CryptoVerif
Computationally Sound, Automatic Cryptographic Protocol Verifier
User Manual

Bruno Blanchet
INRIA, École Normale Supérieure, CNRS, Paris

May 5, 2011

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://www.di.ens.fr/~blanchet/publications.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].
- `-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.

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`, `equiv`, `event`, `expand`, `find`, `forall`, `fun`, `if`, `in`, `inj`, `length`, `let`, `max`, `maxlength`, `new`, `newChannel`, `orfind`, `otheruses`, `out`, `param`, `proba`, `process`, `proof`, `query`, `secret`, `secret1`, `set`, `suchthat`, `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`(`$M_1, \ldots, M_n$`)` applies function f to the result of $M_1, \ldots, M_n$.
- The tuple application `(`$M_1, \ldots, M_n$`)` builds a tuple from $M_1, \ldots, M_n$ (corresponds to the concatenation of $M_1, \ldots, M_n$ with length and type indications so that $M_1, \ldots, M_n$ can be recovered without ambiguity). This is allowed only for $n \neq 1$, so that it is distinguished from parenthesing.
- The array access x`[`$M_1, \ldots, M_n$`]` returns the cell of indexes $M_1, \ldots, M_n$ 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 `new`, `if`, `find`, `let` 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 construct `new` is not allowed to occur in conditions of `find`. We refer the reader to the description of processes below for a fully detailed explanation.

- `new` x`:`T`;`M chooses a new random number uniformly 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`(`$p_1, \ldots, p_n$`)` 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 $f(M_1, \ldots, M_n)$.)
  - `(`$p_1, \ldots, p_n$`)` 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`.

⟨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` ⟨tfindbranch⟩ (`orfind` ⟨tfindbranch⟩)* `else` ⟨term⟩
| `event` ⟨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⟩ = $[(\langle X\rangle\texttt{,})^*\langle X\rangle] = \langle X\rangle\texttt{,}\ldots\texttt{,}\langle X\rangle$. (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$^+\langle X\rangle = (\langle X\rangle\texttt{,})^*\langle X\rangle = \langle X\rangle\texttt{,}\ldots\texttt{,}\langle X\rangle$. (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⟩
| `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`]` `[[required]]` `:=` ⟨term⟩
| `![`⟨ident⟩ `<=]` ⟨ident⟩`(new` ⟨vartype⟩`;)`*⟨fungroup⟩
| `![`⟨ident⟩ `<=]` ⟨ident⟩`(new` ⟨vartype⟩`;)`*`(`seq$^+$⟨fungroup⟩`)`
⟨funmode⟩ ::= ⟨fungroup⟩ `[exist]`
| ⟨fungroup⟩ `[all]`

⟨channel⟩ ::= ⟨ident⟩`[`seq⟨term⟩`]`
⟨oprocess⟩ ::= ⟨ident⟩
| `(`⟨oprocess⟩`)`
| `yield`
| `event` ⟨ident⟩`(`seq⟨term⟩`)` `[;` ⟨oprocess⟩`]`
| `new` ⟨vartype⟩`[;` ⟨oprocess⟩`]`
| `let` ⟨pattern⟩ `=` ⟨term⟩ `[in` ⟨oprocess⟩ `[else` ⟨oprocess⟩`]]`
| `if` ⟨cond⟩ `then` ⟨oprocess⟩ `[else` ⟨oprocess⟩`]`
| `find` ⟨findbranch⟩ `(orfind` ⟨findbranch⟩`)`* `[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

- `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 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.

The construct `event` in terms is also similar to the one for processes, except that it is restricted to events without arguments `event` f and that it aborts the process immediately after the event. It is not allowed to occur in conditions of `find` or in input channels. 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.
- `new` x`:`T`;`P chooses a new random number uniformly in type T, stores it in x, and executes P. T must be declared with option `fixed`.
- `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 $f(M_1, \ldots, M_n)$.)

– $(p_1, \ldots, p_n)$ 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'.

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 $FB_j = u_{j1}$`<=`$n_{j1}, \ldots, u_{jm_j}$`<=`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`<=`$n_1, \ldots, u_m$`<=`n_m `suchthat defined(`$M_1, \ldots, M_l$`) && `M `then` P' `else` P tries to find values of $u_1, \ldots, u_m$ for which $M_1, \ldots, M_l$ 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 $FB_j = u_{j1}$`<=`$n_{j1}, \ldots, u_{jm_j}$`<=`n_{jm_j} `suchthat defined(`$M_{j1}, \ldots, M_{jl_j}$`) && `M_j `then` P_j tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \ldots, u_{jm_j}$ for which $M_{j1}, \ldots, M_{jl_j}$ 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(`$M_{j1}, \ldots, M_{jl_j}$`) && `M_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 executes P. Otherwise, it chooses randomly with uniform[1] probability one j and one value of $u_{j1}, \ldots, u_{jm_j}$ such that the corresponding condition is `true`, and executes P_j.

  In the general case, the conditions $cond_j$ are of the form `defined(`$M_1, \ldots, M_l$`)` [`&&` M] or simply M. The condition `defined(`$M_1, \ldots, M_l$`)` means that $M_1, \ldots, M_l$ 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`.

- Finally, let us explain the output `out(`c`[`$M_1, \ldots, M_l$`],`N`);`Q. A channel c`[`$M_1, \ldots, M_l$`]` consists of both a channel name c (declared by `channel` c) and a tuple of terms $M_1, \ldots, M_l$. Terms $M_1, \ldots, M_l$ 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`[`$M_1, \ldots, M_l$`],`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`[`$M'_1, \ldots, M'_l$`],`p`);`P is chosen randomly with 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 $f(\ldots)$ must be declared with their type.

[1] A probabilistic polynomial-time Turing machine can choose a random number uniformly in a set of cardinal m only when m is a power of 2. When m is not a power of 2, there exist approximate algorithms: for example, in order to obtain a random integer in $[0, m-1]$, we can choose a random integer r uniformly among $[0, 2^k - 1]$ for a certain k large enough and return $r \mod m$. The distribution can be made as close as we wish to the uniform distribution by choosing k large enough.

When a channel is just a channel name c, it is an abbreviation for $c[i_1, \ldots, i_n]$ where $i_1, \ldots, i_n$ 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 $i_1, \ldots, i_n$: x stands for $x[i_1, \ldots, i_n]$. 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: $x[u_1, \ldots, u_k]$ stands for $x[i_1, \ldots, i_{n-k}, u_1, \ldots, u_k]$ where $i_1, \ldots, i_{n-k}$ must be the $n - k$ first replication indexes both at the creation of x and at the usage $x[u_1, \ldots, u_k]$. (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.
- $x[M_1, \ldots, M_n]$ in M or P in a find branch ... `suchthat defined(`$M'_1, \ldots, M'_l$`) && `M `then` P, such that $x[M_1, \ldots, M_n]$ is a subterm of $M'_1, \ldots, M'_l$.
- $x[M_1, \ldots, M_n]$ in M or M'' in a find branch ... `suchthat defined(`$M'_1, \ldots, M'_l$`) && `M `then` M'', such that $x[M_1, \ldots, M_n]$ is a subterm of $M'_1, \ldots, M'_l$.

These syntactic constraints guarantee that a variable is accessed only when it is defined.

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`[`$M_1, \ldots, M_m$`]`, $M_1, \ldots, M_m$ must be of the suitable interval type, that is, a suffix of the types of replication indexes at the definition of x. In f`(`$M_1, \ldots, M_m$`)`, if f has been declared by `fun` f`(`$T_1, \ldots, T_m$`):`T, M_j must be of type T_j, and $f(M_1, \ldots, M_m)$ is then of type T. In `(`$M_1, \ldots, M_n$`)`, 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`(`$M_1, \ldots, M_n$`)`, if e has been declared by `event` e`(`$T_1, \ldots, T_n$`)`, M_j must be of type T_j. In `new` x`:`T, T must be declared with option `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`[`$M_1, \ldots, M_n$`],`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`, simplification tries 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 = false.`
    `set mergeArrays = true.`

    When `true`, simplification 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 commands `merge_branches` and/or `merge_arrays` (defined in Section 7) to perform the merging.

    When it is set to `true`, simplification often undoes `SArename`. That is why it is not set to `true` by default.

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

    We say that a `find` is *unique* when there is at most one branch and one value of the indices that we look up, for which the conditions are true. When `uniqueBranch = true` and some `find`s are proved to be unique, several transformations are enabled as part of `simplify`:

    * If a branch of a `find` is proved to succeed and that `find` is unique, then simplification removes all other branches of that `find`.

* If a `find` occurs in the `then` branch of a `find` and both `find`s are unique, we reorganize them.
* If a `find` occurs in the condition of a `find` and the inner `find` is unique, we reorganize them.

When `uniqueBranch = 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 noAdviceSimplify = false.`
`set noAdviceSimplify = true.`

When `noAdviceSimplify = true`, prevents the simplification from generating advice. Useful when the simplification 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 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 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.

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` $n_1, \ldots, n_m$`.` declares parameters $n_1, \ldots, n_m$. 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 one can generate random numbers with uniform probability in the considered type. In particular, the type is a finite set, so `fixed` implies `bounded`.

    The recommended usage of `fixed` is to declare "`fixed`" types that consist of the set of all bitstrings of a certain length n. Standard random number generators return random bitstrings among such sets. Random bitstrings in other sets, or random bitstrings with non-uniform distributions, can be obtained by applying a function to a random bitstring choosen uniformly among the bitstrings of length n.

    More generally, it is however possible to declare "`fixed`" other sets of bitstrings of cardinal 2^n for some n, when there is a polynomial-time bijection from the set of bitstrings of length n to these sets, since probabilistic bounded-time Turing machines can also choose random numbers uniformly in such sets.

    For finite sets whose cardinal is not a power of 2, probabilistic bounded-time Turing machines can choose random numbers with a distribution as close as we wish to uniform, but not exactly uniform. If you wish to consider such a distribution as a uniform distribution, although this is not completely rigorous, you can declare finite sets whose cardinal is not a power of 2 as "`fixed`".
  - `large`, `password`, and `size`n indicate the size 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 larger the type. CryptoVerif uses this information to determine whether collisions between two elements the considered type of T chosen randomly with uniform probability should be eliminated. By default, collisions are eliminated automatically for types of size at least 15, and may be manually eliminated for types of size at least 5 by the command `simplify coll_elim` ....

    `large` means that the type T is large enough so that collisions between two elements of T chosen randomly with uniform probability can be eliminated. (In asymptotic analyses, one over the cardinal of T is negligible. In exact security analyses, the probability of a collision is correctly expressed by the system as a function of the cardinal of T.) `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 eliminating collisions among passwords yields larger probability bounds than amond data of `large` types. `password` is equivalent to `size10`.

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

  `fun` f`(`$T_1, \ldots, T_n$`):`T`.` declares a function that takes n arguments, of types $T_1, \ldots, T_n$, 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: $f(x_1, \ldots, x_m) = y$ implies for $i \in \{1, \ldots, m\}$, $x_i = f_i^{-1}(y)$ 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 a uniform distribution into a uniform distribution. f must be unary.
- `commut` 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 $f(x, y) \rightarrow f(y, x)$ does not terminate.)

- `const` seq$^+$⟨ident⟩`:`⟨ident⟩`.`

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

- `channel` seq$^+$⟨ident⟩`.`

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

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

  `event` e`(`$T_1, \ldots, T_n$`).` declares an event e that takes arguments of types $T_1, \ldots, T_n$. 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` $x_1 : T_1, \ldots, x_n : T_n$`;`$M$`.` says that for all values of $x_1, \ldots, x_n$ in types $T_1, \ldots, T_n$ 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.

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

  `collision new` x_1`:`T_1`;` ... `new` x_n`:`T_n`;forall` $y_1 : T'_1, \ldots, y_m : T'_m$`;`$M_1$ `<=(`p`)=>` M_2`.` means that when $x_1, \ldots, x_n$ are chosen randomly with uniform probability and independently in $T_1, \ldots, T_n$ respectively, a Turing machine running in time `time` has probability at most p of finding $y_1, \ldots, y_m$ in $T'_1, \ldots, T'_m$ 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 $x_1, \ldots, x_n$ are created by independent random number generations of types $T_1, \ldots, T_n$ respectively. One should be careful of the orientation of the equivalence, in particular for termination.

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

  `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 functions. (They can be translated into processes as explained in [1].)

  - O`(`$x_1 : T_1, \ldots, x_n : T_n$`) :=` M represents a function O that takes arguments $x_1, \ldots, x_n$ of types $T_1, \ldots, T_n$ 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(`$x_1 : T_1, \ldots, x_n : T_n$`) [`$n$`] :=` 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 `[required]` can also be added in the previous functions, which become `O(`$x_1 : T_1, \ldots, x_n : T_n$`) [required] :=` M This indication is also used for the proof strategy: CryptoVerif applies the transformation defined by the equivalence only when the considered function is used at least once in the game.

– `!`i`<=`N `new` y_1`:`T'_1`; ... new` y_m`:`T'_m`;(`$FG_1, \ldots, FG_n$`)` represents N copies of a process that picks fresh random numbers $y_1, \ldots, y_m$ of types $T'_1, \ldots, T'_m$ respectively, and makes available the functions described in $FG_1, \ldots, FG_n$. 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` ... `orfind` u_1 `<=` $N_1, \ldots, u_m$ `<=` N_m `suchthat defined(`$z_1[\widetilde{u_1}], \ldots, z_l[\widetilde{u_l}]$`) &&` M `then` FP ... `else` FP'

where $\widetilde{u_k}$ is a non-empty prefix of $u_1, \ldots, u_m$, at least one $\widetilde{u_k}$ for $1 \leq k \leq l$ is the whole sequence $u_1, \ldots, u_m$, and the implicit prefix of the current array indexes is the same for all $z_1, \ldots, z_l$. (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 $z_1, \ldots, z_l$.) Furthermore, there must exist $k \in \{1, \ldots, l_j\}$ 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.
  - $x_1 : T_1, \ldots, x_n : T_n$; `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), $\ldots$, $e_n(M_{n1}, \ldots, M_{nm_n})$ (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` {$\langle$command$\rangle$`;` ... `;`$\langle$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`$\langle$ident$\rangle$`(`seq$\langle$ident$\rangle$`)` {seq$\langle$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`⟨ident⟩`(`seq⟨ident⟩`).`

  `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 $\bot$. 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 $\bot$. (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, 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((`$T_1, \ldots, T_n$`),` $p_1, \ldots, p_n$`)` designates the maximal length of the result of the tuple function from $T_1 \times \ldots \times T_m$ to `bitstring`, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of this function. n is an integer constant. `time` designates the runtime of the environment (attacker). Finally, `time(`$\ldots$`)` designates the runtime time of each elementary action of a game:

- `time(`$f, p_1, \ldots, p_n$`)` designates the maximal runtime of one call to function symbol f, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of f.
- `time(let` $f, p_1, \ldots, p_n$`)` designates the maximal runtime of one pattern matching operation with function symbol f, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of f.
- `time((`$T_1, \ldots, T_m$`),` $p_1, \ldots, p_n$`)` designates the maximal runtime of one call to the tuple function from $T_1 \times \ldots \times T_m$ to `bitstring`, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of this function.
- `time(let(`$T_1, \ldots, T_m$`),` $p_1, \ldots, p_n$`)` designates the maximal runtime of one pattern matching with the tuple function from $T_1 \times \ldots \times T_m$ to `bitstring`, where $p_1, \ldots, p_n$ 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 uniformly.
- `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 [`$T_1, \ldots, T_m$`]`$T, p_1, \ldots, p_n$`)` represents the time of an output in which the channel indexes are of types $T_1, \ldots, T_m$, the output bitstring is of type T, and the maximum length of the channel indexes and the output bitstring is represented by $p_1, \ldots, p_n$ 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`, `equiv`, `event`, `expand`, `find`, `forall`, `foreach`, `fun`, `if`, `in`, `inj`, `length`, `let`, `max`, `maxlength`, `newOracle`, `orfind`, `otheruses`, `param`, `proba`, `process`, `proof`, `query`, `return`, `secret`, `secret1`, `set`, `suchthat`, `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:

⟨declaration⟩* `process` ⟨odef⟩

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`(`$M_1, \ldots, M_n$`)` applies function f to the result of $M_1, \ldots, M_n$.
- The tuple application `(`$M_1, \ldots, M_n$`)` builds a tuple from $M_1, \ldots, M_n$ (corresponds to the concatenation of $M_1, \ldots, M_n$ with length and type indications so that $M_1, \ldots, M_n$ can be recovered without ambiguity). This is allowed only for $n \neq 1$, so that it is distinguished from parenthesing.
- The array access x`[`$M_1, \ldots, M_n$`]` returns the cell of indexes $M_1, \ldots, M_n$ 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` ⟨tfindbranch⟩ (`orfind` ⟨tfindbranch⟩)* `else` ⟨term⟩
| `event` ⟨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⟩ = $[(\langle X\rangle\,\texttt{,})^*\langle X\rangle] = \langle X\rangle\,\texttt{,}\,\ldots\,\texttt{,}\langle X\rangle$. (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⟩ = $(\langle X\rangle\,\texttt{,})^*\langle X\rangle = \langle X\rangle\,\texttt{,}\,\ldots\,\texttt{,}\langle X\rangle$. (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⟩
| `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`]]` `[[required]]` `:=` ⟨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⟩`[[`seq⟨term⟩`]]`

⟨obody⟩ ::= ⟨ident⟩
| `(`⟨obody⟩`)`
| `end`
| `event` ⟨ident⟩`[(`seq⟨term⟩`)]` [`;` ⟨obody⟩]
| ⟨ident⟩ `<-R` ⟨ident⟩[`;` ⟨obody⟩]
| ⟨ident⟩[`:`⟨ident⟩] `<-` ⟨term⟩[`;` ⟨obody⟩]
| `let` ⟨pattern⟩ `=` ⟨term⟩ [`in` ⟨obody⟩ [`else` ⟨obody⟩]]
| `if` ⟨cond⟩ `then` ⟨obody⟩ [`else` ⟨obody⟩]
| `find` ⟨findbranch⟩ (`orfind` ⟨findbranch⟩)* [`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`, `<-`, `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` and in input channels. The construct `<-R` is not allowed to occur in conditions of `find`. 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 uniformly 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`(`$p_1, \ldots, p_n$`)` 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 $f(M_1, \ldots, M_n)$.)
  - `(`$p_1, \ldots, p_n$`)` 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 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.

The construct `event` in terms is also similar to the one for processes, except that it is restricted to events without arguments `event` f and that it aborts the process immediately after the event. It is not allowed to occur in conditions of `find` or in input channels. 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 $i_1, \ldots, i_n$.

- O`(`$p_1, \ldots, p_n$`) :=` P defines an oracle O taking arguments $p_1, \ldots, p_n$, and returning the result of the oracle body P. The patterns $p_1, \ldots, p_n$ are as in the `let` construct above, except that variables in p that are not under a function symbol $f(\ldots)$ 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`(`$M_1, \ldots, M_n$`);`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.
- x `<-R` T`;`P chooses a new random number uniformly in type T, stores it in x, and executes P. T must be declared with option `fixed`.
- x[`:`T] `<-` M`;`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[`:`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[`:`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 $f(M_1, \ldots, M_n)$.)
  - `(`$p_1, \ldots, p_n$`)` 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'.

  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` ... `orfind` FB_m `else` P where each branch FB_j is $FB_j = u_{j1}$`<=`$n_{j1}, \ldots, u_{jm_j}$`<=`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`<=`$n_1, \ldots, u_m$`<=`n_m `suchthat defined(`$M_1, \ldots, M_l$`) &&` M `then` P' `else` P tries to find values of $u_1, \ldots, u_m$ for which $M_1, \ldots, M_l$ 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 $FB_j = u_{j1}$`<=`$n_{j1}, \ldots, u_{jm_j}$`<=`n_{jm_j} `suchthat defined(`$M_{j1}, \ldots, M_{jl_j}$`) &&` M_j `then` P_j tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \ldots, u_{jm_j}$ for which $M_{j1}, \ldots, M_{jl_j}$ 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(`$M_{j1}, \ldots, M_{jl_j}$`) &&` M_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 executes P. Otherwise, it chooses randomly with uniform[2] probability one j and one value of $u_{j1}, \ldots, u_{jm_j}$ such that the corresponding condition is `true`, and executes P_j.

In the general case, the conditions $cond_j$ are of the form `defined(`$M_1, \ldots, M_l$`)` [`&&` M] or simply M. The condition `defined(`$M_1, \ldots, M_l$`)` means that $M_1, \ldots, M_l$ 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`.

- `return(`$N_1, \ldots, N_l$`);`Q terminates the oracle, returning the result of the terms $N_1, \ldots, N_l$. 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 $i_1, \ldots, i_n$: x stands for $x[i_1, \ldots, i_n]$. 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: $x[u_1, \ldots, u_k]$ stands for $x[i_1, \ldots, i_{n-k}, u_1, \ldots, u_k]$ where $i_1, \ldots, i_{n-k}$ must be the $n - k$ first replication indexes both at the creation of x and at the usage $x[u_1, \ldots, u_k]$. (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 $O[i_1, \ldots, i_n]$.

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

These syntactic constraints guarantee that a variable is accessed only when it is defined.

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 $[1, N_1] \times \ldots \times [1, N_n] \to T$. 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`[`$M_1, \ldots, M_m$`]`, $M_1, \ldots, M_m$ must be of the suitable interval type, that is, a suffix of the types of replication indexes at the definition of x. In f`(`$M_1, \ldots, M_m$`)`, if f has been declared by `fun` f`(`$T_1, \ldots, T_m$`):`T, M_j must be of type T_j, and $f(M_1, \ldots, M_m)$ is then of type T. In `(`$M_1, \ldots, M_n$`)`, 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.

[2] A probabilistic polynomial-time Turing machine can choose a random number uniformly in a set of cardinal m only when m is a power of 2. When m is not a power of 2, there exist approximate algorithms: for example, in order to obtain a random integer in $[0, m-1]$, we can choose a random integer r uniformly among $[0, 2^k - 1]$ for a certain k large enough and return $r \mod m$. The distribution can be made as close as we wish to the uniform distribution by choosing k large enough.

Furthermore, in `event` e`(`$M_1, \ldots, M_n$`)`, if e has been declared by `event` e`(`$T_1, \ldots, T_n$`)`, M_j must be of type T_j. In x `<-R` T, T must be declared with option `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(`$M_1, \ldots, M_n$`)`, 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(`$M_1, \ldots, M_n$`);`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`(`$p_1, \ldots, p_m$`) :=` 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`, simplification tries 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 = false.`
    `set mergeArrays = true.`

When `true`, simplification 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 commands `merge_branches` and/or `merge_arrays` (defined in Section 7) to perform the merging.

When it is set to `true`, simplification often undoes `SArename`. That is why it is not set to `true` by default.

– `set uniqueBranch = true.`
`set uniqueBranch = false.`

We say that a `find` is *unique* when there is at most one branch and one value of the indices that we look up, for which the conditions are true. When `uniqueBranch = true` and some `find`s are proved to be unique, several transformations are enabled as part of `simplify`:

* If a branch of a `find` is proved to succeed and that `find` is unique, then simplification removes all other branches of that `find`.
* If a `find` occurs in the `then` branch of a `find` and both `find`s are unique, we reorganize them.
* If a `find` occurs in the condition of a `find` and the inner `find` is unique, we reorganize them.

When `uniqueBranch = 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 noAdviceSimplify = false.`
  `set noAdviceSimplify = true.`

  When `noAdviceSimplify = true`, prevents the simplification from generating advice. Useful when the simplification 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 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 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.

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` $n_1, \ldots, n_m$`.` declares parameters $n_1, \ldots, n_m$. 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 $\bot$ (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 $\bot$. In other words, the type is a finite subset of bitstrings plus $\bot$.
  - `fixed` means that one can generate random numbers with uniform probability in the considered type. In particular, the type is a finite set, so `fixed` implies `bounded`.

    The recommended usage of `fixed` is to declare "`fixed`" types that consist of the set of all bitstrings of a certain length n. Standard random number generators return random bitstrings among such sets. Random bitstrings in other sets, or random bitstrings with non-uniform distributions, can be obtained by applying a function to a random bitstring choosen uniformly among the bitstrings of length n.

    More generally, it is however possible to declare "`fixed`" other sets of bitstrings of cardinal 2^n for some n, when there is a polynomial-time bijection from the set of bitstrings of length n to these sets, since probabilistic bounded-time Turing machines can also choose random numbers uniformly in such sets.

    For finite sets whose cardinal is not a power of 2, probabilistic bounded-time Turing machines can choose random numbers with a distribution as close as we wish to uniform, but not exactly uniform. If you wish to consider such a distribution as a uniform distribution, although this is not completely rigorous, you can declare finite sets whose cardinal is not a power of 2 as "`fixed`".
  - `large`, `password`, and `size`n indicate the size 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 larger the type. CryptoVerif uses this information to determine whether collisions between two elements the considered type of T chosen randomly with uniform probability should be eliminated. By default, collisions are eliminated automatically for types of size at least 15, and may be manually eliminated for types of size at least 5 by the command `simplify coll_elim` ....

    `large` means that the type T is large enough so that collisions between two elements of T chosen randomly with uniform probability can be eliminated. (In asymptotic analyses, one over the cardinal of T is negligible. In exact security analyses, the probability of a collision is correctly expressed by the system as a function of the cardinal of T.) `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 eliminating collisions among passwords yields larger probability bounds than amond data of `large` types. `password` is equivalent to `size10`.

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

  `fun` f`(`$T_1, \ldots, T_n$`):`T`.` declares a function that takes n arguments, of types $T_1, \ldots, T_n$, 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: $f(x_1, \ldots, x_m) = y$ implies for $i \in \{1, \ldots, m\}$, $x_i = f_i^{-1}(y)$ 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 a uniform distribution into a uniform distribution. f must be unary.
  - `commut` 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 $f(x, y) \rightarrow f(y, x)$ does not terminate.)

- `const` seq$^+$⟨ident⟩`:`⟨ident⟩`.`

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

- `event` ⟨ident⟩`[(`seq⟨ident⟩`)]`.

  `event` e`(`$T_1, \ldots, T_n$`).` declares an event e that takes arguments of types $T_1, \ldots, T_n$. 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` $x_1 : T_1, \ldots, x_n : T_n$`;`$M$`.` says that for all values of $x_1, \ldots, x_n$ in types $T_1, \ldots, T_n$ 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.

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

  `collision` x_1 `<-R` T_1`;` $\ldots$ x_n `<-R` T_n`;forall` $y_1 : T'_1, \ldots, y_m : T'_m$`;` `return(`$M_1$`) <=(`$p$`)=> return(`$M_2$`).`
  means that when $x_1, \ldots, x_n$ are chosen randomly with uniform probability and independently in $T_1, \ldots, T_n$ respectively, a Turing machine running in time `time` has probability at most p of finding $y_1, \ldots, y_m$ in $T'_1, \ldots, T'_m$ 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 $x_1, \ldots, x_n$ are created by independent random number generations of types $T_1, \ldots, T_n$ 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; ... x_m `<-R` T_m; Q in fact stands for `foreach` i`<=`N `do` O`() :=` x_1 `<-R` T_1; ... 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`(`$x_1 : T_1, \ldots x_n : T_n$`) [`$n$`] :=` 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 `[required]` can also be added in the definition of an oracle, which becomes O`(`$x_1 : T_1, \ldots x_n : T_n$`) [required] :=` P. This indication is also used for the proof strategy: CryptoVerif applies the transformation defined by the equivalence only when the considered function is used at least once 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 `<=` $N_1, \ldots, u_m$ `<=` N_m `suchthat defined(`$z_1[\widetilde{u_1}], \ldots, z_l[\widetilde{u_l}]$`) &&` M `then` FP... `else` FP'

    where $\widetilde{u_k}$ is a non-empty prefix of $u_1, \ldots, u_m$, at least one $\widetilde{u_k}$ for $1 \leq k \leq l$ is the whole sequence $u_1, \ldots, u_m$, and the implicit prefix of the current array indexes is the same for all $z_1, \ldots, z_l$. (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 $z_1, \ldots, z_l$.) Furthermore, there must exist $k \in \{1, \ldots, l_j\}$ 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`(`$y_1, \ldots, y_n$`)`. 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`,...,`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 an oracle, designates the number of different calls to the oracle O. $|T|$, where T has been declared by `type` T, 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((`$T_1, \ldots, T_n$`),`$p_1, \ldots, p_n$`)` designates the maximal length of the result of the tuple function from $T_1 \times \ldots \times T_m$ to `bitstring`, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of this function. n is an integer constant. `time` designates the runtime of the environment (attacker). Finally, `time(...)` designates the runtime time of each elementary action of a game:

- `time(`$f, p_1, \ldots, p_n$`)` designates the maximal runtime of one call to function symbol f, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of f.
- `time(let` $f, p_1, \ldots, p_n$`)` designates the maximal runtime of one pattern matching operation with function symbol f, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of f.
- `time((`$T_1, \ldots, T_m$`),`$p_1, \ldots, p_n$`)` designates the maximal runtime of one call to the tuple function from $T_1 \times \ldots \times T_m$ to `bitstring`, where $p_1, \ldots, p_n$ represent the maximum length of the non-bounded arguments of this function.
- `time(let(`$T_1, \ldots, T_m$`),`$p_1, \ldots, p_n$`)` designates the maximal runtime of one pattern matching with the tuple function from $T_1 \times \ldots \times T_m$ to `bitstring`, where $p_1, \ldots, p_n$ 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 uniformly.
- `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`, (`$x_1 : T_1, \ldots, x_l : T_l$`));`$P$	O`(`$x_1 : T_1, \ldots, x_l : T_l$`) := `$P$
`out(`c`, (`$M_1, \ldots, M_l$`));`Q	`return(`$M_1, \ldots, M_l$`);`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 `fixed` (to be able to generate random numbers from it), typically `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 `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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 `fixed` (to be able to generate random numbers from it), typically `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 `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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 `fixed` (to be able to generate random numbers from it), typically `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 `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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 `fixed` (to be able to generate random numbers from it), typically `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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 `fixed` (to be able to generate random numbers from it), typically `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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 `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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 `fixed` (to be able to generate random numbers from it), typically `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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.

- `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 `fixed` (to be able to generate random numbers from it), typically `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 `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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 `fixed` (to be able to generate random numbers from it), typically `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 `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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.

- `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 `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 `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 not must be declared elsewhere, and it can be used only after expanding the macro.

- `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 `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 not must 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 `fixed` (to be able to generate random numbers from it), typically `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 `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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.

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

  keyseed is the type of key seeds, must be `fixed` (to be able to generate random numbers from it), typically `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 `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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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 `fixed` and `large`, of cardinal a prime q).

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

  g: a generator of the group.

  exp: the exponentiation function.

  mult: the multiplication function for exponents, product modulo q in $\{1, \ldots, q-1\}$, 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 not must be declared elsewhere, and they can be used only after expanding the macro.

- `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*.

- `expand Xor(`D, xor`).` defines the function symbol xor to be exclusive or on the set of bitstrings D.

  The type D must be declared before this macro is expanded. The function xor is declared by this macro. It not must be declared elsewhere, and it can be used only after expanding the macro.

  This model of xor is not fully satisfactory. In particular, associativity is missing (but it is obviously still sound).

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 `fixed` 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.
- `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` ...: 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` f `*`: apply a cryptographic transformation determined by the symbol f. f 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` $f\ x_1\ \dots\ x_n$: 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.)
- `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.
- `auto`: switch to automatic mode; try to terminate the proof automatically from the current game.
- `set` ⟨parameter⟩ `=` ⟨value⟩: sets parameters, as the `set` instruction in input files.
- `allowed_collisions` ⟨formulas⟩: determine when to eliminate collisions. ⟨formulas⟩ is a list of comma separated formulas of the form ⟨psize⟩$_1$`^`n_1`*`...`*`⟨psize⟩$_k$`^`n_k/⟨tsize⟩, where the exponents n_i can be omitted when equal to 1; ⟨psize⟩$_i$ 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; ⟨tsize⟩ 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 $\mathit{constant} \times p_1^{n_1} \times \cdots \times p_k^{n_k}/|T|$, where p_i is a parameter of size at most ⟨psize⟩$_i$ and T is a type of size at least ⟨tsize⟩. By default, collisions are eliminated for $\mathit{anything}/|T|$ when T is a `large` type, and for $p/|T|$ when p is `small` and T is a `password` type.

  Additionally, ⟨formulas⟩ may also contain elements of the form `collision *`⟨psize⟩$_1$`^`n_1`*`...`*`⟨psize⟩$_k$`^`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 ⟨psize⟩$_i$. 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 a succinct description of the performed transformation, and the formula giving the difference of probability between these games (if it is not 0). 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`,(`$p_1, \ldots, p_n$`))`	$c(p_1, \ldots, p_n)$
`out(`c`,`M`)`	$\overline{c}\langle M\rangle$
`out(`c`,(`$M_1, \ldots, M_n$`))`	$\overline{c}\langle M_1, \ldots, M_n\rangle$
`!`N	$!^N$
`yield`	$\overline{0}$
`->`	$\rightarrow$
For the `oracles` front-end	
`<-`	$\leftarrow$
`<-R`	$\stackrel{R}{\leftarrow}$

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 on 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.