaking the UF-CMA property in time t, for one key, N signature queries with messages of length at most l.

Psigncoll is the probability of collision between independently generated keys.

The types *keyseed*, *pkey*, *skey*, *signinput*, *signature*, *seed* and the probabilities *Psign*, *Psigncoll* must be declared before this macro is expanded. The functions *skgen*, *pkgen*, *sign*, and *check* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

This macro defines the equivalences named `uf_cma`(*sign*) and `uf_cma_corrupt`(*sign*), for use in the `crypto` command (see Section 7). Both equivalences correspond to the UF-CMA property, but the former does not allow corruption of the secret keys while latter allows it. The latter equivalence is applied only manually, because its automatic application can sometimes be done too early, when other transformations should first be done in order to eliminate uses of the secret keys.

- `expand SUF_CMA_signature(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *seed*, *skgen*, *pkgen*, *sign*, *check*, *Psign*, *Psigncoll*`).` defines a SUF-CMA (strongly unforgeable under chosen message attacks) probabilistic signature scheme. The difference between a UF-CMA signature and a SUF-CMA MAsignature is that, for a UF-CMA signature, the adversary may easily forge a new signature for a message for which he has already seen a signature. Such a forgery is guaranteed to be hard for a SUF-CMA signature. The arguments are the same as for the previous macro. This macro defines the equivalences named `suf_cma`(*sign*) and `suf_cma_corrupt`(*sign*), for use in the `crypto` command (see Section 7).

- `expand ROM_hash(`*key*, *hashinput*, *hashoutput*, *hash*`).` defines a hash function in the random oracle model.

  key is the type of the key of the hash function, which models the choice of the hash function, must be `bounded`, typically `fixed`.

  hashinput is the type of the input of the hash function.

  hashoutput is the type of the output of the hash function, must be `bounded` and `large`, typically `fixed`.

  hash(*hashinput*) : *hashoutput* is the hash function.

  WARNING: *hash* is a keyed hash function. The key must be generated once and for all at the beginning of the game and the hash oracle must be made available to the adversary, by including a process such as `!qH in(c1, x:hashinput); out(c2, hash(k,x))` where `k` is the key, `qH` the number of requests to the hash oracle, `c1` and `c2` channels (or a similar oracle if you use the oracles front-end).

  The types *key*, *hashinput*, and *hashoutput* must be declared before this macro. The function *hash* is defined by this macro. It must not be declared elsewhere, and it can be used only after expanding the macro.

  This macro defines the equivalence named `rom`(*hash*) for use in the `crypto` command (see Section 7).

- `expand CollisionResistant_hash(`*key*, *hashinput*, *hashoutput*, *hash*, *Phash*`).` defines a collision-resistant hash function.

  key is the type of the key of the hash function, must be `bounded`, typically `fixed`.

  hashinput is the type of the input of the hash function.

  hashoutput is the type of the output of the hash function.

hash(*key*, *hashinput*) : *hashoutput* is the hash function.

Phash is the probability of breaking collision resistance. WARNING: A collision resistant hash function is a keyed hash function. The key must be generated once and for all at the beginning of the game, and immediately made available to the adversary.

The types *key*, *hashinput*, and *hashoutput* and the probability *Phash* must be declared before this macro. The function *hash* is defined by this macro. It must not be declared elsewhere, and it can be used only after expanding the macro.

- `expand OW_trapdoor_perm(`*seed*, *pkey*, *skey*, D, *pkgen*, *skgen*, f, *invf*, POW`).` defines a one-way trapdoor permutation.

  seed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  pkey is the type of public keys, must be `bounded`.

  skey is the type of secret keys, must be `bounded`.

  D is the type of the input and output of the permutation, must be `bounded`, typically `fixed`.

  pkgen(*seed*) : *pkey* is the public key generation function.

  skgen(*seed*) : *skey* is the secret key generation function.

  f(*pkey*, D) : D is the permutation (taking as argument the public key)

  invf(*skey*, D) : D is the inverse permutation of f (taking as argument the secret key, i.e. the trapdoor)

  $POW(t)$ is the probability of breaking the one-wayness property in time t, for one key and one permuted value.

  The types *seed*, *pkey*, *skey*, D, and the probability POW must be declared before this macro. The functions *pkgen*, *skgen*, f, *invf* are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalences `remove_invf`(f), which expresses that, for y chosen randomly in D, y and *invf*(*skey*, y) are distributed like for x chosen randomly in D, f(*pkey*, x) and x, and `ow`(f), which corresponds to one-wayness, for use in the `crypto` command (see Section 7).

- `expand OW_trapdoor_perm_RSR(`*seed*, *pkey*, *skey*, D, *pkgen*, *skgen*, f, *invf*, POW`).` defines a one-way trapdoor permutation, with random self-reducibility. The arguments are the same as above, but the probability of breaking one-wayness is bound more precisely. This macro defines the equivalences `remove_invf`(f) as above and `ow_rsr`(f).

- `expand set_PD_OW_trapdoor_perm(`*seed*, *pkey*, *skey*, D, *Dow*, *Dr*, *pkgen*, *skgen*, f, *invf*, *concat*, P_PD_OW`).` defines a set partial-domain one-way trapdoor permutation.

  seed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  pkey is the type of public keys, must be `bounded`.

  skey is the type of secret keys, must be `bounded`.

  D is the type of the input and output of the permutation, must be `bounded`, typically `fixed`. The domain D consists of the concatenation of bitstrings in *Dow* and *Dr*. *Dow* is the set of sub-bitstrings of D on which one-wayness holds (it is difficult to compute the random element x of *Dow* knowing f(*pk*, *concat*(x, y)) where y is a random element of *Dr*). *Dow* and *Dr* must be `bounded`, typically `fixed`.

  pkgen(*seed*) : *pkey* is the public key generation function.

  skgen(*seed*) : *skey* is the secret key generation function.

  f(*pkey*, D) : D is the permutation (taking as argument the public key)

  invf(*skey*, D) : D is the inverse permutation of f (taking as argument the secret key, i.e. the trapdoor)

$concat(Dow, Dr) : D$ is bitstring concatenation.

$P\_PD\_OW(t, l)$ is the probability of breaking the set partial-domain one-wayness property in time t, for one key, one permuted value, and l tries.

The types *seed*, *pkey*, *skey*, D, *Dow*, *Dr* and the probability P_PD_OW must be declared before this macro. The functions *pkgen*, *skgen*, f, *invf*, *concat* are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

This macro defines the equivalences `remove_invf`(f), which expresses that, for y chosen randomly in D, y and $invf(skey, y)$ are distributed like for x chosen randomly in D, $f(pkey, x)$ and x, and `pd_ow`(f), which corresponds to set partial-domain one-wayness, for use in the `crypto` command (see Section 7).

- `expand PRF(`*keyseed*, *key*, *input*, *output*, *kgen*, *f*, *Pprf*`).` defines a pseudo-random function.

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it), typically `fixed` and `large`.

  key is the type of keys, must be `bounded`.

  input is the type of the input of the PRF.

  output is the type of the output of the PRF, must be `bounded`, typically `fixed`.

  $kgen(keyseed) : key$ is the key generation function.

  $f(key, input) : output$ is the PRF function.

  $Pprf(t, N, l)$ is the probability of breaking the PRF property in time t, for one key, N queries to the PRF of length at most l.

  The types *keyseed*, *key*, *input*, *output* and the probability *Pprf* must be declared before this macro is expanded. The functions *kgen* and f are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalence named `prf`(f) for use in the `crypto` command (see Section 7).

- `expand CDH(`G, Z, g, *exp*, *mult*, *PCDH*`).` defines a group that satisfies the computational Diffie-Hellman assumption.

  G: type of group elements (must be `bounded` and `large`, of cardinal a prime q).

  Z: type of exponents (must be `bounded` and `large`, supposed to be $\{1, \dots, q-1\}$).

  g: a generator of the group.

  exp: the exponentiation function.

  mult: the multiplication function for exponents, product modulo q in $\{1, \dots, q-1\}$, i.e. in the group $(\mathbb{Z}/q\mathbb{Z})^*$.

  PCDH: the probability of breaking the CDH assumption for one pair of exponents.

  The types G and Z and the probability *PCDH* must be declared before this macro. The functions g, *exp*, and *mult* are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the following equivalences for use in the `crypto` command (see Section 7):

  - `cdh`(*exp*) whichs corresponds to the CDH property
  - `group_to_exp_strict`(*exp*) which allows to replace a random $X \in G$ with $exp(g, x)$ for a random $x \in Z$, provided $exp(X, \_)$ occurs in the game.
  - `group_to_exp`(*exp*) which allows to replace a random $X \in G$ with $exp(g, x)$ for a random $x \in Z$ in any case. (This transformation is applied only manually.)
  - `exp_to_group`(*exp*) which allows to replace $exp(g, x)$ for a random $x \in Z$ with a random $X \in G$.
  - `exp'_to_group`(*exp*) which allows to replace $exp'(g, x)$ for a random $x \in Z$ with a random $X \in G$. (exp' is a symbol equal to *exp* introduced by the `cdh`(*exp*) equivalence.)

- `expand DDH(`$G, Z, g, exp, mult, PDDH$`).` defines a group that satisfies the decisional Diffie-Hellman assumption. The arguments are the same as for `CDH` above, except that the probability $PCDH$ is replaced with $PDDH$. This macro defines the same equivalences as the `CDH` macro, except that `cdh`(exp) is replaced with `ddh`(exp).
- `expand Xor(`$D, xor, zero$`).` defines the function symbol xor to be exclusive or on the set of bitstrings D, where $zero$ is the bitstring consisting only of zeroes in D.

  The type D must be declared before this macro is expanded. The function xor and the constant $zero$ are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalence named `remove_xor`(xor) for use in the `crypto` command (see Section 7).

7 Interactive Mode

In interactive mode, the user specifies transformations to perform. Here is a list of available instructions:

- `help` or `?`: display a list of available commands.
- `remove_assign useless`: remove useless assignments, that is, assignments to x when x is unused and assignments between variables.
- `remove_assign all`: remove all assignments, by replacing variables with their values. This is not recommended: you should try to specify which assignments to remove more precisely.
- `remove_assign binder` x: remove assignments to x by replacing x with its value. When x becomes unused, its definition is removed. When x is used only in `defined` tests after transformation, its definition is replaced with a constant.
- `move` m: Try to move instructions as follows:
  - Move random number generations down in the syntax tree as much as possible, in order to delay the choice of random numbers. This is especially useful when the random number generations can be moved under a test `if`, `let`, or `find`, so that we can distinguish in which branch of the test the random number is created by a subsequent `SArename` instruction.
  - Move assignments down in the syntax tree but without duplicating them. This is especially useful when the assignment can be moved under a test, in which the assigned variable is used only in one branch. In this case, the assigned term is computed in fewer cases thanks to this transformation. (Note that only assignments without array accesses can be moved, because in the presence of array accesses, the computation would have to be kept in all branches of the test, yielding a duplication that we want to avoid.)

  The argument m specifies which instructions should be moved:

  - `all`: move random number generations and assignments, when this is beneficial, that is, when they can be moved under a test.
  - `noarrayref`: move random number generations and assignments without array accesses, when this is beneficial.
  - `random`: move random number generations, when this is beneficial.
  - `random_noarrayref`: move random number generations without array accesses, when this is beneficial.
  - `assign`: move assignments, when this is beneficial.
  - `binder` x: move random number generations and assignments that define x (even when this is not beneficial).

  - `array` x: move random number generations that define x when x is of a `large` and `bounded` type and x is not used in the process that defines it, until the next output after the definition of x. x is then chosen at the point where it is really first used. (Since this point may depend on the trace, the uses of x are often transformed into `find` instructions that test whether x has been chosen before, and reuse the previously chosen value if this is true.)

- `simplify`: simplify the game.

- `simplify coll_elim` $x_1 \dots x_n$: simplify the game, additionally allowing elimination of collisions on data of types with option `password` at the program points defined by $x_1, \dots, x_n$. The arguments $x_1, \dots, x_n$ can be occurrence numbers in the program, variable names (meaning all occurrences of these variables), or types (meaning all data of those types). If one wants to specify occurrence numbers, one should use the command `show_game occ` to determine which occurrence numbers correspond to the desired program points.

- `global_dep_anal` x performs global dependency analysis on x: it computes all variables that depend on x, and when possible, shows that all output messages are independent of x and that all tests are independent of x after eliminating collisions. The tests are then simplified by eliminating these collisions, so that all dependencies on x can be removed.

  `global_dep_anal` x `coll_elim` $x_1 \dots x_n$ performs global dependency analysis on x, additionally allowing elimination of collisions on data of types with option `password` at the program points defined by $x_1, \dots, x_n$. The arguments $x_1, \dots, x_n$ can be occurrence numbers in the program, variable names (meaning all occurrences of these variables), or types (meaning all data of those types). One must allow elimination on x independently of the occurrences, so if x is not large, x or its type must be mentioned in $x_1, \dots, x_n$; mentioning the occurrences of x is not sufficient.

- `SArename` x: When x is defined at several places, rename x to a different name for each definition. This is useful for distinguishing cases depending on which definition of x is used.

- `all_simplify`: perform several simplifications on the game, as if `simplify`, `move all` if `autoMove = true`, and `remove_assign useless` had been called.

- `crypto` $\dots$: applies a cryptographic transformation that comes from a statement `equiv`. This command can have several forms:

  - `crypto`: list all available `equiv` statements, and ask the user to choose which one should be applied, with which variables of the game corresponding to random number generations of the left-hand side of the equivalence.
  - `crypto` ⟨name⟩ `*`: apply a cryptographic transformation determined by the name ⟨name⟩. This name can be either an identifier *id* or *id*(f), and corresponds to the name given at the declaration of the cryptographic transformation by `equiv`. In case the name is not found, CryptoVerif reverts to the old way of designating cryptographic transformations, in which ⟨name⟩ can be either a function symbol that occurs in the terms in the left-hand side of the `equiv` statement, or a probability function that occurs in the probability formula of the `equiv` statement. When several equivalences correspond, the user is prompted for choice. The transformation is applied as many times as possible. (In this case, the advised transformations are applied automatically even when `set autoAdvice = false`.)
  - `crypto` ⟨name⟩ x_1 ... x_n: apply a cryptographic transformation chosen as above, where $x_1, \dots, x_n$ are variable names of the game corresponding to random number generations in the left-hand side of the equivalence. (CryptoVerif may automatically add variables to the list $x_1, \dots, x_n$ if needed, except when a dot is added at the end of the list $x_1, \dots, x_n$. The transformation is applied only once. If several disjoint lists $x_1, \dots, x_n$ are possible and no variable name is mentioned, CryptoVerif makes a choice. It is better to mention at least one variable name when the left-hand side of the equivalence contains a random number generation, to make explicit which transformation should be applied.)

    In case the command ends with a dot (`.`), CryptoVerif never adds other variable names to those already listed. If the dot is absent, CryptoVerif may add other variable names if that seems necessary to perform the transformation.

  - `crypto` ⟨name⟩ `variables:` x_1`->`$y_1, \dots, x_n$`->`y_n`;` `terms:` o_1`->`$O_1, \dots, o_m$`->`O_m: apply a cryptographic transformation chosen as above, where
    - * $x_1, \dots, x_n$ are variable names of the game which correspond to random number generations $y_1, \dots, y_n$ respectively in the left-hand side of the equivalence. (CryptoVerif may automatically add variables to the list x_1`->`$y_1, \dots, x_n$`->`y_n if needed, except when a dot is added at the end of this list.)
    - * $o_1, \dots, o_m$ are occurrences of terms in the game, which will be transformed using oracles $O_1, \dots, O_m$ respectively of the equivalence. (CryptoVerif may automatically add elements to the list o_1`->`$O_1, \dots, o_m$`->`O_m if needed, except when a dot is added at the end of this list.)

    When the considered equivalence is defined inside a macro, macro expansion adds a prefix `@`k`_` to the variable and oracle names of the equivalence. This prefix must be included in the variable and oracle names used in this command. This happens in particular for primitives defined in the library of primitives of CryptoVerif. The right value of k in the prefix can be determined by issuing a command `crypto` without further indication. This command will display the equivalences as they are stored by CryptoVerif after macro expansion.

    One of the lists of variables or terms may be omitted. In this case, the separating semi-colon `;` is obviously omitted as well.

    When this command occurs in proof environment inside a CryptoVerif file, it is generally necessary to put the lists between quotes:

    `crypto` ⟨name⟩ `"variables:` x_1`->`$y_1, \dots, x_n$`->`y_n`;` `terms:` o_1`->`$O_1, \dots, o_m$`->`O_m`"`

- `insert_event` *e occ* replaces the subprocess at occurrence *occ* with the event `event` *e*. The games may be distinguished if and only if the event *e* is executed, and CryptoVerif then tries to bound the probability of executing that event. One should use the command `show_game occ` to determine which occurrence number *occ* corresponds to the program point where one wants to insert the event. The occurrence number *occ* must correspond to an output process (resp. oracle body in the oracles front-end).

- `insert` *occ ins* inserts instruction *ins* at occurrence *occ*. The instruction *ins* can be

  `new` ⟨vartype⟩
  `if` ⟨cond⟩ `then`
  `event` ⟨ident⟩`[(`seq⟨term⟩`)]`
  `let` ⟨pattern⟩ `=` ⟨term⟩ `in`
  `find` ⟨findbranch⟩ (`orfind` ⟨findbranch⟩)*

  or in the oracles front-end

  ⟨ident⟩ `<-R` ⟨ident⟩
  `if` ⟨cond⟩ `then`
  `event` ⟨ident⟩`[(`seq⟨term⟩`)]`
  ⟨ident⟩`[:`⟨ident⟩`]` `<-` ⟨term⟩
  `let` ⟨pattern⟩ `=` ⟨term⟩ `in`
  `find` ⟨findbranch⟩ (`orfind` ⟨findbranch⟩)*

  where ⟨findbranch⟩ ::= seq⟨identbound⟩ `suchthat` ⟨cond⟩ `then`

  In contrast to the initial game, the terms `new`, `if`, `find`, or `let` are not expanded, so `if`, `find`, `let` can occur only in conditions of `find` and `new` must not occur as a term. The variables of the inserted instruction are not renamed, so one must be careful when redefining variables with the same name. In particular, one is not allowed to add a new definition for a variable on which array accesses are done (because it could change the result of these array accesses). The obtained game must satisfy the required invariants (each variable is defined at most once in each branch of `if`, `find`, or `let`; each usage of a variable x must be either x without array index syntactically under its definition, inside a `defined` condition of a find, or $x[M_1, ..., M_n]$ under a `defined` condition that contains $x[M_1, ..., M_n]$ as a subterm). In case the inserted instruction is not appropriate, an error message explaining the problem is displayed.

The obtained game is indistinguishable from the initial game. The main practical usage of this command is to introduce case distinctions (`if`, `find`, or `let` with a pattern that is not a variable). In this situation, the process that follows the insertion point is duplicated in each branch of `if`, `find`, or `let`, and can subsequently be transformed in different ways in each branch. It may be useful to disable the merging of branches in simplification by `set mergeBranches = false` when a case distinction is inserted, so that the operation is not immediately undone at the next simplification.

One should use the command `show_game occ` to determine which occurrence number *occ* corresponds to the program point where one wants to insert the instruction. The occurrence number *occ* must correspond to an output process (resp. oracle body in the oracles front-end).

- `replace` *occ term* replaces the term at occurrence *occ* with the term *term*. Obviously, CryptoVerif must be able to prove that these two terms are equal. These terms must not contain `if`, `let`, `find`, `new`. One should use the command `show_game occ` to determine which occurrence number *occ* corresponds to the program point where one wants to replace the term. The occurrence number *occ* must correspond to a term.

- `merge_branches` merges the branches of `if`, `find`, and `let` when they execute equivalent code. Such a merging is already done in simplification, but `merge_branches` goes further. It performs several merges simultaneously and takes into account that merges may remove array accesses in conditions of `find` and thus allow further merges. Moreover, it advises `merge_arrays` when variables with different names and with array accesses are used in the branches that we may want to merge.

- `merge_arrays` x_{11} ... x_{1n} `,` ... `,` x_{k1} ... x_{kn} takes as argument k lists of n variables separated by commas. It merges the variables $x_{i1}, \ldots, x_{in}$ into x_{i1}. This is useful when these variables play the same role in different branches of `if`, `find`, `let`: merging them into a single variable may allow to merge the branches of `if`, `find`, `let` by `merge_branches`.

  The k lists to merge must contain the same number of variables n (at least 2). Variables x_{ij} and $x_{i'j'}$ for $i \neq i'$ must never be simultaneously defined for the same value of their array indices. Variables x_{ij} must have the same type and the same array indices for all j. Each variable x_{ij} must have a single definition, and must not be used in queries.

  In general, the variables x_{i1} should preferably belong to the `else` branch of the `if`, `find`, `let` that we want to merge later. Indeed, the code of the `else` branch is often more general than the code of the other branches (which may exploit the conditions that are tested), so merging towards the code of the `else` branch works more often.

  The variables x_{1j} should preferably be defined above the variables x_{ij} for any $i > 1$. If this is true, we can introduce special variables y_j at the definition site of x_{1j} which are used only for testing that branch j has been executed. This allows the merge to succeed more often.

- `quit`: terminate execution.

- `success`: test whether the desired properties are proved in the current game. If yes, display the proof and stop. Otherwise, wait for further instructions.

- `show_game`: display the current game.

- `show_game occ`: display the current game with occurrences. Useful for some commands that require specifying a program point; one can use the displayed numbers to specify program points.

- `show_state`: display the whole sequence of games until the current game.

- `show_facts` *occ*: show the facts that are proved by CryptoVerif in the current game, at the program point of occurrence *occ*. This command is mainly helpful for debugging.

- `auto`: switch to automatic mode; try to terminate the proof automatically from the current game.

- `set` $\langle\text{parameter}\rangle$ `=` $\langle\text{value}\rangle$: sets parameters, as the `set` instruction in input files.

- `allowed_collisions` $\langle\text{formulas}\rangle$: determine when to eliminate collisions. $\langle\text{formulas}\rangle$ is a list of comma separated formulas of the form $\langle\text{psize}\rangle_1\texttt{\^{}}n_1\,\texttt{*}\ldots\texttt{*}\langle\text{psize}\rangle_k\texttt{\^{}}n_k/\langle\text{tsize}\rangle$, where the exponents n_i can be omitted when equal to 1; $\langle\text{psize}\rangle_i$ is an identifier that determines the size of a parameter: `size`n for parameters of size n, `small` for size 0, `passive` for size 10, `noninteractive` for size 20; $\langle\text{tsize}\rangle$ is an identifier that determines the size of a type: `size`n for types of size n, `small` for size 0, `password` for size 10, `large` for size 20. (See also the declarations `param` and `type` for explanations of sizes.)

  Collisions are eliminated when the probability that they generate is at most of the form $\mathit{constant}\times p_1^{n_1}\times\cdots\times p_k^{n_k}\times\texttt{Pcoll1rand}(T)$, where p_i is a parameter of size at most $\langle\text{psize}\rangle_i$ and T is a type of size at least $\langle\text{tsize}\rangle$. By default, collisions are eliminated for $\mathit{anything}\times\texttt{Pcoll1rand}(T)$ when T is a `large` type, and for $p\times\texttt{Pcoll1rand}(T)$ when p is `small` and T is a `password` type.

  Additionally, $\langle\text{formulas}\rangle$ may also contain elements of the form $\texttt{collision}\,\texttt{*}\langle\text{psize}\rangle_1\texttt{\^{}}n_1\,\texttt{*}\ldots\texttt{*}\langle\text{psize}\rangle_k\texttt{\^{}}n_k$. These formulas allow the transformation of terms by `collision` statements, provided the number of times the collision statement is applied is at most $\mathit{constant}\times p_1^{n_1}\times\cdots\times p_k^{n_k}$ where p_i is a parameter of size at most $\langle\text{psize}\rangle_i$. By default, `collision` statements can always be applied.

- `undo`: undo the last transformation.

- `undo` n: undo the last n transformations.

- `restart`: restart the proof from the beginning. (Still simplify automatically the first game.)

- `interactive`: starts interactive mode. Allowed in `proof` environments, but not when one is already in interactive mode. Useful to start interactive mode after some proof steps.

The following indications can help finding a proof:

- When a message contains several nested cryptographic primitives, it is in general better to apply first the security definition of the outermost primitive.

- In order to distinguish more cases, one can start by applying the security of primitives used in the first messages, before applying the security of primitives used in later messages.

Running CryptoVerif inside a text editor such as `emacs` can be helpful, in order to use the search function to look for definitions or usages of variables in large games. For example, when trying to prove secrecy of x, one may look for usages of x, for definitions of x, and for usages of other variables used in those definitions.

8 Output of the system

The system outputs the executed transformations when it performs them. At the end, it outputs the sequence of games that leads to the proof of the desired properties. Between consecutive games, it prints the name of the performed transformation and details of what it actually did, and the formula giving the difference of probability between these games (if it is not 0). The description of the transformation between game may refer to program points in the previous game. These program points may not be completely accurate for the following reasons:

- When a step of the transformations transforms the same part of the game as a previous step, the occurrence in the second step actually refers to the code generated by the previous step, so it is not found in the previously displayed game.

- When a step transforms part of the game that was duplicated by a previous step of the transformation, the displayed program point is in fact ambiguous: one does not know which of the copies is actually transformed.

One can usually clarify the ambiguities by looking at the previous and next games.

Lines that begin with `RESULT` give the proved results. They may indicate that a property is proved and give an upper bound of the probability that the adversary breaks the property. In the end, they may also list the properties that could not be proved, if any.

When the `-tex` command-line option is specified, CryptoVerif also outputs a LaTeX file containing the sequence of games and the proved properties.

Correspondence between ACSII and LaTeX outputs To use nicer and more conventional notations, the LaTeX output sometimes differs from the ASCII output. Here is a table of correspondence:

ASCII	LaTeX
`<=(`p`)=>`	$\approx_p$
`&&`	$\wedge$
`||`	$\vee$
`<>`	$\neq$
`<=`	$\leq$
`orfind`	$\oplus$
`==>`	$\Longrightarrow$
For the `channels` front-end	
`in(`c`,`p`)`	$c(p)$
`in(`c`,(`$p_1, \dots, p_n$`))`	$c(p_1, \dots, p_n)$
`out(`c`,`M`)`	$\overline{c}\langle M\rangle$
`out(`c`,(`$M_1, \dots, M_n$`))`	$\overline{c}\langle M_1, \dots, M_n\rangle$
`!`N	$!^N$
`yield`	$\overline{0}$
`->`	$\rightarrow$
For the `oracles` front-end	
`<-`	$\leftarrow$
`<-R`	$\overset{R}{\leftarrow}$

9 Implementation

CryptoVerif can generate an OCaml implementation of the protocol from the CryptoVerif specification, using the option `-impl`.

CryptoVerif generates the code for the protocol itself, but the code for the cryptographic primitives and for interacting with the network and the application has to be manually written in OCaml.

- For the cryptographic primitives, one can specify which OCaml function corresponds to which CryptoVerif function as explained in Section 9.3 below. For the security guarantees to hold, the OCaml implementation must satisfy the security assumptions mentioned in the CryptoVerif specification. The subdirectory `implementation` provides a basic implementation for some cryptographic primitives, in the module `Crypto`. This module has two implementations:

  - `crypto_real.ml` corresponds to real cryptographic primitives, implemented by relying on the OCaml cryptographic library `Cryptokit` (`http://forge.ocamlcore.org/projects/cryptokit/`). You need to install this library in order to run the protocol implementations generated by CryptoVerif. (It is used at least for random number generation even if you implement the cryptographic primitives by other means.)
  - `crypto_dbg.ml` is a debugging implementation, which constructs terms instead of applying the real cryptographic primitives.

  You can choose which implementation to use by linking `crypto.ml` to the desired implementation. If you implement your own protocol, you will probably need to define your own cryptographic primitives.

  The module `Base` contains functions used by code generated by CryptoVerif. It should not be modified.

- The network and application code calls the code generated by CryptoVerif. From the point of view of security, this code can be considered as part of the adversary. We require that this code does not use unsafe OCaml functions (such as `Obj.magic` or marshalling/unmarshalling with different types) to bypass the typesystem (in particular to access the environment of closures and send it on the network).

  We also require that this code does not mutate the values received from or passed to functions generated by CryptoVerif. This can be guaranteed by using unmutable types, with the previous requirement. However, OCaml typically uses `string` for cryptographic functions and for network input/output, and the type `string` is mutable in OCaml. For simplicity and efficiency, the generated code uses the type `string`, with the requirement mentioned above.

  We also require that all data structures manipulated by the generated code are non-circular. This is necessary because we use OCaml structural equality to compare values, and this equality may not terminate in the presence of circular data structures. This can easily be guaranteed by requiring that all OCaml types declared in the CryptoVerif input file are non-recursive.

  We also require that this code does not fork after obtaining but before calling an oracle that can be called only once (because it is not under a replication in the CryptoVerif specification). Indeed, forking at this point would allow the oracle to be called several times. In practice, forking generally occurs only at the very beginning of the protocol, when the server starts a new session, so this requirement should be easily fulfilled.

  Finally, we require that the programs do not perform several simultaneous writes to the same file and do not simultaneously read and write in the same file. This requirement could be enforced using locks, but in practice, it is generally obtained for free if the programs are run as intended. More precisely, we have two categories of files:

  - Files that are created to store variables defined in a program and used in another program, for example, long-term keys generated by a key generation program, then used by the protocol. These files are written in one program, and read at the beginning of another program. These two programs should not be run concurrently, and the program that writes the file should be run once on each machine, not several times.
  - Files that store tables of keys. The programs that insert elements in the table should be run one at a time. The insertion in the table is actually appending the file, so the system should support reading the table while inserting elements in it. (Elements not yet completely inserted are ignored.)

The subdirectories `implementation/nspk` and `implementation/wlsk` provide two complete examples, with the CryptoVerif specification and the OCaml network and application code.

9.1 Restrictions on the processes for implementation

The following two constraints must be satisfied:

- `find` must not be used. You can obtain a similar result using `insert` and `get`, which are supported.

- Let us name "oracles" the parts of the process that are between an `in`/⟨ident⟩`(`seq⟨pattern⟩`) := ...` and an `out`/`return` statement, because in the oracle frontend, they correspond exactly to that.

  Let us define the signature of an oracle as the pair containing

  - the type $T_1 \times \ldots \times T_k \rightarrow T'_1 \times \ldots \times T'_n$, where $T_1 \times \ldots \times T_k$ are the types of the arguments expected in the `in`/⟨ident⟩`(`seq⟨pattern⟩`) :=` statement, and $T'_1 \times \ldots \times T'_n$ are the types of the result given in the `out`/`return` statements, and
  - the list containing for each of the following oracles, its name and whether it is under a replication or not.

  An oracle can have multiple `out`/`return` statements. To be able to implement it, we must be able to define the signature above for each oracle, that is, all `out`/`return` must return the same type of

⟨mod_opt⟩ ::= ⟨ident⟩(< | >)⟨string⟩

⟨odef⟩ :: + = ⟨ident⟩[[seq$^+$⟨mod_opt⟩]] { ⟨odef⟩

If channel frontend, ⟨obody⟩ :: + = out(⟨channel⟩, ⟨term⟩)[}][; ⟨odef⟩]

If oracle frontend, ⟨obody⟩ :: + = return(seq⟨term⟩)[}][; ⟨odef⟩]

Figure 5: Extensions to the syntax

elements, and the oracles present after each out/return statement must be the same. Moreover, if an oracle with the same name is defined at several places, all its definitions must have the same signature.

9.2 Defining modules

The syntax of the processes is extended to add annotations, described in Figure 5. The symbol :: + = means that we add the rule at the right-hand side to the non-terminal symbol at the left-hand side.

The terminals { and } are used to mark the boundary of a module. Different modules typically correspond to different programs, for instance, key generation, client, and server of a protocol. More precisely, the following two constructs define respectively the beginning and the end of a module:

- μ[x_1>"*filex*$_1$", $\dots$,x_n>"*filex*$_n$",y_1<"*filey*$_1$", $\dots$,y_m<"*filey*$_m$"] { Q: The module μ will contain the oracles defined in Q. The implementation of the module μ will write the contents of the variables $x_1, \dots, x_n$ upon instanciation in the files *filex*$_1$, $\dots$, *filex*$_n$ respectively. The variables $x_1, \dots, x_n$ must be defined under no replication inside module μ. These variables can then be used in other modules defined after the end of μ; these modules will read them automatically from the files *filex*$_1$, $\dots$, *filex*$_n$ respectively. The module μ will read at initialization the value of the variables $y_1, \dots, y_m$ from the files *filey*$_1$, $\dots$, *filey*$_m$ respectively. The variables $y_1, \dots, y_m$ must be free in μ. (They are defined before the beginning of μ.)
- In the channel frontend, out(c, t)}; Q, or in the oracle frontend return($t_1, \dots, t_n$)}; Q: The module being defined will not contain Q.

We transform the oracles present in the module into functions taking the arguments given to the oracle, and returning a tuple containing the result of the oracle and closures corresponding to the oracles following the current oracle that are in the same module. A module implementation contains only one function: the function init, which returns closures corresponding to the oracles accessible at the beginning of the module.

9.3 Implementation options

The implementation options declares how the implementation should translate functions, tables and types, and one must declare them after the declaration of the element it modifies and before use. The syntax is described in Figure 6.

The available implementation options are described hereafter:

- type T="ty": Sets the OCaml type ty to be the type corresponding to the type T.

  This also can be followed by options between brackets and separated by semicolons. These options are:

  – serial="s","d": Sets the serialization/deserialization of the type. There is no default, and this is required when a variable of type T is written or read to a file/table, or when it is contained in a tuple. The serialization function s must be of type ty → string, the deserialization function d must be of type string → ty. When deserialization fails, it must raise exception Match_fail.

seq;$^+$⟨N⟩ ::= N | N;seq;$^+$⟨N⟩

⟨impl_block⟩ ::= `implementation` ⟨impl_opt⟩(;⟨impl_opt⟩)*.

⟨type_opt⟩ ::= ⟨ident⟩=seq$^+$⟨string⟩

⟨fun_opt⟩ ::= ⟨ident⟩=⟨string⟩

⟨impl_opt⟩ ::= `type` ⟨ident⟩=⟨string⟩ [[seq;$^+$⟨type_opt⟩]]

| `type` ⟨ident⟩=⟨integer⟩ [[seq;$^+$⟨type_opt⟩]]

| `table` ⟨ident⟩=⟨string⟩

| `fun` ⟨ident⟩=⟨string⟩ [[seq;$^+$⟨fun_opt⟩]]

| `const` ⟨ident⟩=⟨string⟩

Figure 6: Grammar for implementation options

  - `pred="p"`: Sets the predicate function, this function must be an OCaml function of type `ty` → `bool`. It returns whether an element is of type T or not. The default predicate function is a function that accepts every element.
  - `random="f"`: Sets the random generation function. This function must be an OCaml function of type `unit` → `ty`, and must return uniformly a random element of type `ty`. In particular, if a predicate function has been defined, the predicate function must return `true` on every element returned by the random generation function.

- `type` T=n: Sets the size of the `fixed` type T. The size must be a multiple of 8 and then will be represented by a string or 1 and then by a boolean. This can be followed by options between brackets and separated by semicolons. The only allowed option is:
  - `serial="s","d"`: Modifies the default serialization/deserialization of the type (used when a variable of this type is read/written to a file/table).
- `table` *tbl*=`"file"`: Sets the file in which the table *tbl* is written.
- `fun` f=`"s"`: Sets the implementation of the function f to the OCaml function `s`. If the function f takes arguments of type $T_1 \times \ldots \times T_n$ and returns a result of type T, the type of `s` must be $st_1 \rightarrow st_2 \rightarrow \ldots \rightarrow st_n \rightarrow st$, where for all i between 1 and n, st_i must be the corresponding type declared using the `type` declaration for the type T_i, and st is the corresponding type for T. For functions f with no arguments, the type of the function `s` must be `unit` → st, with st the type corresponding to T. This can take the following options:
  - `inverse="s_inv"`: If f has the `compos` attribute, this declares `s_inv` as the inverse function. With the previous notations, this function must be of type $st \rightarrow st_1 \times st_2 \times \ldots \times st_n$. `s_inv` x must return a tuple $(x_1, \ldots, x_n)$ such that `s` x_1 ... $x_n = x$. If there is no such element, `s_inv` must raise `Match_fail`.

  CryptoVerif allows one to define macros by `letfun`. Specifying an OCaml implementation for these macros is optional. When the OCaml implementation is not specified, CryptoVerif generates code according to the `letfun` macro. When the OCaml implementation is specified, it is used when generating the OCaml code, while the CryptoVerif macro defined by `letfun` is used for proving the protocol. This feature can be used, for instance, to define probabilistic functions: the OCaml implementation generates the random choices inside the function, while the CryptoVerif definition by `letfun` first makes the random choices, then calls a deterministic function.
- `const` f=`"s"`: Sets the implementation of the function f that has no arguments to an OCaml constant. If the constant is a string, one can write, for example, `const` f=`"\"constant\""`.

Acknowledgments

CryptoVerif was partly developed while the authors were at École Normale Supérieure, Paris.

References

[1] B. Blanchet. A computationally sound mechanized prover for security protocols. Cryptology ePrint Archive, Report 2005/401, Nov. 2005. Available at `http://eprint.iacr.org/2005/401`.

[2] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, Oakland, California, May 2006. Extended version available as ePrint Report 2005/401, `http://eprint.iacr.org/2005/401`.

[3] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In C. Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554, Santa Barbara, CA, Aug. 2006. Springer.

[4] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. Cryptology ePrint Archive, Report 2006/069, Feb. 2006. Available at `http://eprint.iacr.org/2006/069`.

[5] P. Laud. Secrecy types for a simulatable cryptographic library. In *12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 26–35, Alexandria, VA, Nov. 2005. ACM.