CryptoVerif
Computationally Sound, Automatic Cryptographic Protocol Verifier
User Manual

Bruno Blanchet and David Cadé
INRIA Paris, France

June 27, 2022

Contents

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 [3, 2, 4, 5] that can be downloaded at
`https://bblanche.gitlabpages.inria.fr/publications/index.html`.

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

2 Command Line

The syntax of the command line is as follows:

./cryptoverif [*options*] ⟨filename⟩

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

- -in ⟨frontend⟩: Chooses the frontend to use by CryptoVerif. ⟨frontend⟩ can be either channels (the default) or oracles. The channels frontend uses a calculus inspired by the pi calculus, described in Section 3 and in [3, 2]. The oracles frontend uses a calculus closer to cryptographic games, described in Section 4 and in [4, 5]. By default, CryptoVerif uses the oracles frontend when the input ⟨filename⟩ ends with .ocv, and otherwise it uses the channels frontend.
- -lib ⟨filename⟩: Specifies a library file to be 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 extension .cvl or .ocvl may also be included in ⟨filename⟩.) Library files typically contain default declarations useful for several protocols.

  When no -lib option appears, CryptoVerif loads the default library, default.cvl in the channels front-end, default.ocvl in the oracles front-end. The default library is searched in the current directory, then in the directory that contains the executable cryptoverif.

  Multiple libraries can be specified by using -lib for each library. The libraries are loaded in the same order as they appear on the command line.
- -oproof ⟨filename⟩: Output the proof in the given file name, instead of displaying it on the standard output.
- -ocommands ⟨filename⟩: Output the interactive commands in the given file name. By default, this file is in the current directory. If command-line option -o is present, it is in the directory specified by that option.
- -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.
- -oequiv ⟨filename⟩: Append the generated special equivalences to the given file. (See equiv ... special.)
- -impl: Instead of proving the protocol, generate an implementation in OCaml corresponding to the modules defined in the input file.
- -o ⟨directory⟩: Outputs the files generated by out_game, out_state, out_facts, out_equiv, and out_commands in the given directory. If the -impl option is given, outputs the implementation files in the given directory.

Input files with a name that ends in .pcv are meant to be analyzed by both CryptoVerif and ProVerif. When CryptoVerif analyzes such a file, it first preprocesses it with m4 with CryptoVerif defined. Similarly, when ProVerif analyzes such a file, it first preprocesses it with m4 with ProVerif defined. That allows you to conditionally include parts of the file depending on whether CryptoVerif or ProVerif analyzes it.

3 channels Front-end

Comments can be included in input files. Comments are surrounded by (* and *). Nested comments are supported.

Identifiers (⟨ident⟩) 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: builtin, channel, collision, const, def, defined, do, else, eps_find, eps_rand, equation, equiv,

- $[M]$ means that M is optional; $(M)^*$ means that M occurs 0 or any number of times.
- $\mathrm{seq}\langle\mathrm{X}\rangle$ is a sequence of X: $\mathrm{seq}\langle\mathrm{X}\rangle = [(\langle\mathrm{X}\rangle\texttt{,})^*\langle\mathrm{X}\rangle] = \langle\mathrm{X}\rangle\texttt{,}\ldots\texttt{,}\langle\mathrm{X}\rangle$. (The sequence can be empty, it can be one element $\langle\mathrm{X}\rangle$, or it can be several elements $\langle\mathrm{X}\rangle$ separated by commas.)
- $\mathrm{seq}^+\langle\mathrm{X}\rangle$ is a non-empty sequence of X: $\mathrm{seq}^+\langle\mathrm{X}\rangle = (\langle\mathrm{X}\rangle\texttt{,})^*\langle\mathrm{X}\rangle = \langle\mathrm{X}\rangle\texttt{,}\ldots\texttt{,}\langle\mathrm{X}\rangle$. (It can be one or several elements of $\langle\mathrm{X}\rangle$ separated by commas.)

Figure 1: Grammar notations

`equivalence`, `event`, `event_abort`, `expand`, `find`, `forall`, `foreach`, `fun`, `get`, `if`, `implementation`, `in`, `inf`, `inj-event`, `insert`, `is-cst`, `length`, `let`, `letfun`, `letproba`, `max`, `maxlength`, `min`, `new`, `newChannel`, `number`, `optim-if`, `orfind`, `out`, `param`, `Pcoll1rand`, `Pcoll2rand`, `proba`, `process`, `proof`, `public_vars`, `query`, `query_equiv`, `return`, `secret`, `set`, `special`, `suchthat`, `table`, `then`, `time`, `type`, `yield`.

Strings ($\langle$string$\rangle$) start and end with `"`. Inside the string, `\"` stands for `"`, `\'` stands for `'`, `\n` for linefeed, `\t` for tab, `\b` for backspace, `\r` for carriage return, and `\\` for `\`. Other combinations with `\` are not allowed. Characters other than `"` and `\` stand for themselves.

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 may consist of a list of declarations followed by a process:

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

The process describes the considered security protocol; the declarations specify in particular hypotheses on the cryptographic primitives and the security properties to prove.

Alternatively, the input may also consist of a list of declarations followed by an equivalence query:

$$\langle\mathrm{declaration}\rangle^*\ \texttt{equivalence}\ \langle\mathrm{iprocess}\rangle\ \langle\mathrm{iprocess}\rangle\ [\texttt{public\_vars}\ \mathrm{seq}\langle\mathrm{ident}\rangle]$$

The query `equivalence` Q_1 Q_2 tells CryptoVerif to show that the processes (games) Q_1 and Q_2 are computationally indistinguishable. When it is present, the indication `public_vars` $x_1, \ldots, x_n$ means that the adversary has read access to the variables $x_1, \ldots, x_n$.

Finally, the input may also be:

$$\langle\mathrm{declaration}\rangle^*\ \texttt{query\_equiv}[\texttt{(}\langle\mathrm{ident}\rangle[\texttt{(}\langle\mathrm{ident}\rangle\texttt{)}]\texttt{)}]$$
$$\langle\mathrm{omode}\rangle\ [\texttt{|}\ \ldots\ \texttt{|}\langle\mathrm{omode}\rangle]\ \texttt{<=(?)=>}\ [\texttt{[}n\texttt{]}]\ [\texttt{[}\mathrm{seq}^+\langle\mathrm{option}\rangle\texttt{]}]\ \langle\mathrm{ogroup}\rangle\ [\texttt{|}\ \ldots\ \texttt{|}\langle\mathrm{ogroup}\rangle]$$

The keyword `query_equiv` is followed by an indistinguishability property specified in the same syntax as assumptions on security primitives (see the declaration `equiv`), except that the probability of distinguishing the two sides is replaced with `?`. CryptoVerif is going to bound this probability, so we do not need to give it.

- When the option `[computational]` is absent, CryptoVerif then converts this assumption into an `equivalence` between two processes and tries to prove it.
- When the option `[computational]` is present, CryptoVerif then converts this assumption into the unreachability of an event triggered when the oracles on the two sides return different results. The unreachability of this event implies that both sides are indistinguishable. In this case, the random values marked `[unchanged]` are shared between both sides, while the others are considered independent. In principle, any mapping from the random values of the left-hand side to the random values of the right-hand side could allow us to prove the desired indistinguishability property, as long as it preserves the probability distributions; however, CryptoVerif only supports the case in which some random values are equal on both sides and others are independent.

⟨identbound⟩ ::= [⟨ident⟩ =] ⟨ident⟩ <= ⟨ident⟩

⟨vartype⟩ ::= seq$^+$⟨ident⟩:⟨ident⟩

⟨vartypeb⟩ ::= seq$^+$⟨ident⟩:⟨ident⟩
| seq$^+$⟨ident⟩ <= ⟨ident⟩

⟨basicpat⟩ ::= ⟨ident⟩
| ⟨ident⟩:⟨ident⟩
| ⟨ident⟩ <= ⟨ident⟩

⟨simpleterm⟩ ::= ⟨ident⟩
| ⟨ident⟩(seq⟨simpleterm⟩)
| (seq⟨simpleterm⟩)
| ⟨ident⟩[seq⟨simpleterm⟩]
| ⟨simpleterm⟩ = ⟨simpleterm⟩
| ⟨simpleterm⟩ <> ⟨simpleterm⟩
| ⟨simpleterm⟩ || ⟨simpleterm⟩
| ⟨simpleterm⟩ && ⟨simpleterm⟩

⟨letterm⟩ ::= ... (as in ⟨simpleterm⟩ with ⟨letterm⟩ instead of ⟨simpleterm⟩)
| ⟨basicpat⟩ <- ⟨letterm⟩;⟨letterm⟩
| **let** ⟨basicpat⟩ = ⟨letterm⟩ **in** ⟨letterm⟩

⟨term⟩ ::= ... (as in ⟨simpleterm⟩ with ⟨term⟩ instead of ⟨simpleterm⟩)
| **new** ⟨ident⟩:⟨ident⟩;⟨term⟩
| ⟨ident⟩ <-R ⟨ident⟩;⟨term⟩
| ⟨basicpat⟩ <- ⟨term⟩;⟨term⟩
| **let** ⟨pattern⟩ = ⟨term⟩ **in** ⟨term⟩ [**else** ⟨term⟩]
| **if** ⟨cond⟩ **then** ⟨term⟩ **else** ⟨term⟩
| **find**[**[unique]**] ⟨tfindbranch⟩ (**orfind** ⟨tfindbranch⟩)* **else** ⟨term⟩
| **event** ⟨ident⟩[(seq⟨term⟩)]; ⟨term⟩
| **event_abort** ⟨ident⟩
| **insert** ⟨ident⟩(seq⟨term⟩); ⟨term⟩
| **get**[**[unique]**] ⟨ident⟩(seq⟨pattern⟩) [**suchthat** ⟨term⟩] **in** ⟨term⟩ **else** ⟨term⟩

⟨varref⟩ ::= ⟨ident⟩[seq⟨simpleterm⟩]
| ⟨ident⟩

⟨cond⟩ ::= **defined**(seq$^+$⟨varref⟩) [&& ⟨term⟩]
| ⟨term⟩

⟨tfindbranch⟩ ::= seq⟨identbound⟩ **suchthat** ⟨cond⟩ **then** ⟨term⟩

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

Figure 2: Grammar for terms and patterns

$\langle$event$\rangle$ ::= `event(`$\langle$ident$\rangle$[`(`seq$\langle$letterm$\rangle$`)`]`)`
| `inj-event(`$\langle$ident$\rangle$[`(`seq$\langle$letterm$\rangle$`)`]`)`
$\langle$queryterm$\rangle$ ::= `(`$\langle$queryterm$\rangle$`)`
| $\langle$queryterm$\rangle$ `&&` $\langle$queryterm$\rangle$
| $\langle$queryterm$\rangle$ `||` $\langle$queryterm$\rangle$
| $\langle$queryterm$\rangle$ `==>` $\langle$queryterm$\rangle$
| $\langle$event$\rangle$
| $\langle$basicpat$\rangle$ `<-` $\langle$letterm$\rangle$`;`$\langle$queryterm$\rangle$
| `let` $\langle$basicpat$\rangle$ `=` $\langle$letterm$\rangle$ `in` $\langle$queryterm$\rangle$
| $\langle$letterm$\rangle$
$\langle$query$\rangle$ ::= `secret` $\langle$ident$\rangle$ [`public_vars` seq$\langle$ident$\rangle$] [`[onesession]`]
| $\langle$queryterm$\rangle$ [`public_vars` seq$\langle$ident$\rangle$]

Figure 3: Grammar for queries

The goal of this query is to build modular proofs: we can prove a property using this query, and then use it as assumption in a subsequent proof by just copy-pasting it.

A library file (specified on the command-line by the `-lib` option) consists of a list of declarations. Notations are summarized in Figure 1 and various syntactic elements are described in Figures 2, 3, 4, 5, and 7.

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

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

Terms contain further constructs `<-R`, `<-`, `event`, `event_abort`, `if`, `find`, `let`, `new`, `insert`, and `get` 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`. The constructs `event` and `insert` are not allowed to occur in conditions of `find` or `get`. We refer the reader to the description of processes below for a fully detailed explanation.

- `new` x`:`T`;`M chooses a new random number in type T, stores it in x, and returns the result of M.

  x `<-R` T`;`M is equivalent to `new` x`:`T`;`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:

$$
\begin{array}{rcl}
\langle\text{proba}\rangle & ::= & \texttt{(}\langle\text{proba}\rangle\texttt{)} \\
& | & \langle\text{proba}\rangle\ \texttt{+}\ \langle\text{proba}\rangle \\
& | & \langle\text{proba}\rangle\ \texttt{-}\ \langle\text{proba}\rangle \\
& | & \langle\text{proba}\rangle\ \texttt{*}\ \langle\text{proba}\rangle \\
& | & \langle\text{proba}\rangle\ \texttt{/}\ \langle\text{proba}\rangle \\
& | & \langle\text{proba}\rangle\texttt{\^{}}\langle\text{int}\rangle \\
& | & \texttt{max(}\text{seq}^{+}\langle\text{proba}\rangle\texttt{)} \\
& | & \texttt{min(}\text{seq}^{+}\langle\text{proba}\rangle\texttt{)} \\
& | & \langle\text{ident}\rangle[\texttt{(}\text{seq}\langle\text{proba}\rangle\texttt{)}] \\
& | & \texttt{|}\langle\text{ident}\rangle\texttt{|} \\
& | & \texttt{maxlength(}\langle\text{simpleterm}\rangle\texttt{)} \\
& | & \texttt{length(}\langle\text{ident}\rangle[\texttt{,}\ \text{seq}^{+}\langle\text{proba}\rangle]\texttt{)} \\
& | & \texttt{length((}\text{seq}\langle\text{ident}\rangle\texttt{)}[\texttt{,}\ \text{seq}^{+}\langle\text{proba}\rangle]\texttt{)} \\
& | & n \\
& | & \#\langle\text{ident}\rangle \\
& | & \#(\langle\text{ident}\rangle\ \texttt{foreach}\ \text{seq}^{+}\langle\text{ident}\rangle) \\
& | & \texttt{eps\_find} \\
& | & \texttt{eps\_rand(}T\texttt{)} \\
& | & \texttt{Pcoll1rand(}T\texttt{)} \\
& | & \texttt{Pcoll2rand(}T\texttt{)} \\
& | & \texttt{time} \\
& | & \texttt{time(}\langle\text{ident}\rangle[\texttt{,}\ \text{seq}^{+}\langle\text{proba}\rangle]\texttt{)} \\
& | & \texttt{time(let}\ \langle\text{ident}\rangle[\texttt{,}\ \text{seq}^{+}\langle\text{proba}\rangle]\texttt{)} \\
& | & \texttt{time((}\text{seq}\langle\text{ident}\rangle\texttt{)}[\texttt{,}\ \text{seq}^{+}\langle\text{proba}\rangle]\texttt{)} \\
& | & \texttt{time(let (}\text{seq}\langle\text{ident}\rangle\texttt{)}[\texttt{,}\ \text{seq}^{+}\langle\text{proba}\rangle]\texttt{)} \\
& | & \texttt{time(=}\ \langle\text{ident}\rangle[\texttt{,}\ \text{seq}^{+}\langle\text{proba}\rangle]\texttt{)} \\
& | & \texttt{time(!)} \\
& | & \texttt{time(foreach)} \\
& | & \texttt{time([}n\texttt{])} \\
& | & \texttt{time(\&\&)} \\
& | & \texttt{time(||)} \\
& | & \texttt{time(new}\ \langle\text{ident}\rangle\texttt{)} \\
& | & \texttt{time(<-R}\ \langle\text{ident}\rangle\texttt{)} \\
& | & \texttt{time(newChannel)} \\
& | & \texttt{time(if)} \\
& | & \texttt{time(find}\ n\texttt{)} \\
& | & \texttt{time(out}\ [\texttt{[}\text{seq}^{+}\langle\text{ident}\rangle\texttt{]}]\langle\text{ident}\rangle[\texttt{,}\ \text{seq}^{+}\langle\text{proba}\rangle]\texttt{)} \\
& | & \texttt{time(in}\ n\texttt{)} \\
& | & \texttt{optim-if}\ \langle\text{optimcond}\rangle\ \texttt{then}\ \langle\text{proba}\rangle\ \texttt{else}\ \langle\text{proba}\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle\text{optimcond}\rangle & ::= & \texttt{(}\langle\text{optimcond}\rangle\texttt{)} \\
& | & \texttt{is-cst(}\langle\text{proba}\rangle\texttt{)} \\
& | & \langle\text{proba}\rangle\ \texttt{=}\ \langle\text{proba}\rangle \\
& | & \langle\text{proba}\rangle\ \texttt{<=}\ \langle\text{proba}\rangle \\
& | & \langle\text{proba}\rangle\ \texttt{>=}\ \langle\text{proba}\rangle \\
& | & \langle\text{proba}\rangle\ \texttt{<}\ \langle\text{proba}\rangle \\
& | & \langle\text{proba}\rangle\ \texttt{>}\ \langle\text{proba}\rangle \\
& | & \langle\text{optimcond}\rangle\ \texttt{\&\&}\ \langle\text{optimcond}\rangle \\
& | & \langle\text{optimcond}\rangle\ \texttt{||}\ \langle\text{optimcond}\rangle
\end{array}
$$

Figure 4: Grammar for probabilities

⟨repl⟩ ::= ![⟨ident⟩ <=] ⟨ident⟩
| foreach ⟨ident⟩ <= ⟨ident⟩ do
⟨res⟩ ::= new ⟨ident⟩:⟨ident⟩;
| ⟨ident⟩ <-R ⟨ident⟩;
⟨obody_equiv⟩ ::= (⟨obody_equiv⟩)
| event_abort ⟨ident⟩
| ⟨res⟩ ⟨obody_equiv⟩
| ⟨basicpat⟩ <- ⟨term⟩; ⟨obody_equiv⟩
| let ⟨pattern⟩ = ⟨term⟩ in ⟨obody_equiv⟩ [else ⟨obody_equiv⟩]
| if ⟨cond⟩ then ⟨obody_equiv⟩ else ⟨obody_equiv⟩
| find[[unique]] ⟨ffindbranch⟩ (orfind ⟨ffindbranch⟩)* else ⟨obody_equiv⟩
| insert ⟨ident⟩(seq⟨term⟩); ⟨obody_equiv⟩
| get[[unique]] ⟨ident⟩(seq⟨pattern⟩) [suchthat ⟨term⟩] in ⟨obody_equiv⟩ else ⟨obody_equiv⟩
| return(⟨term⟩)
⟨ffindbranch⟩ ::= seq⟨identbound⟩ suchthat ⟨cond⟩ then ⟨obody_equiv⟩
⟨ogroup⟩ ::= ⟨ident⟩(seq⟨vartypeb⟩) [[n]] [[useful_change]] := ⟨obody_equiv⟩
| [⟨repl⟩] ⟨res⟩* ⟨ogroup⟩
| [⟨repl⟩] ⟨res⟩* (⟨ogroup⟩ | ... | ⟨ogroup⟩)
⟨omode⟩ ::= ⟨ogroup⟩ [[exist]]
| ⟨ogroup⟩ [all]
⟨specialarg⟩ ::= ⟨ident⟩
| ⟨string⟩
| (seq⟨specialarg⟩)

Figure 5: Grammar for equivalences

⟨dim⟩ ::= time[^⟨int⟩]
| length[^⟨int⟩]
| number
| ⟨dim⟩ * ⟨dim⟩
| ⟨dim⟩ / ⟨dim⟩

⟨vardim⟩ ::= seq^+⟨ident⟩:⟨dim⟩

Figure 6: Grammar for dimensions

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

Figure 7: Grammar for processes (channels front-end)

- 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 [data]. 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 [data] 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).

- 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'.
- 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 $FB_j = u_{j1}$ = i_{j1} <= $n_{j1}, \ldots, u_{jm_j}$ = i_{jm_j} <= n_{jm_j} suchthat $cond_j$ then M_j evaluates the conditions $cond_j$ for each j and each value of $i_{j1}, \ldots, i_{jm_j}$ in $[1, n_{j1}] \times \ldots \times [1, n_{jm_j}]$. If none of these conditions is true, it returns the result of M. Otherwise, it chooses randomly with (almost) uniform probability one j and one value of $i_{j1}, \ldots, i_{jm_j}$ such that the corresponding condition is true, stores it in $u_{j1}, \ldots, u_{jm_j}$ and returns the result of M_j. See the explanation of the find process below for more details.
- event e($M_1, \ldots, M_n$);P executes the event e($M_1, \ldots, M_n$), then executes P. Events serve in recording the execution of certain parts of the program for using them in queries. The symbol e must have been declared by an event declaration.
- event_abort e executes event e and aborts the game. It is intended to be used in the right-hand side of the definitions of some cryptographic primitives. (See also the equiv declaration; events in the right-hand side can be used when the simulation of left-hand side by the right-hand side fails. CryptoVerif is going to find a bound for the probability that the event is executed and include it in the probability of success of an attack.)
- insert *tbl*($M_1, \ldots, M_n$);M inserts the tuples $(M_1, \ldots, M_n)$ in the table *tbl*, then returns the result of M. The table *tbl* must have been declared with the appropriate types using the table declaration.
- get *tbl*($p_1, \ldots, p_n$) suchthat M in M' else M'' tries to find an element of the table *tbl* that matches the patterns $p_1, \ldots, p_n$ and such that M is true. If it succeeds, it returns the result of M' with the variables of $p_1, \ldots, p_n$ bound to that element of the table. If several elements match, one of them is chosen randomly with (almost) uniform probability. If no element matches, it returns the result of M''.

  When suchthat M is omitted, it is equivalent to suchthat *true*.

  A variant of get is get[unique], which guarantees that at most one element of the table satisfies the condition, except in cases of negligible probability.

  Internally, get is converted into find by CryptoVerif.

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:

- *proc*($M_1, \ldots, M_n$) is replaced with $P\{M_1/x_1, \ldots, M_n/x_n\}$ when *proc* is declared by let *proc*(x_1 : $T_1, \ldots, x_n$: T_n) = P. where P is an input process. The terms $M_1, \ldots, M_n$ must contain only variables, replication indices, and function applications.

- `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 indices* at that point are $i_1, \dots, i_n$.

  `foreach` i `<=` N `do` Q is equivalent to `!`i `<=` N Q.
- 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:

- $proc(M_1, \dots, M_n)$ is replaced with `let` $x_1 = M_1$ `in` ... `let` $x_n = M_n$ `in` P when $proc$ is declared by `let` $proc(x_1 : T_1, \dots, x_n : T_n)$ `=` P`.` where P is an output process.
- `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` $e(M_1, \dots, M_n)$`;`P executes the event $e(M_1, \dots, M_n)$, then executes P. Events serve in recording the execution of certain parts of the program for using them in queries. The symbol e must have been declared by an `event` declaration.
- `event_abort` e executes event e and terminates the game. (Nothing can be executed after this instruction, neither by the protocol nor by the adversary.) The symbol e must have been declared by an `event` declaration, without any argument.
- `new` x`:`T`;`P or x `<-R` T`;`P chooses a new random number in type T, stores it in x, and executes P. T must be declared with option `fixed`, `bounded`, or `nonuniform`. Each such type T comes with an associated default probability distribution D_T; the random number is chosen according to that distribution. The time for generated random numbers in that distribution is bounded by `time(new` T`)` or equivalently `time(<-R` T`)`.
  - When the type T is `nonuniform`, the default probability distribution D_T for type T may be non-uniform. It is left unspecified. (Notice that random bitstrings with non-uniform distributions can also be obtained by applying a function to a random bitstring choosen uniformly among a finite set of bitstrings, chosen in another type.)
  - When the type T is `fixed`, it consists of the set of all bitstrings of a certain length n. Probabilistic Turing machines can return uniformly distributed random numbers in such types, in bounded time. If T is not marked `nonuniform`, the default probability distribution D_T for T is the uniform distribution.
  - For other `bounded` types T, probabilistic bounded-time Turing machines can choose random numbers with a distribution as close as we wish to uniform, but may not be able to produce exactly a uniform distribution. If T is not marked `nonuniform`, the default probability distribution D_T is such that its distance to the uniform distribution is at most `eps_rand(`T`)`. The distance between two probability distributions D_1 and D_2 for type T is

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

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

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

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

so

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

Therefore

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

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

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

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

  The pattern p can be:

  - x[`:`T] variable, possibly with its type. Matches any bitstring (in type T), and stores it in x.
  - f`(`$p_1, \ldots, p_n$`)` where the function symbol f is declared `[data]`. 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 `[data]` 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`.

- 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.

- `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 $FB_j = u_{j1}$ `=` i_{j1} `<=` $n_{j1}, \ldots, u_{jm_j}$ `=` i_{jm_j} `<=` n_{jm_j} `suchthat` $cond_j$ `then` P_j.

  A simple example is the following: `find` u `=` i `<=` n `suchthat defined(`$x[i]$`) &&` $x[i] = a$ `then` P' `else` P tries to find an index i such that $x[i]$ is defined and $x[i] = a$, and when such an i is found, it stores that i in u and executes P'; 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 `=` i_1 `<=` $n_1, \ldots, u_m$ `=` i_m `<=` n_m `suchthat defined(`$M_1, \ldots, M_l$`) &&` M `then` P' `else` P tries to find values of $i_1, \ldots, i_m$ for which $M_1, \ldots, M_l$ are defined and M is true.

In case of success, it stores the values of $i_1, \dots, i_m$ in $u_1, \dots, u_m$ 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 $FB_j = u_{j1}$ `=` i_{j1} `<=` $n_{j1}, \dots, u_{jm_j}$ `=` i_{jm_j} `<=` n_{jm_j} `suchthat defined(`$M_{j1}, \dots, M_{jl_j}$`) &&` M_j `then` P_j tries to find a branch j in $[1, m]$ such that there are values of $i_{j1}, \dots, i_{jm_j}$ for which $M_{j1}, \dots, M_{jl_j}$ are defined and M_j is true. In case of success, it stores the value of $i_{j1}, \dots, i_{jm_j}$ in $u_{j1}, \dots, u_{jm_j}$ and executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions $cond_j =$ `defined(`$M_{j1}, \dots, M_{jl_j}$`) &&` M_j for each j and each value of $i_{j1}, \dots, i_{jm_j}$ in $[1, n_{j1}] \times \dots \times [1, n_{jm_j}]$. If none of these conditions is `true`, it executes P. Otherwise, it chooses randomly with almost uniform probability[1] one j and one value of $i_{j1}, \dots, i_{jm_j}$ such that the corresponding condition is `true`, stores that value in $u_{j1}, \dots, iu_{jm_j}$ and executes P_j.

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

The variables $i_{j1}, \dots, i_{jm_j}$ are considered as replication indices, and are used in the `defined` condition and in M_j: they are temporary variables that are used as loop indices to look for indices that satisfy the desired conditions. Once suitable indices are found, their value is stored in $u_{j1}, \dots, u_{jm_j}$ and the `then` branch is executed using these variables. It is possible to make array accesses to $u_{j1}, \dots, u_{jm_j}$ (such as $u_{j1}[M_1, \dots, M_k]$) elsewhere in the game, which is not possible for $i_{j1}, \dots, i_{jm_j}$.

As an abbreviation, one may write $FB_j = u_{j1}$ `<=` $n_{j1}, \dots, u_{jm_j}$ `<=` n_{jm_j} `suchthat defined(`$M_{j1}, \dots, M_{jl_j}$`)` && M_j `then` P_j. In this case, the same identifier u_{jk} is used for both the variable and the associated replication index i_{jk}.

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

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

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

  When `else` P' is omitted, it is equivalent to `else yield`. When `suchthat` M is omitted, it is equivalent to `suchthat` *true*.

  A variant of `get` is `get[unique]`, which guarantees that at most one element of the table satisfies the condition, except in cases of negligible probability.

  Internally, `get` is converted into `find` by CryptoVerif.

- Finally, let us explain the output `out(`c`[`$M_1, \dots, M_l$`],`N`);`Q. A channel $c[M_1, \dots, M_l]$ consists of both a channel name c (declared by `channel` c) and a tuple of terms $M_1, \dots, M_l$. Terms $M_1, \dots, M_l$ are intuitively analogous to IP addresses and ports which are numbers that the adversary may guess. Two channels are equal when they have the same channel name and terms that

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

evaluate to the same bitstrings. A semantic configuration always consists of a single output process (the process currently being executed) and several input processes. When the output process executes `out(`c`[`$M_1, \dots, M_l$`],`N`);`Q, one looks for an input on the same channel in the available input processes. If no such input process is found, the process blocks. Otherwise, one such input process `in(`c`[`$M'_1, \dots, M'_l$`],`p`);`P is chosen randomly with (almost) uniform probability. The communication is then executed: the output message N is evaluated, its result is truncated to the maximum length of bitstrings on channel c, the obtained bitstring is matched against pattern p. Finally, the output process P that follows the input is executed. The input process Q that follows the output is stored in the available input processes for future execution.

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

In the game as given to CryptoVerif, the channel can be either $c[i_1, \dots, i_n]$ where $i_1, \dots, i_n$ are the current replication indices at the considered input or output, or just a channel name c, as an abbreviation for $c[i_1, \dots, i_n]$. 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 indices.

Note that the syntax requires an output to be followed by an input process, as in [8]. 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`, `<-R`, `<-`, `let`, `find`, `in`) under replications `!`i_1 `<=` N_1, ..., `!`i_n `<=` N_n, x has implicitly indices $i_1, \dots, i_n$: x stands for $x[i_1, \dots, i_n]$. Arrays allow us to have full access to the state of the process. Arrays can be read using `find`. Similarly, when x is used with $k < n$ indices the missing $n-k$ indices are implicit: $x[u_1, \dots, u_k]$ stands for $x[i_1, \dots, i_{n-k}, u_1, \dots, u_k]$ where $i_1, \dots, i_{n-k}$ must be the $n-k$ first replication indices both at the creation of x and at the usage $x[u_1, \dots, u_k]$. (So the usage and creation of x must be under the same $n-k$ top-most replications.)

In the initial game, several variables may be defined with the same name, but they are immediately renamed to different names, so that after renaming, each variable is defined once. When several variables are defined with the same name, they can be referenced only under their definition without explicit array indices, 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 indices. 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 indices the current replication indices at its definition.)
- x possibly with array indices inside the `defined` condition of a find.
- $x[M_1, \dots, M_n]$ in M in a find branch ... `suchthat defined(`$M'_1, \dots, M'_l$`) &&` M `then` ..., such that $x[M_1, \dots, M_n]$ is a subterm of $M'_1, \dots, M'_l$.
- $x[M_1, \dots, M_n]$ in P in a find branch u_1 `=` i_1 `<=` $n_1, \dots, u_m$ `=` i_m `<=` n_m `suchthat defined(`M'_1, $\dots, M'_l$`) &&` ... `then` P, such that $x[M_1, \dots, M_n] = M\{u_1/i_1, \dots, u_m/i_m\}$ and M is a subterm of $M'_1, \dots, M'_l$.
- $x[M_1, \dots, M_n]$ in M'' in a find branch u_1 `=` i_1 `<=` $n_1, \dots, u_m$ `=` i_m `<=` n_m `suchthat defined(`M'_1, $\dots, M'_l$`) &&` ... `then` M'', such that $x[M_1, \dots, M_n] = M\{u_1/i_1, \dots, u_m/i_m\}$ and M is a subterm of $M'_1, \dots, M'_l$.

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

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

In the first pass, it collects the type of each variable: when a variable x is defined with type T under replications `!`N_1, ..., `!`N_n, x has type $[1, N_1] \times \ldots \times [1, N_n] \to T$. When the type of x is not explicitly given in its declaration (in `<-` or 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 indices 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 or x `<-R` T, T must be declared with option `bounded` (or `fixed`). In `if` M `then ... else ...`, M must be of type `bool`. Similarly, for

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

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

    By default (`allowUndefinedVar = false`), variables in `defined` conditions must be defined somewhere in the game. The setting `allowUndefinedVar = true` allows `defined` conditions with variables that are defined nowhere. The corresponding branch of `find` is then removed immediately, since the `defined` condition does not hold. This setting is useful to parse intermediate games generated by CryptoVerif, because such impossible `defined` conditions may occur in these games.

  - `set allowUnprovedUnique = false.`
    `set allowUndefinedVar = true.`

    By default (`allowUnprovedUnique = false`), CryptoVerif tries to prove that the `[unique]` annotations of `find` and `get` in the initial game are correct, and it is an error if it fails. With `allowUndefinedVar = true`, a warning is emitted if this proof fails, and CryptoVerif continues trying to prove the protocol. It is then the responsability of the user to make sure that the `[unique]` annotations are correct. This setting is not recommended as it may lead to unsound results in case the `[unique]` annotations are incorrect.

  - `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` or `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 autoMergeBranches = true.`
  `set autoMergeBranches = false.`

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

- `set autoMergeArrays = true.`
  `set autoMergeArrays = false.`

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

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

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

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

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

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

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

- `set inferUnique = false.`
  `set inferUnique = true.`

  When `inferUnique = true`, CryptoVerif tries to infer that a `find` that is not explicitly tagged `[unique]` is in fact unique, by showing that having several solutions for this `find` leads to a contradiction. When this proof succeeds, the `find` becomes `find[unique]`.

  When `inferUnique = false`, CryptoVerif does not try to make such proofs and just exploits explicit `[unique]` tags.

- `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 indices 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 autoRemoveAssignFindCond = true.`
  `set autoRemoveAssignFindCond = false.`

  When `true`, the default removal of assignments performed by CryptoVerif removes assignments on variables x defined by `let` $x = M$ `in` ... inside a condition of `find`. When `false`, the removal of this assignments is not performed automatically, but in manual proofs, it can be requested by the command `remove_assign findcond`.

- `set autoRemoveIfFindCond = true.`
  `set autoRemoveIfFindCond = false.`

  When `true`, simplification removes `if` in defined conditions of `find` by transforming them into logical formulae. When `false`, this removal is not performed.

- `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` or `<-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 autoExpand = true.`
  `set autoExpand = false.`

  When `true`, the transformation `expand` is automatically executed after transformations that result in a game containing `if`, `let`, `find`, `event`, `event_abort`, or `new` terms. The transformation `expand` expands these terms into processes. That leads to distinguishing the branches until the end of the process, which may help the proof by distinguishing more cases, but may lead to very large games. This is also needed because some game transformations of CryptoVerif do not support non-expanded games (`global_dep_anal`, `insert`, `merge_arrays`, `merge_branches`, `move`; furthermore, `simplify` is weaker when it is applied to a non-expanded game, and `success` fails to prove equivalence queries in non-expanded games and correspondence queries when the arguments of the considered events contain `if`, `let`, `find`, `event`, `event_abort`, or `new`).

  When `false`, the transformation `expand` is never automatically executed.

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

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

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

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

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

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

– `set noAdviceGlobalDepAnal = false.`
  `set noAdviceGlobalDepAnal = true.`

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

– `set simplifyAfterSARename = true.`
  `set simplifyAfterSARename = false.`

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

– `set backtrackOnCrypto = false.`
  `set backtrackOnCrypto = true.`

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

– `set useKnownEqualitiesInCryptoTransform = true.`
  `set useKnownEqualitiesInCryptoTransform = false.`

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

– `set priorityEventUnchangedRand =` n. (default: 5)

  During the cryptographic transformation, variables that occur in event and are mapped to random variables marked `[unchanged]` in the equivalence can be left unchanged.

  Sometimes, it is also possible to transform the term that contains them using one of the oracles of the equivalence.

  This settings determines which option is chosen: CryptoVerif prefers leaving the variable unchanged rather than using an oracle with priority at least n. It prefers using an oracle with priority less than n rather than leaving the variable unchanged.

– `set casesInCorresp = true.`
  `set casesInCorresp = false.`

  When `casesInCorresp = true`, CryptoVerif distinguishes cases depending on the definition point of variables, to infer more facts in order to prove correspondence properties. However, this can be slow in complex cases. Using `set casesInCorresp = false` disables this case distinction and speeds up the proof of correspondences.

– `set elsefindFactsInReplace = true.`
  `set elsefindFactsInReplace = false.`

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

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

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

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

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

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

- `set elsefindFactsInSuccess = true.`
  `set elsefindFactsInSuccess = false.`

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

- `set elsefindFactsInSuccessSimplify = true.`
  `set elsefindFactsInSuccessSimplify = false.`

  Similar to `elsefindFactsInReplace`, but applies in the elimination of useless code in `success simplify` operations.

- `set elsefindAdditionalDisjunct = true.`
  `set elsefindAdditionalDisjunct = false.`

  When `elsefindAdditionalDisjunct = true`, the procedure that infers facts from false conditions of `find` (see `set elsefindFactsInReplace`) is enriched: in case $y_j[M_{j1}, \dots, M_{jl_j}]$ may be defined at the same time as $x[M_1, \dots, M_l]$, we additionally assume that they have different indices, that is, $(M_{j1}, \dots, M_{jl_j}) \neq (M_1, \dots, M_l)$ to eliminate this case. Therefore, we infer $(M_{j1}, \dots, M_{jl_j}) \neq (M_1, \dots, M_l) \Rightarrow \mathtt{not}(t)$ or equivalently $(M_{j1}, \dots, M_{jl_j}) = (M_1, \dots, M_l) \vee \mathtt{not}(t)$. This is typically more costly and more precise than the basic procedure that just infers $\mathtt{not}(t)$ when possible.

- `set improvedFactCollection = false.`
  `set improvedFactCollection = true.`

  When `improvedFactCollection = true`, and CryptoVerif collects the facts that hold at each program point, it also takes into account variables that cannot be defined at a certain program point, variables that cannot be simultaneously defined, and elsefind facts, in order to prove more facts.

  It is a bit costly, so it is disabled by default (`improvedFactCollection = false`).

- `set useEqualitiesInSimplifyingFacts = false.`
  `set useEqualitiesInSimplifyingFacts = true.`

  When `useEqualitiesInSimplifyingFacts = true`, CryptoVerif uses known equalities between terms to determine whether a fact is equal to another fact.

  It is a bit costly, so it is disabled by default (`useEqualitiesInSimplifyingFacts = false`).

- `set useKnownEqualitiesWithFunctionsInMatching = false.`
  `set useKnownEqualitiesWithFunctionsInMatching = true.`

  When `useKnownEqualitiesWithFunctionsInMatching = true`, CryptoVerif uses known equalities $M_1 = M_2$ where the root of M_1 is a function application to normalize terms before testing whether they match an equation or collision statement or an oracle in a cryptographic transformation. That can allow to apply these statements or transformations more often.

  It is a bit costly, so it is disabled by default (`useKnownEqualitiesWithFunctionsInMatching = false`).

- `set ignoreSmallTimes =` $\langle$n$\rangle$. (default 3)

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

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

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

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

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

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

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

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

- `set maxTryNoVarDepth =` ⟨n⟩. (default 20)

  Sets the maximum depth of recursion in the replacement of variables with their values. When $n \leq 0$, puts no limit on this depth of recursion. Putting a limit avoids an infinite loop in some rare cases.

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

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

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

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

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

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

- `set maxElsefind =` n. (default `50`)

  Maximum of facts guaranteed in `else` branches of `find` collected from a single term.

- `set minAutoCollElim =` ⟨s⟩. (default `pest80`)

  Sets the maximum probability for which elimination of collisions is possible automatically (which corresponds to a minimum cardinal for the type, when the probability distribution is uniform). The argument ⟨s⟩ can be `large` (probability 2^{-160}), `password` (probability 2^{-20}), or `pest`n (probability 2^{-n}; see also the `type` declaration).

- `set maxGuess =` ⟨s⟩. (default `size40`)

  Sets the maximum size for which we can guess the value of a certain type. The argument ⟨s⟩ can be `default` or `passive` (size 2^{30}), `small` (size 2^{2}), or `size`n (size 2^{n}).

- `set forgetOldGames = false.`
  `set forgetOldGames = true.`

  When `forgetOldGames = true`, old games are removed from memory after each cryptographic transformation or each interactive command. That allows to save some memory, but prevents `undo`. The display of the games is saved into a temporary file to allow displaying the games at the end of the proof.

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]` | `[default]` | `[small]` | `[size`n`]`]`.`

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

  The options `[noninteractive]`, `[passive]`, `[default]`, `[small]`, or `[size`n`]` indicate to CryptoVerif an order of magnitude of the parameter. The option `[size`n`]` (where n is a constant integer) indicates the parameter is at most 2^n. CryptoVerif uses this information to optimize the computed probability bounds: when several bounds are correct, it chooses the smallest one. It also uses it to estimate the probability of collisions, and decide whether to eliminate the collision or not.

  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 `[size80]`.

  The absence of option, the option `[default]`, and the option `[passive]` correspond to adversary interacting with the tested protocol without any limitation on the number of sessions. This can correspond to two situations:

  - The protocol can start new sessions without limit even if it could detect that an active attack happened in previous sessions.
  - The adversary listens passively to sessions of the protocol that run as expected (hence the word `[passive]`). Therefore, for such runs, the adversary is undetected.

  No option, `[default]`, and `[passive]` are equivalent to `[size30]`.

  The option `[small]` should be used for sessions in which the adversary actively interacts with the honest participants and mounts detectable attacks, when these participants stop after a certain number of failed attempts (e.g. credit cards are blocked after 3 incorrect PINs). `[small]` is equivalent to `[size2]`.

- `proba` ⟨ident⟩[`(`seq⟨dim⟩`)`] [`[`⟨pest⟩`]`]`.`

  `proba` p`(`$d_1, \dots, d_n$`).` declares a probability function p taking n arguments of dimensions $d_1, \dots, d_n$ respectively. The syntax of dimensions is given in Figure 6, where `*`, `/`, and `^` are the usual product, division, and exponentiation. After reduction, dimensions are of the form $\texttt{time}^t \times \texttt{length}^l$, where t and l are integers. The dimension `number` corresponds to $\texttt{time}^0 \times \texttt{length}^0$.

  `proba` p`.` declares a probability function p taking any arguments. In this case, CryptoVerif checks that the number and dimensions of the arguments of p are compatible across calls to p.

  When `[`⟨pest⟩`]` (Probability ESTimate) is present, it gives an estimate of the value of the probability: `pest`n, where n is an integer, means that the probability is at most 2^{-n}; `password` is equivalent to `pest20`, i.e. probability at most 2^{-20}; `large` is equivalent to `pest160`, i.e. probability at most 2^{-160}. When `[`⟨pest⟩`]` is absent, `large` is the default. When the probability p appears in a `collision` statement and the command `allowed_collisions pest`n' has been issued, CryptoVerif applies the `collision` statement only when the probability of collision (taking into account how many times it is applied) is less than $2^{-n'}$. The estimate is only used to decide whether to eliminate collisions or not. The probability formula output by CryptoVerif at the end of the proof remains correct even if the estimates are incorrect. However, incorrect estimates may have the consequence that, when evaluating this probability, its value is larger than desired.

- `letproba` ⟨ident⟩[`(`seq$^+$⟨vardim⟩`)`] `=` ⟨proba⟩`.`

  `letproba` p`(`x_1 `:` $d_1, \dots, x_n$ `:` d_n`) =` *prob*`.` declares a probability function p with n arguments x_i of dimention d_i, equal to the probability formula *prob*. See `proba` above for an explanation of dimensions. The formula *prob* must represent a probability. It may refer to $x_1, \dots, x_n$. It is instantiated with the appropriate values of $x_1, \dots, x_n$ every time the probability function p is applied.

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

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

  - `bounded` means that the type is a set of bitstrings of bounded length or perhaps $\perp$. In other words, the type is a finite subset of bitstrings plus $\perp$.
  - `fixed` means that the type is the set of all bitstrings of a certain length n. In particular, the type is a finite set, so `fixed` implies `bounded`.
  - `nonuniform` means that random numbers may be chosen in the type with a non-uniform distribution. (When `nonuniform` is absent, random numbers are chosen using a uniform distribution for `fixed` types, an almost uniform distribution for `bounded` types, and random values cannot be chosen among other types. Note that `fixed, nonuniform` and `bounded, nonuniform` are also allowed to have a non-uniform distribution on a `fixed` or `bounded` type.)
  - `size`n indicates the order of magnitude of the cardinal of the type: `size`n means that its cardinal is $|T| = 2^n$, where n is an integer (like the set of bitstrings of length n).

    `size`min`_`max means that $2^{min} \leq |T| \leq 2^{max}$, where min and max are integers.
  - `pcoll`n (Probability of COLLision) means that `Pcoll1rand`$(T) \leq 2^{-n}$, where n is an integer. (`Pcoll1rand`(T) is the probability of collision between a random element chosen according to the default probability distribution D_T for the considered type T, and an independent element of type T.)

    When the default distribution is uniform or almost uniform (`fixed` and `bounded` types), `Pcoll1rand`$(T) = \frac{1}{|T|}$, so CryptoVerif estimates the probability of collision from the cardinal of the type and conversely, so mentioning one of `size`n or `pcoll`n is sufficient.

    CryptoVerif uses this information to determine whether collisions with random elements of the considered type T should be eliminated. For collisions to be eliminated, two conditions must be satisfied:

    1. `Pcoll1rand`$(T) \leq 2^{-n'}$, that is, T has option `pcoll`n with $n \geq n'$, where n' is set by `set minAutoCollElim = pest`n' (the default is $n' = 80$), or elimination of collisions on this data has been manually requested by the command `simplify coll_elim(`...`)` or `global_dep_anal` x `coll_elim(`...`)`.
    2. the probability of collision satisfies the conditions specified by the command `allowed_collisions` (used inside a `proof` environment). By default, collisions are eliminated when
       * either `Pcoll1rand`$(T) \leq 2^{-160}$ (T has option `pcoll`n with $n \geq 160$ or option `large`)
       * or `Pcoll1rand`$(T) \leq 2^{-20}$ (T has option `pcoll`n with $n \geq 20$ or option `password`), the collision is repeated at most N times, and N is a parameter of size at most 2.

       See the command `allowed_collisions` for more details.
  - `large` is equivalent to `size160_1000000000, pcoll160`, that is, $|T| \geq 2^{160}$ and `Pcoll1rand`$(T) \leq 2^{-160}$. By default, `large` means that the type T is large enough so that all collisions with random elements of T can be eliminated. (In asymptotic analyses, `Pcoll1rand`(T) is negligible. In exact security analyses, the probability of a collision is correctly expressed by the system.)
  - `password` is equivalent to `size20_40, pcoll20`, that is, $2^{20} \leq |T| \leq 2^{40}$ and `Pcoll1rand`$(T) \leq 2^{-20}$. `password` is intended for passwords in password-based security protocols. These passwords are taken in a dictionary whose size is much smaller than the size of a nonce for instance, so the probability of collisions among passwords is larger than among data of `large` types. CryptoVerif assumes that passwords are taken in a dictionary of between about one million (2^{20}) and about one trillion (2^{40}) elements.
  - `small` is equivalent to `size0_2`, that is, $|T| \leq 2^2$. By default, such a type is small enough so that its value can be guessed by the `guess` command.

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

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

  - `[data]` means that f is injective and that its inverses can be computed in polynomial time: $f(x_1, \dots, x_m) = y$ implies for $i \in \{1, \dots, m\}$, $x_i = f_i^{-1}(y)$ for some functions f_i^{-1}. (In the vocabulary of [2], f is poly-injective.) f can then be used for pattern matching.
  - `[projection]` 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 `projection` functions, by just declaring the corresponding poly-injective function `data`.)
  - `[uniform]` means that f maps the default distribution of its argument into the default distribution of its result. f must be unary; the argument and the result of f must be of types marked `fixed`, `bounded`, or `nonuniform`.

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

  `letfun` f`(`x_1`:`$T_1, \dots, x_n$`:`T_n`)=`M`.` declares a function f that takes n arguments named $x_1, \dots, x_n$ of types $T_1, \dots, T_n$, respectively. The subsequent calls to this function are replaced by the term M in which we replace $x_1, \dots, x_n$ with the arguments given by the caller. (We use x_i`<=`N_i instead of x_i`:`T_i when x_i is of type $[1, N_i]$, where N_i is a parameter, declared by `param` N_i.)

  Variables defined inside `letfun` can be used in array references and in queries, provided the process after expansion of `letfun` satisfies the required conditions for that.

- `const` seq$^+$⟨ident⟩`:`⟨ident⟩`.`

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

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

  `table` *tbl*`(`$T_1, \dots, T_n$`).` declares the table *tbl*, whose elements are tuples of type $T_1, \dots, T_n$. Types T_i may be replaced with parameters N_i, to declare a table that contains a replication index of type $[1, N_i]$. Elements can be inserted in the table by `insert` *tbl*`(`$M_1, \dots, M_n$`)` and the table can be read using `get`.

- `channel` seq$^+$⟨ident⟩`.`

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

- `event` ⟨ident⟩`[(`seq⟨ident⟩`)]`.

  `event` e`(`$T_1, \dots, T_n$`).` declares an event e that takes arguments of types $T_1, \dots, T_n$. When there are no arguments, we can simply declare `event` e. Types T_i may be replaced with parameters N_i, to declare an event that takes as argument a replication index of type $[1, N_i]$.

- `let` ⟨ident⟩`[(`seq⟨vartypeb⟩`)] =` ⟨oprocess⟩`.`
  `let` ⟨ident⟩`[(`seq⟨vartypeb⟩`)] =` ⟨iprocess⟩`.`

  `let` *proc*$(x_1 : T_1, \dots, x_n : T_n)$ `=` P`.` says that *proc* takes n arguments, x_1 of type T_1, …, x_n of type T_n, and is equal to the process P. (We use x_i`<=`N_i instead of x_i`:`T_i when x_i is of type $[1, N_i]$, where N_i is a parameter, declared by `param` N_i.) When parsing a process, *proc*$(M_1, \dots, M_n)$ will be replaced with $P\{M_1/x_1, \dots, M_n/x_n\}$ when P is an input process. In this case, the terms $M_1, \dots, M_n$ must contain only variables, replication indices, and function applications and the variables $x_1, \dots, x_n$ cannot have array accesses. The process *proc*$(M_1, \dots, M_n)$ will be replaced with `let` $x_1 = M_1$ `in` … `let` $x_n = M_n$ `in` P when P is an output process.

- `equation` `[forall` seq⟨vartype⟩`;]`⟨letterm⟩ `[if` ⟨letterm⟩`]`.

  `equation forall` $x_1 : T_1, \dots, x_n : T_n$`;`$M$`.` says that for all values of $x_1, \dots, x_n$ in types $T_1, \dots, T_n$ respectively, M is true. The term M must be a simple term without array accesses. All bound variables $x_1, \dots, x_n$ must occur in M. 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. In this case, all bound variables $x_1, \dots, x_n$ must occur in M_1, so that the target term M_2 is entirely determined knowing the instance of M_1. 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.

Variables bound by assignments inside M are replaced by their value.

`equation forall` $x_1 : T_1, \dots, x_n : T_n$`;`$M$ `if` M'`.` says that for all values of $x_1, \dots, x_n$ in types $T_1, \dots, T_n$ respectively such that M' is true, we have that M is true. The terms M and M' must be simple terms without array accesses. CryptoVerif tries to prove the precondition M', and in case of success, rewrites terms as explained above.

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

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

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

- `collision` ⟨res⟩*`[`[`random_choices_may_be_equal`]`][`[`forall` seq⟨vartype⟩`;`]
  `return(`⟨letterm⟩`) <=(`⟨proba⟩`)=> return(`⟨letterm⟩`)`[`if` ⟨cond⟩]`.`

where

$$
\begin{array}{rl}
\langle\text{cond}\rangle ::= & (\langle\text{cond}\rangle) \\
| & \langle\text{letterm}\rangle \\
| & \langle\text{ident}\rangle\ \texttt{independent-of}\ \langle\text{ident}\rangle \\
| & \langle\text{cond}\rangle\ \texttt{\&\&}\ \langle\text{cond}\rangle \\
| & \langle\text{cond}\rangle\ \texttt{||}\ \langle\text{cond}\rangle \\
| & \langle\text{basicpat}\rangle\ \texttt{<-}\ \langle\text{letterm}\rangle\texttt{;}\langle\text{cond}\rangle \\
| & \texttt{let}\ \langle\text{basicpat}\rangle\ \texttt{=}\ \langle\text{letterm}\rangle\ \texttt{in}\ \langle\text{cond}\rangle
\end{array}
$$

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

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

`collision new` x_1`:`T_1`;`...`new` x_n`:`T_n`;forall` $y_1 : T'_1, \dots, y_m : T'_m$`;`
    `return(`M_1`) <=(`p`)=> return(`M_2`) if` c`.`
means that the previous property holds when the condition c is true, where c is built by conjunctions or disjunctions of simple terms and independence conditions "y_i `independent-of` x_j", where y_i is bound by `forall` and x_j is bound by `new`. (However, disjunctions cannot mix terms and independence conditions.)

The option `[random_choices_may_be_equal]`, when it is present, allows several random number generations among $x_1, \dots, x_n$ to be the same, instead of being independent. One can then group, in a single `collision` statement, situations in which $x_1, \dots, x_n$ are the same or they are independent. The indices of the variables corresponding to $x_1, \dots, x_n$ in the game are still made independent of $x_1, \dots, x_n$. Hence, there are two cases: either x_i is the same as x_j, or x_i and x_j are independent of each other. With the option `[random_choices_may_be_equal]`, the independence conditions can also be "x_i `independent-of` x_j", where x_i and x_j are both bound by `new`. This condition then means x_i and x_j are different random choices, so x_j is also independent of x_i.

Variables bound by assignments inside M_1, M_2, c are replaced by their value.

- `equiv`[`(`⟨ident⟩[`(`⟨ident⟩`)`]`)`]
  ⟨omode⟩ [`|` ... `|`⟨omode⟩] `<=(`⟨proba⟩`)=>` [`[`n`]`] [`[`seq$^+$⟨option⟩`]`] ⟨ogroup⟩ [`|` ... `|`⟨ogroup⟩]`.`

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

  L and R define sets of oracles. (They can be translated into processes as explained in [2].)

  - $O(x_1 : T_1, \dots, x_n : T_n)$ `:=` *FP* represents an oracle O that takes arguments $x_1, \dots, x_n$ of types $T_1, \dots, T_n$ respectively, and returns the result computed by *FP*. The oracle body *FP* is similar to term, but terminates with a `return` as shown in the grammar of ⟨obody_equiv⟩ (Figure 5).

– Optionally, in the left-hand side, an integer between brackets [n] ($n \geq 0$) can be added in the definition of an oracle, which becomes O($x_1 : T_1, \dots, x_n : T_n$) [$n$] := FP. 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.

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

– !i <= N new y_1:T'_1; ...new y_m:T'_m;(FG_1 | ... | FG_n) represents N copies of a process that picks fresh random numbers $y_1, \dots, y_m$ of types $T'_1, \dots, T'_m$ respectively, and makes available the functions described in $FG_1, \dots, FG_n$. Each copy has a different value of $i \in [1, N]$. The identifier i cannot be referred to explicitly in the process; it is used only implicitly as array index of variables defined under !i <= N. The replication !i <= N can be abbreviated !N.

  The replication !i <= N can be omitted only at the root of the equivalence, when it contains a single ⟨omode⟩ on the left-hand side, and a single ⟨ogroup⟩ on the right-hand side. CryptoVerif then automatically adds a replication internally, and adjusts the probability accordingly.

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].)

Optionally, an integer between brackets [n] ($n \geq 0$) can be added in an equivalence. This integer does not change the semantics of the equivalence, but is used for the proof strategy: CryptoVerif uses preferably the equivalences with the smallest integers n when several equivalences can be used. When no integer is mentioned, $n = 0$ is assumed, so the equivalence has the highest priority.

Two options can specified for an equivalence, in [seq$^+$⟨option⟩]:

– The manual option, 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 option, 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 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 are of the form O(...) := ⟨obody_equiv⟩.

– 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[[unique]]` $\ldots$
  `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
  $\ldots$ `else` FP'

  where $\widetilde{u_k}$ is a non-empty suffix of $u_1, \ldots, u_m$, at least one $\widetilde{u_k}$ for $1 \leq k \leq l$ is the whole sequence $u_1, \ldots, u_m$ optionally followed by a sequence of indices $\widetilde{u_0}$, and the implicit suffix of the current array indices 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 indices, but the first $n' < n$ indices are left implicit when they are equal to the current indices 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 indices 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 suffix of $\widetilde{u_k}$.

  When $\widetilde{u_0}$ is not empty, the `find` is automatically transformed into

  `find[[unique]]` $\ldots$
  `orfind` u_1 `<=` $N_1, \ldots, u_m$ `<=` $N_m, \widetilde{u'_0}$ `<=` $\widetilde{N'_0}$ `suchthat defined(`$z_1[\widetilde{u'_1}], \ldots, z_l[\widetilde{u'_l}]$`) &&`
  $(\widetilde{u'_0}, \widetilde{i}) = (\widetilde{u_0}, \widetilde{i})$ `&&` M' `then` FP''
  $\ldots$ `else` FP'

  where $\widetilde{u'_0}$ are fresh variables of the same type as $\widetilde{u_0}$ and $\widetilde{N'_0}$ are their bounds ($\widetilde{u'_0}$ `<=` $\widetilde{N'_0}$ abbreviates a sequence of possibly several inequalities), $u'_k = u_k\{\widetilde{u'_0}/\widetilde{u_0}\}$, $M' = M\{\widetilde{u'_0}/\widetilde{u_0}\}$, $FP'' = FP\{\widetilde{u'_0}/\widetilde{u_0}\}$, and $\widetilde{i}$ is the implicit suffix of the current array indices. After this transformation, we are in the situation above with an empty $\widetilde{u_0}$.

  In case a variable z_k is defined by a `find` in R, z_k is automatically renamed into a fresh variable z'_k at its definition, and z_k is defined by `let` $z_k = z'_k$ in the `then` branch of the `find` that defines z'_k. The array accesses to z_k are left unchanged. After this transformation, the variables z_k on which array accesses are performed are never defined by a `find` in R.

– In addition to making array accesses, a limited usage of indices is allowed in R. Precisely, the following sequences of indices are allowed:

  1. the current array indices, and any suffix thereof;
  2. the sequence of indices $u_1, \ldots, u_m$ defined by a `find` followed by the associated implicit suffix of the current array indices (see above), and any suffix thereof;
  3. indices received as argument by the oracle, when a variable in L has these indices.

  When such a sequence of indices contains a single element, it is represented by the index itself. When it contains several elements, it is represented as a tuple `(...)` containing the indices. Such sequences of indices can be stored in variables (using `let`), and the sequences or variables containing them can be compared using equality `=` or disequality `<>`. In such comparisons, the types of the indices inside the sequences must be the same on both sides of the comparison. No other operation on indices is allowed, to make sure that the result is independent of the numbering of the oracle calls.

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.

- `equiv[(`⟨ident⟩`[(`⟨ident⟩`)])] special` ⟨ident⟩`(seq`⟨specialarg⟩`) [[manual]` | `[`n`]].`

`equiv(`*name*`) special` *specialname*`(`$a_1, \dots a_n$`)` declares an equivalence (that is, indistinguishability) between two games, like the previous version of `equiv`. However, instead of using games given explicitly, CryptoVerif generates the games from *specialname*`(`$a_1, \dots a_n$`)`.

The following values of *specialname* are supported: `rom` and `rom_partial` for random oracles, `prf` and `prf_partial` for pseudo-random functions, `prp` and `prp_partial` for pseudo-random permutations, `sprp` and `sprp_partial` for super pseudo-random permutations (pseudo-random permutations whose inverse is also a pseudo-random permutations), `icm` and `icm_partial` for the ideal cipher model.

Let us first explain the cases `rom`, `rom_partial`, `prf`, `prf_partial`, `prp`, and `prp_partial`. They take the following arguments seq⟨specialarg⟩ $= a_1, \dots a_n$:

1. A string *key_pos*, which can be `"key_first"` when the key is the first argument of the considered function, `"key_last"` when it is the last argument, or `"key` n`"` when it is its n-th argument. (n is an integer between 1 and the number of arguments of f.)
2. An identifier f, the considered function. The function f must be declared before the `equiv` declaration. For `prp` and `prp_partial`, the function f must take one argument in addition to the key, the type T of this argument must be the same as the type of the result of f, and it must be large enough so that collisions between a random element of the domain and an independent value can be eliminated (because we model a PRF and apply the PRF/PRP switching lemma), that is, `Pcoll1rand`$(T) \leq 2^{-n'}$, that is, T has option `pcoll`n with $n \geq n'$ where n' is set by `set minAutoCollElim` $=$ `pest`n'; the default is $n' = 80$. For other values of *specialname*, the function f must take at least one argument in addition to the key. In all cases, we must be able to choose an element randomly in the type of the key and in the type of the result of f, that is, these types must be declared `fixed`, `bounded`, or `nonuniform`.
3. When *specialname* is not `rom` nor `rom_partial`, an identifier p such that $p(t, N, l_1, \dots, l_m)$ is the probability that an adversary breaks the PRF (resp. PRP) assumption in time t, with at most N queries to the function f, with arguments of lengths at most $l_1, \dots, l_m$. The length is omitted when the corresponding type is bounded. The identifier p must be declared with `proba` p. This argument is omitted for random oracles because the probability is always 0.
4. A tuple of identifiers (k, r, x, y, z, u) for ROM and PRF, (k, r, x, u) for PRP, which are used to determine identifiers of variables in the generated equivalence:
   - k is the identifier of the key;
   - r is the identifier of the random result of f after game transformation;
   - x is the identifier used for arguments of f in most oracles;
   - y and z are the identifiers used for arguments of the two calls to f in oracles generated by the collision LHS `"`*Ocoll* : `new` r_1`:`T`;new` r_2`:`T`;forall` $a_1 : T_1, \dots, a_n : T_n; M$`"` (see the next argument);
   - u is the identifier used for indices of `find`.

   The identifiers x, y, z, u are suffixed by `_` and the name of the oracle in which they are used. The identifier r is suffixed by `_` and the suffix of the name of the oracle in which it is used. Moreover, if needed to avoid name clashes or to generate several variables, a suffix $\_n$ may be added to these identifiers or modified if they already have one. Using identifiers not used elsewhere allows the user to have stable identifiers in the generated equivalence.
5. A tuple of strings *collisions_LHS*, which can be either `("large")` or a tuple of strings of the following forms:
   - `"`*Ocoll* : `forall` $a_1 : T_1, \dots, a_n : T_n$`;new` r_1`:`$T; M$`"` where T is the type of the result of f and the simple term M uses the variables $a_1, \dots a_n, r_1$. In this case, CryptoVerif tries to simplify M assuming r_1 is a random value and $a_1, \dots, a_n$ do not depend on r_1. If it rewrites M into a term N that does not contain r_1, then it uses this information to transform terms $M\{f(\dots)/r_1\}$ into N when the result of $f(\dots)$ is a fresh random value, in the generated cryptographic transformation. (See files `examples/obasic/undeniable-sig.ocv` and `examples/obasic/undeniable-sig2.ocv` for examples.)

– "*Ocoll* : `new` r_1:T;`forall` $a_1 : T_1, \dots, a_n : T_n; M$" where T is the type of the result of f and the simple term M that uses the variables $a_1, \dots a_n, r_1$. In this case, CryptoVerif tries to simplify M assuming r_1 is a random value ($a_1, \dots, a_n$ may depend on r_1). If it rewrites M into a term N that does not contain r_1, then it uses this information to transform terms $M\{f(\dots)/r_1\}$ into N, in the generated cryptographic transformation.

– "*Ocoll* : `new` r_1:T;`new` r_2:T;`forall` $a_1 : T_1, \dots, a_n : T_n; M$" where T is the type of the result of f and the simple term M that uses the variables $a_1, \dots a_n, r_1, r_2$. In this case, CryptoVerif tries to rewrite M assuming r_1 and r_2 are independent random values into a term N_2 that does not contain r_1 nor r_2, and to rewrite M assuming $r_1 = r_2$ is a random value into a term N_1 that does not contain r_1 nor r_2. If it succeeds, then it uses this information to transform terms $M\{f(args_1)/r_1, f(args_2)/r_2\}$ into `if` $args_1 = args_2$ `then` N_1 `else` N_2, in the generated cryptographic transformation.

Obviously, when $n = 0$, `forall` $a_1 : T_1, \dots, a_n : T_n$; is omitted. The identifiers *Ocoll* are used to form the oracle names in the generated equivalence (see below); they must not contain `_`, and must be different from `O` and pairwise distinct. Only the first form is allowed for `prp` and `prp_partial`. Even when a single string is present, the argument must be a tuple of strings, so this string must be between parentheses.

When *collisions_LHS* is `("large")`, the type T of the result of f must be large enough so that collisions between a random element of the domain and an independent value can be eliminated, that is, `Pcoll1rand`$(T) \leq 2^{-n'}$, that is, T has option `pcoll`n with $n \geq n'$ where n' is set by `set minAutoCollElim = pest`n'; the default is $n' = 80$. In this case, this is equivalent to *collisions_LHS* containing:

– "`Oeq: forall` $a_1 : T$;`new` r_1:T;$r_1 = a_1$". Assuming a_1 does not depend on r_1, $r_1 = a_1$ simplifies into `false`, so $f(\dots) = a_1$ is transformed into `false` in the generated cryptographic transformation, when the result of $f(\dots)$ is a fresh random value.

– When *specialname* is not `prp` nor `prp_partial`, "`Ocoll: new` r_1:T;`new` r_2:T;$r_1 = r_2$". The term $r_1 = r_2$ simplifies into `false` when r_1 and r_2 are independent random values and into `true` when $r_1 = r_2$, so $f(args_1) = f(args_2)$ is transformed into $args_1 = args_2$ in the generated cryptographic transformation.

The argument *collisions_LHS* can be overriden when the equivalence is used in a `crypto` command, by passing the desired *collisions_LHS* as special argument to the `crypto` command.

The last or the last two arguments may be omitted.

When *specialname* is `rom`, `prf`, or `prp`, the generated equivalence provides the following oracles:

– Oracle `O` evaluates f on its arguments in the left-hand side, and performs a lookup into previous arguments of `O` in the right-hand side: it returns the previous result when the current arguments are equal to previous arguments and otherwise it returns a fresh random value.

– For each element of *collisions_LHS*, oracle *Ocoll* evaluates M with r_i replaced with a call to f in left-hand side and uses the simplified form of M in the right-hand side.

When *specialname* is `rom_partial`, `prf_partial`, or `prp_partial`, the generated equivalence provides oracles named *Ocoll* for each element "*Ocoll* : `new` r_1:T;`forall` $a_1 : T_1, \dots, a_n : T_n; M$" or "*Ocoll* : `new` r_1:T;`new` r_2:T;`forall` $a_1 : T_1, \dots, a_n : T_n; M$" of *collisions_LHS*. These oracles act like the oracle of the same name when *specialname* is `rom` (resp. `prf`—there are no such oracles for `prp_partial`).

It also provides oracles named *prefix*_*suffix* where *prefix* is `O` or an identifier *Ocoll* from a collision "*Ocoll* : `forall` $a_1 : T_1, \dots, a_n : T_n$;`new` r_1:T;M" in *collisions_LHS* and *suffix* is arbitrary. These oracles act like the oracle *prefix* when *specialname* is `rom` (resp. `prf` or `prp`), except that:

– When *suffix* starts with `leave`, the right-hand side still uses calls to f; it does not replace them with fresh random values. However, it still performs look ups as needed to make sure that the returned result is coherent with results previously returned by other oracles.

– It uses a collision matrix to determine whether arguments of oracles with various suffixes are allowed to collide with non-negligible probability. By default, this collision matrix says that the arguments of two oracles are allowed to collide when they have the same suffix or when one of the suffixes starts with `leave`. A different collision matrix can be specified by passing a string as a special argument to the `crypto` command, which can be either `"no collisions"` or statements seq$^+\langle$suffix$\rangle$ `may collide with previous` seq$^+\langle$suffix$\rangle$ separated by semi-colons (`;`). `"no collisions"` says that the arguments of two oracle calls are never allowed to collide. $\mathit{suffix}_1, \dots, \mathit{suffix}_n$ `may collide with previous` $\mathit{suffix}'_1, \dots, \mathit{suffix}'_{n'}$ says that the arguments of oracles with suffix suffix_i ($i \in \{1, \dots, n\}$) are allowed to collide with arguments of previous calls to oracles with suffix suffix'_j ($j \in \{1, \dots, n'\}$). In the right-hand side, the oracles execute `event_abort ev_coll` when a disallowed collision happens. That avoids generating further code in this case, and thus may considerably reduce the size of the generated game after applying the cryptographic transformation. However, in case a disallowed collision actually happens with non-negligible probability, CryptoVerif will be unable to prove that event `ev_coll` does not happen, so the proof will fail.

The oracles *prefix*`_leave` are generated by default. The other oracles are generated on demand when they are present in the `terms:` information of the `crypto` command. Therefore, you must explicitly mention in the `terms:` information all occurrences of terms that should be transformed by an oracle different from *prefix*`_leave`. (See file `examples/arinc823/sharedkey/lemmaEnc_equiv_v2_optim.ocv` for an example with `prf_partial`.)

Let us now explain the cases `sprp`, `sprp_partial`, `icm`, and `icm_partial`. They take the following arguments seq$\langle$specialarg$\rangle = a_1, \dots a_n$:

1. A tuple of strings *arg_order*, which contains the strings `"msg"`, `"key"`, and for `icm` and `icm_partial`, `"local_key"`, in the order in which the encryption and decryption functions take their arguments. (For the ideal cipher model, the `"key"` is the key that models the choice of the encryption scheme, and the `"local_key"` is the key passed to each encryption and decryption.)
2. A identifier *enc* and an identifier *dec*, which are respectively the encryption and decryption functions. These functions must have the same type, and take arguments as specified by *arg_order*. We must be able to choose an element randomly in the type of the argument `"key"` and in the type of the result of *enc* and *dec*, that is, these types must be declared `fixed`, `bounded`, or `nonuniform`. The type of the argument `"msg"` must be the same as the type of the result of *enc* and *dec*. (*enc* and *dec* are permutations of this type.) This type T must be large enough so that collisions between a random element of this type and an independent value can be eliminated (because we model a PRF and apply the PRF/PRP switching lemma), that is, `Pcoll1rand`$(T) \leq 2^{-n'}$, that is, T has option `pcoll`n with $n \geq n'$ where n' is set by `set minAutoCollElim` $=$ `pest`n'; the default is $n' = 80$.
3. When *specialname* is `sprp` or `sprp_partial`, an identifier p such that $p(t, N, N', l, l')$ is the probability that an adversary breaks the SPRP assumption in time t, with at most N queries to the function *enc*, with messages of length at most l, and at most N' queries to the function *dec*, with ciphertexts of length at most l'. The lengths are omitted when the type is bounded. The identifier p must be declared with `proba` p. This argument is omitted for the ideal cipher model because the probability is always 0.
4. A tuple of identifiers (k, lk, m, c, u) for ICM, (k, m, c, u) for SPRP, which are used to determine identifiers of variables in the generated equivalence:
   - k is the identifier of the key;
   - lk is the identifier of the local key;
   - m is the identifier of cleartext messages;
   - c is the identifier of ciphertexts;
   - u is the identifier used for indices of `find`.

   The identifiers lk, m, c, u are suffixed by `_` and the name of the oracle in which they are used. Moreover, if needed to avoid name clashes or to generate several variables, a suffix $\_n$ may be added to these identifiers or modified if they already have one. Using identifiers not used elsewhere allows the user to have stable identifiers in the generated equivalence.

5. A tuple of strings *collisions_LHS*, which can be either `("large")` or a tuple of strings of the following form:

   $$\texttt{"}\mathit{Ocoll} : \texttt{forall}\ a_1 : T_1, \dots, a_n : T_n; \texttt{new}\ r_1 \texttt{:} T; M\texttt{"}$$

   where T is the type of the result of *enc* and the simple term M uses the variables $a_1, \dots a_n, r_1$. In this case, CryptoVerif tries to simplify M assuming r_1 is a random value and $a_1, \dots, a_n$ do not depend on r_1. If it rewrites M into a term N that does not contain r_1, then it uses this information to transform terms $M\{f(\dots)/r_1\}$ into N when the result of $f(\dots)$ is a fresh random value, in the generated cryptographic transformation. Obviously, when $n = 0$, `forall` $a_1 : T_1, \dots, a_n : T_n;$ is omitted. The identifiers *Ocoll* are used to form the oracle names in the generated equivalence (see below); they must not contain `_`, and must be different from `O` and pairwise distinct. Even when a single string is present, the argument must be a tuple of strings, so this string must be between parentheses.

   When *collisions_LHS* is `("large")`, this is equivalent to *collisions_LHS* containing:

   $$\texttt{"Oeq: forall}\ a_1 : T; \texttt{new}\ r_1 \texttt{:} T; r_1 = a_1\texttt{"}$$

   Assuming a_1 does not depend on r_1, $r_1 = a_1$ simplifies into `false`, so $f(\dots) = a_1$ is transformed into `false` in the generated cryptographic transformation, when the result of $f(\dots)$ is a fresh random value.

   The argument *collisions_LHS* can be overriden when the equivalence is used in a `crypto` command, by passing the desired *collisions_LHS* as special argument to the `crypto` command.

The last or the last two arguments may be omitted.

When *specialname* is `icm` or `sprp`, the generated equivalence provides the following oracles:

- Oracle `O_enc` evaluates *enc* on its arguments in the left-hand side, and performs a lookup into previous cleartexts (and local keys for `icm`) of calls to `O_enc` and `O_dec` in the right-hand side: it returns the previous ciphertext when the current arguments are equal to previous cleartexts (and local keys for `icm`) and otherwise it returns a fresh random value.
- Oracle `O_dec` evaluates *dec* on its arguments in the left-hand side, and performs a lookup into previous ciphertexts (and local keys for `icm`) of calls to `O_enc` and `O_dec` in the right-hand side: it returns the previous cleartext when the current arguments are equal to previous ciphertexts (and local keys for `icm`) and otherwise it returns a fresh random value.
- For each element of *collisions_LHS*, oracles *Ocoll_enc* and *Ocoll_dec* evaluate M with r_i replaced with a call to *enc* (resp. *dec*) in left-hand side and uses the simplified form of M in the right-hand side.

When *specialname* is `icm_partial` or `sprp_partial`, the generated equivalence provides oracles named *prefix*_*middle*_*suffix* where *prefix* is `O` or an identifier *Ocoll* from a collision in *collisions_LHS*, *middle* is `enc` or `dec`, and *suffix* is arbitrary. These oracles act like the oracle *prefix*_*middle* when *specialname* is `icm` (resp. `sprp`), except that it uses a collision matrix to determine whether arguments of oracles with various suffixes are allowed to collide with non-negligible probability. By default, this collision matrix says that the arguments of two oracles are allowed to collide when they have the same suffix or when one of the suffixes is `default`. A different collision matrix can be specified by passing a string as a special argument to the `crypto` command, which can be either `"no collisions"` or statements $\mathrm{seq}^+\langle\mathrm{suffix}\rangle$ `may collide with previous` $\mathrm{seq}^+\langle\mathrm{suffix}\rangle$ separated by semi-colons (`;`). `"no collisions"` says that the arguments of two oracle calls are never allowed to collide. $\mathit{suffix}_1, \dots, \mathit{suffix}_n$ `may collide with previous` $\mathit{suffix}'_1, \dots, \mathit{suffix}'_{n'}$ says that the arguments of oracles with suffix suffix_i ($i \in \{1, \dots, n\}$) are allowed to collide with arguments of previous calls to oracles with suffix suffix'_j ($j \in \{1, \dots, n'\}$). In the right-hand side, the oracles execute `event_abort ev_coll` when a disallowed collision happens. That avoids generating further code in this case, and thus may considerably reduce the size of the generated game after applying the cryptographic transformation. However, in case a disallowed collision actually happens with non-negligible probability, CryptoVerif will be unable to prove that event `ev_coll` does not happen, so the proof will fail.

The oracles with suffix `default` are generated by default. The other oracles are generated on demand when they are present in the `terms:` information of the `crypto` command. Therefore, you must explicitly mention in the `terms:` information all occurrences of terms that should be transformed by an oracle with a suffix different from `default`.

The `[manual]` option, when it is present in the declaration, prevents the automatic application of the transformation. The transformation is then applied only using the manual `crypto` command. Alternatively, an integer between brackets `[`n`]` ($n \geq 0$) can also be added to the declaration. This integer does not change the semantics of the equivalence, but is used for the proof strategy: CryptoVerif uses preferably the equivalences with the smallest integers n when several equivalences can be used. When no integer is mentioned, $n = 0$ is assumed, so the equivalence has the highest priority.

- `query` [seq⟨vartypeb⟩`;`]⟨query⟩(`;`⟨query⟩)*`.`

  The `query` declaration indicates which security properties we would like to prove. It is of the form `query` x_1`:`T_1`,` ... `,`x_n`:`T_n`;`Q_1`;` ... `;`Q_n. First, we declare the types of all variables $x_1, \dots, x_n$ that occur in correspondence queries that follow. (We use x_i`<=`N_i instead of x_i`:`T_i when x_i is of type $[1, N_i]$, where N_i is a parameter, declared by `param` N_i.) Second, we give the queries themselves. The available queries Q_i are as follows:

  – `secret` x [`public_vars` l]: show that the array x is indistinguishable from an array of independent random numbers (by several test queries), even when the variables in l are public. The list l is considered empty when it is omitted. In the vocabulary of [2], this is secrecy.

  – `secret` x [`public_vars` l] `[cv_onesession]`: show that any element of the array x cannot be distinguished from a random number (by a single test query), even when the variables in l are public. The list l is considered empty when it is omitted. In the vocabulary of [2], this is one-session secrecy.

    In addition to the option `cv_onesession`, the options `real_or_random`, `cv_real_or_random` and all options starting with `pv_` are also allowed, but ignored. Real-or-random secrecy is the default for CryptoVerif and the options starting with `pv_` are for ProVerif.

  – M_0 [`public_vars` l], where in M_0, `==>` is not allowed under `&&` or `||`, and the term before `==>` cannot contain `||`, so that after replacing variables bound by assignments by their value, M_0 is of form M `==>` M' or M, where M and M' do not contain `==>`. The query M is an abbreviation for M `==> false`, so we only need to consider M `==>` 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, even when the variables in l are public.

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

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

    When `inj-event` is present, the system proves an injective correspondence, that is, it shows that several different events marked `inj-event` before `==>` imply the execution of several different events marked `inj-event` after `==>`. More precisely, `inj-event(`$e_1(M_{11}, \dots, M_{1m_1})$`)` `&&` ... `&&` `inj-event(`$e_n(M_{n1}, \dots, M_{nm_n})$`)` `&&` ... `==>` M' means that for each tuple of executed events $e_1(M_{11}, \dots, M_{1m_1})$ (executed N_1 times), ..., $e_n(M_{n1}, \dots, M_{nm_n})$ (executed N_n times), M' holds, considering that an event `inj-event(`$e'(M_1, \dots, M_m)$`)` in M' holds when it has been executed at least $N_1 \times \dots \times N_n$ times. The `inj-event` 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"`.

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

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

- `expand` ⟨ident⟩`(`seq⟨ident⟩`).`

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

The following identifiers are predefined:

- The type `bitstring` is the type of all bitstrings.
- The type `bitstringbot` is the type that contains all bitstrings and $\perp$.
- 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 `+`, `-`, `*`, `/`, `^`. (`^` is the exponentiation, its second argument must be an integer; `^` has higher priority than `*` and `/`, which have higher priority than `+` and `-`, as usual), as well as the maximum, denoted `max(`p_1`,`...`,`p_n`)`, and minimum, denoted `min(`p_1`,`...`,`p_n`)`. They may also contain

- P or $P(p_1, \dots, p_n)$ where P has been declared by `proba` P and $p_1, \dots, p_n$ are probability formulas; this formula represents an unspecified probability depending on $p_1, \dots, p_n$.
- N, where N has been declared by `param` N, designates the number of copies of a replication.
- $\#O$, where O is an oracle, designates the number of different calls to the oracle O.
- $\#(O$ `foreach` $x)$, where O is an oracle and x is a random variable, designates the maximum number of different calls to the oracle O for each choice of the random variable x. The variable x must be chosen in a sequence of random variables at least one replication above the definition of oracle O.
- $\#(O$ `foreach` $i_1, \dots, i_n)$, where O is an oracle and $i_1, \dots, i_n$ are replication indices, designates the maximum number of different calls to the oracle O for each value of the replication indices $i_1, \dots, i_n$. These replication indices must be a strict suffix of the current replication indices at the definition of O. (i_n must be the index of the replication at the top of `equiv` statement, i_{n-1} must be a replication index of replication just under the one with index i_n, and so on.)
- $|T|$, where T has been declared by `type` T and is `fixed` or `bounded`, designates the cardinal of T.

- `maxlength(`M`)` is the maximum length of term M (M must be a simple term without array access, and must be of a non-bounded type).
- `length(`$f, p_1, \dots, p_n$`)` designates the maximal length of the result of a call to f, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of f (p_i must be built from `max`, `maxlength(`M`)`, and `length(`$f', \dots$`)`, where M is a term of the type of the corresponding argument of f and the result of f' is of the type of the corresponding argument of f).
- `length`(T) designates the maximal length of a bitstring of type T, where T is a bounded type.
- `length((`$T_1, \dots, T_n$`),`$p_1, \dots, p_n$`)` designates the maximal length of the result of the tuple function from $T_1 \times \ldots \times T_m$ to `bitstring`, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of this function.
- n is an integer constant.
- `eps_find` is 2 times the maximum distance between the uniform probability distribution and the probability distribution used for choosing elements in `find`.
- `eps_rand(`T`)` is the maximum distance between the uniform probability distribution and the default probability distribution D_T for type T (when T is `bounded`).
- `Pcoll1rand(`T`)` is the maximum probability of collision between a random value X of type T chosen according to the default distribution D_T for type T and an element of type T that does not depend on it (when T is `nonuniform`). This is also the maximum probability of choosing any given element of T in the default distribution for that type:

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

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

$$\frac{1}{|T|} \leq \texttt{Pcoll2rand(}T\texttt{)} = \sum_{a \in T} \Pr[X = a]^2 \leq \texttt{Pcoll1rand(}T\texttt{)}$$

  where X is chosen according to the default distribution D_T.
- `optim-if` *condition* `then` p_1 `else` p_2 evaluates to p_1 when the condition *condition* is proved to be true and to p_2 otherwise. Hence, the formula p_2 must always be a sound estimate, whether the condition is true or not (because it may happen that the condition is true and CryptoVerif does not manage to prove it). The formula p_1 is typically a better estimate valid when the condition holds. The grammar for the condition *condition* is defined in Figure 4. The condition `is-cst(`p`)` is true when p is a constant. The other conditions have their usual meaning.
- `time` designates the runtime of the environment (attacker).

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

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

- `time(let(`$T_1, \dots, T_m$`)`$, p_1, \dots, p_n$`)` designates the maximal runtime of one pattern matching with the tuple function from $T_1 \times \ldots \times T_m$ to `bitstring`, where $p_1, \dots, p_n$ represent the maximum length of the non-bounded arguments of this function.
- `time(=`$T[, p_1, p_2]$`)` designates the maximal runtime of one call to bitstring comparison function for bitstrings of type T, where p_1, p_2 represent the maximum length of the arguments of this function when T is non-bounded.
- `time(!)` or `time(foreach)` is the maximum time of an access to a replication index.
- `time([`n`])` is the maximum time of an array access with n indices.
- `time(&&)` is the maximum time of a boolean and.
- `time(||)` is the maximum time of a boolean or.
- `time(new` T`)` or `time(<-R` T`)` is the maximum time needed to choose a random number of type T according to the default distribution for type T.
- `time(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 indices to choose. (Essentially, the time to store the values of the indices in a list and part of the time needed to randomly choose an element of that list.)
- `time(out [`$T_1, \dots, T_m$`]`$T, p_1, \dots, p_n$`)` represents the time of an output in which the channel indices are of types $T_1, \dots, T_m$, the output bitstring is of type T, and the maximum length of the channel indices and the output bitstring is represented by $p_1, \dots, p_n$ when they are non-bounded.
- `time(in` n`)` is the maximum time to store an input in which the channel has n indices in the list of available inputs.

CryptoVerif checks the dimension of probability formulas.

4 `oracles` Front-end

The `oracles` front-end is similar to the `channels` with the following differences. The keyword `newChannel` is replaced with `newOracle`, `run` is a keyword, and `channel` and `out` are not keywords.

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

⟨declaration⟩* `process` ⟨odef⟩

⟨declaration⟩* `equivalence` ⟨odef⟩ ⟨odef⟩ [`public_vars` seq⟨ident⟩]

⟨declaration⟩* `query_equiv`[`(`⟨ident⟩[`(`⟨ident⟩`)`]`)`]
  ⟨omode⟩ [`|` ... `|`⟨omode⟩] `<=(?)=>` [`[`n`]`] [`[`seq$^+$⟨option⟩`]`] ⟨ogroup⟩ [`|` ... `|`⟨ogroup⟩]

The syntax of processes is given in Figure 8. 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.

The oracle definition `run` $proc(M_1, \dots, M_n)$ is replaced with $P\{M_1/x_1, \dots, M_n/x_n\}$ when $proc$ is declared by `let` $proc(x_1 : T_1, \dots, x_n : T_n)$ `=` P`.` where P is an oracle definition. The terms $M_1, \dots, M_n$ must contain only variables, replication indices, and function applications.

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

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

Figure 8: Grammar for processes (oracles front-end)

When an oracle O is defined under `foreach` i_1`<=`N_1, ..., `foreach` i_n`<=`N_n, it also implicitly defines $O[i_1, \dots, i_n]$.

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 oracle bodies:

- `run` $proc(M_1, \dots, M_n)$ is replaced with `let` x_1 `=` M_1 `in` ... `let` x_n `=` M_n `in` P when $proc$ is declared by `let` $proc(x_1 : T_1, \dots, x_n : T_n)$ `=` P`.` where P is an oracle body.
- `yield` terminates the oracle, returning control to the caller.
- `return(`$N_1, \dots, N_l$`);`Q terminates the oracle, returning the result of the terms $N_1, \dots, N_l$. Then, it makes available the oracles defined in Q.
- The other oracle bodies are similar to output processes in the `channels` front-end.

In `return(`$M_1, \dots, M_n$`)`, M_j must be of a bitstring type T_j for all $j \leq n$ and that return instruction is said to be of type $T_1 \times \dots \times T_n$. All return instructions in an oracle body P (excluding return instructions that occur in oracle definitions Q in processes of the form `return(`$M_1, \dots, M_n$`);`Q) must be of the same type, and that type is said to be the type of the oracle body P. For each oracle definition O`(`$p_1, \dots, p_m$`) :=` P under `foreach` i_1`<=`N_1, ..., `foreach` i_n`<=`N_n, the oracle O is said to be of type $[1, N_1] \times \dots \times [1, N_n] \to T'_1 \times \dots \times T'_m \to T_1 \times \dots \times T_n$ where p_j is of type T'_j for all $j \leq m$ and P is of type $T_1 \times \dots \times T_n$. 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 $Q \mid Q'$ (so that several definitions of the same oracle cannot be simultaneously available). An oracle O must not occur under a definition of the same oracle O in the syntax tree, and oracles defined in different branches of `if`, `find`, `let`, `get` must have compatible structures (oracles with the same name must occur at the same place in the sequence of possible oracle calls). The other constructs are typed as in the `channels` front-end.

The `channel` seq$^+\langle$ident$\rangle$`.` declaration is removed, since channels do not exist in the `oracles` front-end.

In probability formulas (Figure 4), `time(out ...)` and `time(in` n`)` are removed and `time(newChannel)` is replaced with `time(newOracle)`. `time(newOracle)` is the maximum time to create a new private oracle.

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

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

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

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.

Finally, both front-ends accept two syntaxes for replication, generation of random numbers, and assignments. However, the default syntax for the display differs:

display in `channels`	display in `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 assignment x:T `<-` M can be used only for 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(`*key*, *cleartext*, *ciphertext*, *enc*, *dec*, *injbot*, *Z*, *Penc*`).` defines a IND-CPA (indistinguishable under chosen plaintext attacks) probabilistic symmetric encryption scheme.

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  $enc(cleartext, key) : ciphertext$ is the encryption function. Internally, it generates random coins, so that it is probabilistic.

  $dec(ciphertext, key)$: `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 *key*, *cleartext*, *ciphertext* and the probability *Penc* must be declared before this macro is expanded. The functions *enc*, *dec*, *injbot*, and *Z* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

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

- `expand IND_CPA_sym_enc_all_args(`*key*, *cleartext*, *ciphertext*, *enc_seed*, *enc*, *enc_r*, enc_r', *dec*, *injbot*, *Z*, *Penc*`).` is similar to the above, with three additional arguments.

  enc_seed is the type of random coins for encryption, must be `bounded`.

  $enc\_r(cleartext, key, enc\_seed) : ciphertext$ is the encryption function that takes coins as argument (instead of generating them internally).

  enc_r' is the symbol that replaces enc_r after game transformation.

- `expand IND_CPA_sym_enc_nonce(`*key*, *cleartext*, *ciphertext*, *nonce*, *enc*, *dec*, *injbot*, *Z*, *Penc*`).` defines a IND-CPA (indistinguishable under chosen plaintext attacks) probabilistic symmetric encryption scheme using a nonce (which must have a different value in each call to encryption) instead of random coins generated by encryption.

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

nonce is the type of nonces.

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

$dec(ciphertext, key, nonce)$: `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 *key*, *cleartext*, *ciphertext*, *nonce* and the probability *Penc* must be declared before this macro is expanded. The functions *enc*, *dec*, *injbot*, and Z are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

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

- `expand IND_CPA_sym_enc_nonce_all_args(`*key*, *cleartext*, *ciphertext*, *nonce*, *enc*, enc', *dec*, *injbot*, Z, *Penc*`).` is similar to the above, with one additional argument: enc' is the symbol that replaces *enc* after game transformation.

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

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  $enc(cleartext, key) : ciphertext$ is the encryption function. Internally, it generates random coins, so that it is probabilistic.

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

  This macro defines the equivalences named `ind_cpa`(*enc*), `int_ctxt`(*enc*), and `int_ctxt_corrupt`(*enc*) for use in the `crypto` command (see Section 7). The first equivalence corresponds to the IND-CPA property, the last two to the INT-CTXT property. The equivalence `int_ctxt_corrupt`(*enc*) is used when the key may be corrupted. It is applied only manually. The equivalence `int_ctxt`(*enc*) should generally be applied before `ind_cpa`(*enc*), because `int_ctxt`(*enc*) eliminates the decryption oracle.

- `expand IND_CPA_INT_CTXT_sym_enc_all_args(`*key*, *cleartext*, *ciphertext*, *enc_seed*, *enc*, *enc_r*, *enc_r'*, *dec*, *injbot*, *Z*, *Penc*, *Pencctxt*`).` is similar to the above, with three additional arguments.

  enc_seed is the type of random coins for encryption, must be `bounded`.

  $enc\_r(cleartext, key, enc\_seed) : ciphertext$ is the encryption function that takes coins as argument (instead of generating them internally).

  enc_r' is the symbol that replaces enc_r after game transformation.

- `expand IND_CPA_INT_CTXT_sym_enc_nonce(`*key*, *cleartext*, *ciphertext*, *nonce*, *enc*, *dec*, *injbot*, *Z*, *Penc*, *Pencctxt*`).` defines a IND-CPA (indistinguishable under chosen plaintext attacks) and INT-CTXT (ciphertext integrity) probabilistic symmetric encryption scheme using a nonce (which must have a different value in each call to encryption) instead of random coins generated by encryption.

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  nonce is the type of nonces.

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

  $dec(ciphertext, key, nonce)$: `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 *key*, *cleartext*, *ciphertext*, *nonce* and the probabilities *Penc* and *Pencctxt* must be declared before this macro is expanded. The functions *enc*, *dec*, *injbot*, and *Z* are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalences named `ind_cpa`(*enc*), `int_ctxt`(*enc*), and `int_ctxt_corrupt`(*enc*) for use in the `crypto` command (see Section 7). The first equivalence corresponds to the IND-CPA property, the last two to the INT-CTXT property. The equivalence `int_ctxt_corrupt`(*enc*) is used when the key may be corrupted. It is applied only manually. The equivalence `int_ctxt`(*enc*) should generally be applied before `ind_cpa`(*enc*), because `int_ctxt`(*enc*) eliminates the decryption oracle.

- `expand IND_CPA_INT_CTXT_sym_enc_nonce_all_args(`*key*, *cleartext*, *ciphertext*, *nonce*, *enc*, *enc'*, *dec*, *injbot*, *Z*, *Penc*, *Pencctxt*`).` is similar to the above, with one additional argument: *enc'* is the symbol that replaces *enc* after game transformation.

- `expand AEAD(`*key*, *cleartext*, *ciphertext*, *add_data*, *enc*, *dec*, *injbot*, *Z*, *Penc*, *Pencctxt*`).` defines an authenticated encryption scheme with additional data.

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

add_data is the type of additional data.

$enc(cleartext, add\_data, key) : ciphertext$ is the encryption function. Internally, it generates random coins, so that it is probabilistic.

$dec(ciphertext, add\_data, key)$: `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.

$Pencctxt(t, N, N', l, l', ld, ld')$ 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', additional data for encryption of length at most ld, and additional data for decryption of length at most ld'.

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

This macro defines the equivalences named `ind_cpa`(enc), `int_ctxt`(enc), and `int_ctxt_corrupt`(enc) for use in the `crypto` command (see Section 7). The first equivalence corresponds to the IND-CPA property, the last two to the INT-CTXT property. The equivalence `int_ctxt_corrupt`(enc) is used when the key may be corrupted. It is applied only manually. The equivalence `int_ctxt`(enc) should generally be applied before `ind_cpa`(enc), because `int_ctxt`(enc) eliminates the decryption oracle.

- `expand AEAD_all_args(`key, $cleartext$, $ciphertext$, add_data, enc_seed, enc, enc_r, enc_r', dec, $injbot$, Z, $Penc$, $Pencctxt$`).` is similar to the above, with three additional arguments.

  enc_seed is the type of random coins for encryption, must be `bounded`.

  $enc\_r(cleartext, add\_data, key, enc\_seed) : ciphertext$ is the encryption function that takes coins as argument (instead of generating them internally).

  enc_r' is the symbol that replaces enc_r after game transformation.

- `expand AEAD_nonce(`key, $cleartext$, $ciphertext$, add_data, $nonce$, enc, dec, $injbot$, Z, $Penc$, $Pencctxt$`).` defines an authenticated encryption scheme with additional data, using a nonce that must have a different value in each call to encryption. A typical example is AES-GCM.

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of enc without mentioning the length of the key), typically `fixed` and `large`.

  $cleartext$ is the type of cleartexts.

  $ciphertext$ is the type of ciphertexts.

  add_data is the type of additional data.

  $nonce$ is the type of nonces.

  $enc(cleartext, add\_data, key, nonce) : ciphertext$ is the encryption function.

  $dec(ciphertext, add\_data, key, nonce)$: `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.

$Pencctxt(t, N, N', l, l', ld, ld')$ 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', additional data for encryption of length at most ld, and additional data for decryption of length at most ld'.

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

This macro defines the equivalences named `ind_cpa`(*enc*), `int_ctxt`(*enc*), and `int_ctxt_corrupt`(*enc*) for use in the `crypto` command (see Section 7). The first equivalence corresponds to the IND-CPA property, the last two to the INT-CTXT property. The equivalence `int_ctxt_corrupt`(*enc*) is used when the key may be corrupted. It is applied only manually. The equivalence `int_ctxt`(*enc*) should generally be applied before `ind_cpa`(*enc*), because `int_ctxt`(*enc*) eliminates the decryption oracle.

- `expand AEAD_nonce_all_args(`*key*, *cleartext*, *ciphertext*, *add_data*, *nonce*, *enc*, *enc'*, *dec*, *injbot*, Z, *Penc*, *Pencctxt*`).` is similar to the above with one additional argument.

  enc' is the symbol that replaces *enc* after game transformation.

- `expand INDdollar_CPA_sym_enc(`*key*, *cleartext*, *ciphertext*, *cipher_stream*, *enc*, *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*`).`
  `expand INDdollar_CPA_sym_enc_all_args(`*key*, *cleartext*, *ciphertext*, *enc_seed*, *cipher_stream*, *enc*, *enc_r*, *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*`).`
  `expand INDdollar_CPA_sym_enc_nonce(`*key*, *cleartext*, *ciphertext*, *nonce*, *cipher_stream*, *enc*, *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*`).`
  `expand INDdollar_CPA_INT_CTXT_sym_enc(`*key*, *cleartext*, *ciphertext*, *cipher_stream*, *enc*, *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*, *Pencctxt*`).`
  `expand INDdollar_CPA_INT_CTXT_sym_enc_all_args(`*key*, *cleartext*, *ciphertext*, *enc_seed*, *cipher_stream*, *enc*, *enc_r* *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*, *Pencctxt*`).`
  `expand INDdollar_CPA_INT_CTXT_sym_enc_nonce(`*key*, *cleartext*, *ciphertext*, *nonce*, *cipher_stream*, *enc*, *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*, *Pencctxt*`).`
  `expand AEAD_INDdollar_CPA(`*key*, *cleartext*, *ciphertext*, *add_data*, *cipher_stream*, *enc*, *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*, *Pencctxt*`).`
  `expand AEAD_INDdollar_CPA_all_args(`*key*, *cleartext*, *ciphertext*, *add_data*, *enc_seed*, *cipher_stream*, *enc*, *enc_r*, *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*, *Pencctxt*`).`
  `expand AEAD_INDdollar_CPA_nonce(`*key*, *cleartext*, *ciphertext*, *add_data*, *nonce*, *cipher_stream*, *enc*, *dec*, *injbot*, Z, *enc_len*, *truncate*, *Penc*, *Pencctxt*`).`
  define macros similar to the ones above, but with the IND$-CPA property instead of IND-CPA. IND$-CPA means that the length of the ciphertext only depends on the length of the cleartext, and that the ciphertext is indistinguishable from a random bitstring of the same length. In comparison with the previous macros, they do not have the primed encryption argument (*enc_r'* or *enc'*), so the `_all_args` variant disappears for encryptions with a nonce since *enc'* was the only additional argument. They additionally have the following arguments:

  cipher_stream is the type of unbounded streams (must be `nonuniform`).

  enc_len(*cleartext*) : *ciphertext* is a function that returns, for each bitstring x, a bitstring of the same length as the encryption of x, consisting only of zeroes.

  truncate(*cipher_stream*, *ciphertext*) : *ciphertext* is the function such that $truncate(s, x)$ is the truncation of s to the length of x, where s is a stream of unbounded length.

  The type *cipher_stream* must be declared before these macros are expanded. The functions *enc_len* and *truncate* are declared by these macros. They must not be declared elsewhere, and they can be used only after expanding one of the macros.

  These macros define the equivalence `inddollar_cpa`(*enc*) instead of `ind_cpa`(*enc*).

- `expand IND_CCA2_sym_enc(`*key*, *cleartext*, *ciphertext*, *enc*, *dec*, *injbot*, *Z*, *Penc*`).` defines a IND-CCA2 (indistinguishable under adaptive chosen ciphertext attacks) probabilistic symmetric encryption scheme.

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  $enc(cleartext, key)$: *ciphertext* is the encryption function. Internally, it generates random coins, so that it is probabilistic.

  $dec(ciphertext, key)$: `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.

  $Penc(t, N, Nu, N', l, l')$ is the probability of breaking the IND-CCA2 property in time t for one key, N encryption queries that are different in both sides of the IND-CCA2 equivalence, Nu encryption queries that are the same in both side of the IND-CCA2 equivalence, N' decryption queries with cleartexts of length at most l and ciphertexts of length at most l'.

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

  This macro defines the equivalences named `ind_cca2(`*enc*`)` and `ind_cca2_partial(`*enc*`)`, for use in the `crypto` command (see Section 7). While the equivalence `ind_cca2(`*enc*`)` replaces all cleartexts with zeroes, the equivalence `ind_cca2_partial(`*enc*`)` replaces only some of them with zeroes. The latter equivalence can be applied only manually. The user should map the occurrences of encryption that he wants to transform to oracle *Oenc*, the ones he wants to leave unchanged to oracle *Oenc_unchanged*, and the ones that have already been transformed by a previous application of this equivalence to oracle $Oenc\_unchanged'$.

- `expand IND_CCA2_sym_enc_all_args(`*key*, *cleartext*, *ciphertext*, *enc_seed*, *enc*, *enc_r*, enc_r', *dec*, dec', *injbot*, *Z*, *Penc*`).` is similar to the above, with four additional arguments.

  enc_seed is the type of random coins for encryption, must be `bounded`.

  $enc\_r(cleartext, key, enc\_seed)$: *ciphertext* is the encryption function that takes coins as argument (instead of generating them internally).

  enc_r' and dec' are the symbols that replace *enc_r* and *dec* respectively after game transformation.

- `expand INT_PTXT_sym_enc(`*key*, *cleartext*, *ciphertext*, *enc*, *dec*, *injbot*, *Pencptxt*`).` defines an INT-PTXT (plaintext integrity) probabilistic symmetric encryption scheme.

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  $enc(cleartext, key)$: *ciphertext* is the encryption function. Internally, it generates random coins, so that it is probabilistic.

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

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

$Pencptxt(t, N, N', Nu', l, l')$ is the probability of breaking the INT-PTXT property in time t for one key, N encryption queries, N' decryption queries that are modified by the transformation, and Nu' decryption queries that are left unchanged by the transformation, with cleartexts of length at most l and ciphertexts of length at most l'.

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

This macro defines the equivalences named `int_ptxt`(*enc*) and `int_ptxt_corrupt_partial`(*enc*), for use in the `crypto` command (see Section 7). While the equivalence `ind_ptxt`(*enc*) replaces all decryption with lookups in encryption queries, the equivalence `ind_ptxt_corrupt_partial`(*enc*) may replace only some of them and supports corruption of the key. The latter equivalence can be applied only manually. To transform only some occurrences of decryption, the user should map the occurrences of decryption that he wants to transform to oracle *Odec*, the ones he wants to leave unchanged to oracle *Odec_unchanged*, and the ones that have already been transformed by a previous application of this equivalence to oracle $Odec\_unchanged'$.

- `expand INT_PTXT_sym_enc_all_args(`*key*, *cleartext*, *ciphertext*, *enc_seed*, *enc*, *enc_r*, *dec*, dec', *injbot*, *Pencptxt*`).` is similar to the above, with three additional arguments.

  enc_seed is the type of random coins for encryption, must be `bounded`.

  $enc\_r(cleartext, key, enc\_seed) : ciphertext$ is the encryption function that takes coins as argument (instead of generating them internally).

  dec' is the symbol that replaces *dec* after game transformation.

- `expand IND_CCA2_INT_PTXT_sym_enc(`*key*, *cleartext*, *ciphertext*, *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.

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

  $enc(cleartext, key) : ciphertext$ is the encryption function. Internally, it generates random coins, so that it is probabilistic.

  $dec(ciphertext, key)$: `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.

  $Penc(t, N, Nu, N', l, l')$ is the probability of breaking the IND-CCA2 property in time t for one key, N encryption queries that are different in both sides of the IND-CCA2 equivalence, Nu encryption queries that are the same in both side of the IND-CCA2 equivalence, N' decryption queries with cleartexts of length at most l and ciphertexts of length at most l'.

  $Pencptxt(t, N, N', Nu', l, l')$ is the probability of breaking the INT-PTXT property in time t for one key, N encryption queries, N' decryption queries that are modified by the transformation, and Nu' decryption queries that are left unchanged by the transformation, with cleartexts of length at most l and ciphertexts of length at most l'.

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

  This macro defines the equivalences named `ind_cca2`(*enc*), `ind_cca2_after_int_ptxt`(*enc*), `ind_cca2_partial`(*enc*), `int_ptxt`(*enc*), `int_ptxt_after_ind_cca2`(*enc*), and `int_ptxt_corrupt_partial`(*enc*), for use in the `crypto` command (see Section 7). The first three correspond to the IND-CCA2 property, the last three to the INT-PTXT property. The equivalence `ind_cca2`(*enc*) can be applied before applying the INT-PTXT property, while `ind_cca2_after_int_ptxt`(*enc*) can be applied after applying the INT-PTXT property. Similarly, the equivalence `int_ptxt`(*enc*) can be applied before applying the IND-CCA2 property, while `int_ptxt_after_ind_cca2`(*enc*) can be applied after applying the IND-CCA2 property. The equivalences `ind_cca2_partial`(*enc*) and `int_ptxt_corrupt_partial`(*enc*) may transform only some occurrences of encryption and/or decryption, and `int_ptxt_corrupt_partial`(*enc*) supports corruption of the key. They can be applied only manually, in any order. For `ind_cca2_partial`(*enc*), the user should map the occurrences of encryption that he wants to transform to oracle *Oenc*, the ones he wants to leave unchanged to oracle *Oenc_unchanged*. For `int_ptxt_partial`(*enc*), the user should map the occurrences of decryption that he wants to transform to oracle *Odec*, the ones he wants to leave unchanged to oracle *Odec_unchanged*.

CryptoVerif often needs manual guidance with this property, because it does not know which property (IND-CCA2 or INT-PTXT) to apply first. Moreover, when empty plaintexts are not allowed, IND-CCA2 and INT-PTXT is equivalent to IND-CPA and INT-CTXT, which is much easier to use for CryptoVerif, so we recommend using the latter property when possible.

- `expand IND_CCA2_INT_PTXT_sym_enc_all_args(`*key*, *cleartext*, *ciphertext*, *enc_seed*, *enc*, *enc_r*, *enc_r'*, *dec*, *dec'*, *injbot*, *Z*, *Penc*, *Pencptxt*`).` is similar to the above, with four additional arguments.

  enc_seed is the type of random coins for encryption, must be `bounded`.

  $enc\_r(cleartext, key, enc\_seed) : ciphertext$ is the encryption function that takes coins as argument (instead of generating them internally).

  enc_r' and dec' are the symbols that replace enc_r and dec respectively after game transformation.

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

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

  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`.)

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

  This macro defines equivalences named `sprp`(*enc*) and `sprp_partial`(*enc*) for use in the `crypto` command (see Section 7). These equivalences are generated via `equiv` ... `special`; the `crypto` command therefore supports special arguments *collisions_LHS* and, for `sprp_partial`(*enc*), collision matrix. See the explanation of the *collisions_LHS* argument, the collision matrix, and the oracles present in these equivalences in the documentation of `equiv` ... `special`.

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

key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *enc* without mentioning the length of the key), typically `fixed` and `large`.

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`.)

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

This macro defines equivalences named `prp(`*enc*`)` and `prp_partial(`*enc*`)` for use in the `crypto` command (see Section 7). These equivalences are generated via `equiv` ... `special`; the `crypto` command therefore supports special arguments *collisions_LHS* and, for `prp_partial(`*enc*`)`, collision matrix. See the explanation of the *collisions_LHS* argument, the collision matrix, and the oracles present in these equivalences in the documentation of `equiv` ... `special`.

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

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

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

  blocksize is type of the input and output of the cipher, must be `bounded` or `nonuniform` (to be able to generate random numbers from it; typically `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`.)

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

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

  enc_dec_oracle is a parametric process that allows the adversary to call the encryption and decryption functions. 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 the process *enc_dec_oracle*(*ck*) where *ck* is the cipherkey.

  qE is the number of queries to the encryption oracle.

  qD is the number of queries to the decryption oracle.

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

  This macro defines equivalences named `icm(`*enc*`)` and `icm_partial(`*enc*`)` for use in the `crypto` command (see Section 7). These equivalences are generated via `equiv` ... `special`; the `crypto` command therefore supports special arguments *collisions_LHS* and, for `icm_partial(`*enc*`)`, collision matrix. See the explanation of the *collisions_LHS* argument, the collision matrix, and the oracles present in these equivalences in the documentation of `equiv` ... `special`.

- `expand SUF_CMA_det_mac(`*mkey*, *macinput*, *macres*, *mac*, *check*, *Pmac*`).` defines an SUF-CMA (strongly unforgeable under chosen message attacks) deterministic MAC (message authentication code).

  The difference between a UF-CMA (unforgeable under chosen message attacks) 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. For deterministic MACs, the verification can be done by recomputing the MAC, and in this case, an UF-CMA MAC is always SUF-CMA, so we model only SUF-CMA deterministic MACs. This macro transforms tests $mac(k, m) = m'$ into $check(k, m, m')$, so that the MAC verification can also be written $mac(k, m) = m'$.

  mkey is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *mac* without mentioning the length of the key), typically `fixed` and `large`.

  macinput is the type of inputs of MACs

  macres is the type of MACs.

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

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

  $Pmac(t, N, N', Nu', l)$ is the probability of breaking the SUF-CMA property in time t for one key, N MAC queries, N' verification queries modified by the transformation and Nu verification queries left unchanged by the transformation for messages of length at most l.

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

  This macro defines the equivalences named `suf_cma`(*mac*), `suf_cma_corrupt`(*mac*), and `suf_cma_corrupt_partial`(*mac*), for use in the `crypto` command (see Section 7). All equivalences correspond to the SUF-CMA property, but the first one does not allow corruption of the secret keys while last two allow it. The last two equivalences are applied only manually, in particular because their automatic application can sometimes be done too early, when other transformations should first be done in order to eliminate uses of the secret keys. The equivalence `suf_cma_corrupt_partial`(*mac*) allows the user to transform only some occurrences of the MAC verification into a lookup in the MACed messages. The user should map the occurrences he wants to transform to the oracle *Ocheck* and the ones he does not want to transform to the oracle *Ocheck_unchanged*.

- `expand SUF_CMA_det_mac_all_args(`*mkey*, *macinput*, *macres*, *mac*, *mac'*, *check*, *Pmac*`).` is similar to the above, with one additional argument.

  mac' is the symbol that replaces *mac* after game transformation.

- `expand UF_CMA_proba_mac(`*mkey*, *macinput*, *macres*, *mac*, *check*, *Pmac*`).` defines a UF-CMA (unforgeable under chosen message attacks) probabilistic MAC (message authentication code). The arguments are the same as for `SUF_CMA_det_mac`, but the *mac* function chooses random coins internally so that it is probabilistic, and the verification is not done by recomputing the MAC. This macro defines the equivalences named `uf_cma`(*mac*), `uf_cma_corrupt`(*mac*), and `uf_cma_corrupt_partial`(*mac*) for use in the `crypto` command (see Section 7), similarly to `SUF_CMA_det_mac`.

- `expand UF_CMA_proba_mac_all_args(`*mkey*, *macinput*, *macres*, *mac_seed*, *mac*, *mac_r*, *mac_r'*, *check*, *check'*, *Pmac*`).` is similar to the above, with four additional arguments.

  mac_seed is the type of random coins for MAC, must be `bounded`.

  $mac\_r(macinput, mkey, mac\_seed) : macres$ is the MAC function that takes coins as argument (instead of generating them internally).

  mac_r' and *check'* are the symbols that replace *mac_r* and *check* respectively after game transformation.

- `expand SUF_CMA_proba_mac(`*mkey*, *macinput*, *macres*, *mac*, *check*, *Pmac*`).` defines a SUF-CMA (strongly unforgeable under chosen message attacks) probabilistic MAC (message authentication code). The arguments are the same as for `SUF_CMA_det_mac`, but the *mac* function chooses random coins internally so that it is probabilistic, and the verification is not done by recomputing the MAC. This macro defines the equivalences named `suf_cma`(*mac*), `suf_cma_corrupt`(*mac*), and `suf_cma_corrupt_partial`(*mac*), for use in the `crypto` command (see Section 7), similarly to `SUF_CMA_det_mac`.

- `expand SUF_CMA_proba_mac_all_args(`*mkey*, *macinput*, *macres*, *mac_seed*, *mac*, *mac_r*, mac_r', *check*, *Pmac*`).` is similar to the above, with three additional arguments.

  mac_seed is the type of random coins for MAC, must be `bounded`.

  $mac\_r(macinput, mkey, mac\_seed)$: *macres* is the MAC function that takes coins as argument (instead of generating them internally).

  mac_r' is the symbol that replaces *mac_r* after game transformation.

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

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *pkgen* without mentioning the length of the key), typically `fixed` and `large`.

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

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

  cleartext is the type of cleartexts.

  ciphertext is the type of ciphertexts.

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

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

  $enc(cleartext, pkey)$: *ciphertext* is the encryption function. Internally, it generates random coins, so that it is probabilistic.

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

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

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

  $Penc(t, N)$ 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*, and the probabilities *Penc*, *Penccoll* must be declared before this macro is expanded. The functions *skgen*, *pkgen*, *enc*, *dec*, *injbot*, and Z are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

  This macro defines the equivalences named `ind_cca2`(*enc*) and `ind_cca2_partial`(*enc*) for use in the `crypto` command (see Section 7). The equivalence `ind_cca2_partial`(*enc*) can be applied only manually and allows the user to replace the encryption of a message with the encryption of zeroes for only some occurrences of encryption under the considered key, the ones in which the public key appears explicitly.

- `expand IND_CCA2_public_key_enc_all_args(`*keyseed*, *pkey*, *skey*, *cleartext*, *ciphertext*, *enc_seed*, *skgen*, $skgen'$, *pkgen*, $pkgen'$, *enc*, *enc_r*, enc_r', *dec*, dec', *injbot*, *Z*, *Penc*, *Penccoll*`).` is similar to the above, with six additional arguments.

enc_seed is the type of random coins for encryption, must be `bounded`.

$enc\_r(cleartext, pkey, enc\_seed) : ciphertext$ is the encryption function that takes coins as argument (instead of generating them internally).

$pkgen'$, $skgen'$, enc_r', and dec' are the symbols that replace *pkgen*, *skgen*, *enc_r* and *dec* respectively after game transformation.

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

  keyseed is the type of key seeds, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of *pkgen* without mentioning the length of the key), typically `fixed` and `large`.

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

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

  signinput is the type of signature inputs.

  signature is the type of signatures.

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

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

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

  This macro defines the equivalences named `uf_cma`(*sign*), `uf_cma_corrupt`(*sign*), and `uf_cma_corrupt_partial`(*sign*), for use in the `crypto` command (see Section 7). All three equivalences correspond to the UF-CMA property, but the first one does not allow corruption of the secret keys while last two allow it. The last two equivalences are applied only manually, in particular because their automatic application can sometimes be done too early, when other transformations should first be done in order to eliminate uses of the secret keys. The equivalence `uf_cma_corrupt_partial`(*sign*) allows the user to transform only some occurrences of the signature verification into a lookup in the signed messages, the ones in which the public key appears explicitly.

- `expand UF_CMA_det_signature_all_args(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *skgen*, $skgen'$, *pkgen*, $pkgen'$, *sign*, $sign'$, *check*, $check'$, *Psign*, *Psigncoll*`).` is similar to the above with four additional arguments.

  $pkgen'$, $skgen'$, $sign'$, and $check'$ are the symbols that replace *pkgen*, *skgen*, *sign* and *check* respectively after game transformation.

- `expand SUF_CMA_det_signature(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *skgen*, *pkgen*, *sign*, *check*, *Psign*, *Psigncoll*`).` defines a SUF-CMA (strongly unforgeable under chosen message attacks) deterministic 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 `UF_CMA_det_signature`. This macro defines the equivalences named `suf_cma`(*sign*), `suf_cma_corrupt`(*sign*), and `suf_cma_corrupt_partial`(*sign*), for use in the `crypto` command (see Section 7).

- expand `SUF_CMA_det_signature_all_args(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *skgen*, *skgen*$'$, *pkgen*, *pkgen*$'$, *sign*, *sign*$'$, *check*, *check*$'$, *Psign*, *Psigncoll*`).` is similar to the above with four additional arguments.

  pkgen$'$, *skgen*$'$, *sign*$'$, and *check*$'$ are the symbols that replace *pkgen*, *skgen*, *sign* and *check* respectively after game transformation.

- expand `UF_CMA_proba_signature(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *skgen*, *pkgen*, *sign*, *check*, *Psign*, *Psigncoll*`).` defines a UF-CMA (strongly unforgeable under chosen message attacks) probabilistic signature scheme. The arguments are the same as for `UF_CMA_det_signature`, but the signature function internally generated random coins, so that it is probabilistic. This macro defines the equivalences named `uf_cma`(*sign*), `uf_cma_corrupt`(*sign*), and `uf_cma_corrupt_partial`(*sign*), for use in the `crypto` command (see Section 7).

- expand `UF_CMA_proba_signature_all_args(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *sign_seed*, *skgen*, *skgen*$'$, *pkgen*, *pkgen*$'$, *sign*, *sign_r*, *sign_r*$'$, *check*, *check*$'$, *Psign*, *Psigncoll*`).` is similar to the above, with six additional arguments.

  sign_seed is the type of random coins for signature, must be `bounded`.

  $sign\_r(signinput, skey, sign\_seed) : signature$ is the signature function that takes coins as argument (instead of generating them internally).

  pkgen$'$, *skgen*$'$, *sign_r*$'$, and *check*$'$ are the symbols that replace *pkgen*, *skgen*, *sign_r* and *check* respectively after game transformation.

- expand `SUF_CMA_proba_signature(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *skgen*, *pkgen*, *sign*, *check*, *Psign*, *Psigncoll*`).` defines a SUF-CMA (strongly unforgeable under chosen message attacks) probabilistic signature scheme. The arguments are the same as for `UF_CMA_det_signature`, but the signature function internally generated random coins, so that it is probabilistic. This macro defines the equivalences named `suf_cma`(*sign*), `suf_cma_corrupt`(*sign*), and `suf_cma_corrupt_partial`(*sign*), for use in the `crypto` command (see Section 7).

- expand `SUF_CMA_proba_signature_all_args(`*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *sign_seed*, *skgen*, *pkgen*, *sign*, *sign_r*, *check*, *Psign*, *Psigncoll*`).` is similar to the above, with six additional arguments.

  sign_seed is the type of random coins for signature, must be `bounded`.

  $sign\_r(signinput, skey, sign\_seed) : signature$ is the signature function that takes coins as argument (instead of generating them internally).

  pkgen$'$, *skgen*$'$, *sign_r*$'$, and *check*$'$ are the symbols that replace *pkgen*, *skgen*, *sign_r* and *check* respectively after game transformation.

- expand `ROM_hash(`$key, hashinput, hashoutput, hash, hashoracle, qH$`).` defines a hash function in the random oracle model [1].

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

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

  hashoutput is the type of the output of the hash function, must be `bounded` or `nonuniform` (typically `fixed`).

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

  hashoracle is a process that allows the adversary to call the hash function. WARNING: 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 $hashoracle(hk)$ in the executed process, where hk is the key.

  qH is the number of queries to the hash oracle.

The types *key*, *hashinput*, and *hashoutput* must be declared before this macro. The function *hash*, the process *hashoracle*, and the parameter *qH* are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

This macro defines equivalences named `rom`(*hash*) and `rom_partial`(*hash*) for use in the `crypto` command (see Section 7). These equivalences are generated via `equiv` ... `special`; the `crypto` command therefore supports special arguments *collisions_LHS* and, for `rom_partial`(*hash*), collision matrix. See the explanation of the *collisions_LHS* argument, the collision matrix, and the oracles present in these equivalences in the documentation of `equiv` ... `special`.

- `expand ROM_hash_large(`*key*, *hashinput*, *hashoutput*, *hash*, *hashoracle*, *qH*`).` defines a random oracle with a large output, that is, it optimizes the definition by eliminating collisions between random output elements. Its interface is the same as the one of `ROM_hash` above.

- `expand CollisionResistant_hash(`*key*, *hashinput*, *hashoutput*, *hash*, *hashoracle*, *Phash*`).` defines a collision-resistant hash function [10], [7, Section 8.2].

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

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

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

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

  hashoracle is a process that leaks the key that it receives as argument. 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, for instance by including the process *hashoracle*(*hk*), where *hk* is the key.

  Phash(t) is the probability of breaking collision resistance, for an adversary that runs in time at most t. (t is the time since the choice of the hash function, that is, of the key *hk*.)

  The types *key*, *hashinput*, and *hashoutput* and the probability *Phash* must be declared before this macro. The function *hash* and the process *hashoracle* are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

- `expand HiddenKeyCollisionResistant_hash(`*key*, *hashinput*, *hashoutput*, *hash*, *hashoracle*, *qH*, *Phash*`).` defines a hidden-key collision-resistant hash function [7, Section 8.6]. It differs from collision-resistance in that the adversary is not allowed to access the key that defines the hash function; it is just allowed to query the hash oracle. The interface is similar to collision-resistant hash functions above.

  hashoracle is a process that provides a hash oracle to the adversary. WARNING: A hidden-key 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 the hash oracle for that key must be provided to the adversary by including the process *hashoracle*(*hk*), where *hk* is the key.

  qH is the number of calls to the hash oracle provided by *hashoracle*.

  Phash(t, N) is the probability of breaking collision resistance, for an adversary that runs in time at most t and calls the hash oracle at most N times.

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

- `expand SecondPreimageResistant_hash(`*key*, *hashinput*, *hashoutput*, *hash*, *hashoracle*, *Phash*`).` defines a second-preimage-resistant hash function [10]. The interface is the same as for collision-resistant hash functions above. However, note that the argument type *hashinput* must be `bounded` or `nonuniform` so that one can generate random values in it. It is typically `fixed` and `large`.

- `expand HiddenKeySecondPreimageResistant_hash(`*key*, *hashinput*, *hashoutput*, *hash*, *hashoracle*, *qH*, *Phash*`).` defines a hidden-key second-preimage-resistant hash function. The interface is the same as for hidden-key collision-resistant hash functions above. However, note that the argument type *hashinput* must be `bounded` or `nonuniform` so that one can generate random values in it. It is typically `fixed` and `large`.

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

- `expand FixedSecondPreimageResistant_hash(`*hashinput*, *hashoutput*, *hash*, *Phash*`).` defines a second-preimage-resistant hash function, for a hash function without key. (It can also be interpreted as a hash function with a fixed key as in [10], which we omit in our model.)

  hashinput is the type of the input of the hash function. It must be `bounded` or `nonuniform` so that one can generate random values in it. It is typically `fixed` and `large`.

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

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

  Phash(t) is the probability of breaking second-preimage resistance, for an adversary that runs in time at most t.

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

- `expand PreimageResistant_hash(`*key*, *hashinput*, *hashoutput*, *hash*, *hashoracle*, *Phash*`).` defines a preimage-resistant hash function [10]. The interface is the same as for collision-resistant hash functions above. However, note that the argument type *hashinput* must be `bounded` or `nonuniform` so that one can generate random values in it. It is typically `fixed` and `large`.

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

  `expand PreimageResistant_hash_all_args(`*key*, *hashinput*, *hashoutput*, *hash*, *hash*$'$, *hashoracle*, *Phash*`).` is similar, with an additional argument *hash*$'$, which is a symbol that replaces *hash* after game transformation.

- `expand HiddenKeyPreimageResistant_hash(`*key*, *hashinput*, *hashoutput*, *hash*, *hashoracle*, *qH*, *Phash*`).` defines a hidden-key preimage-resistant hash function. The interface is the same as for hidden-key collision-resistant hash functions above. However, note that the argument type *hashinput* must be `bounded` or `nonuniform` so that one can generate random values in it. It is typically `fixed` and `large`.

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

  `expand HiddenKeyPreimageResistant_hash_all_args(`*key*, *hashinput*, *hashoutput*, *hash*, *hash*$'$, *hashoracle*, *qH*, *Phash*`).` is similar, with an additional argument *hash*$'$, which is a symbol that replaces *hash* after game transformation.

- `expand FixedPreimageResistant_hash(`*hashinput*, *hashoutput*, *hash*, *Phash*`).` defines a preimage-resistant hash function, for a hash function without key. (It can also be interpreted as a hash function with a fixed key as in [10], which we omit in our model.) The interface is the same as for fixed second-preimage-resistant hash functions above.

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

  `expand FixedPreimageResistant_hash_all_args(`*hashinput*, *hashoutput*, *hash*, *hash*$'$, *Phash*`).` is similar, with an additional argument *hash*$'$, which is a symbol that replaces *hash* after game transformation.

- Similarly to the macros above, for N from 1 to 10, the macros
  `expand ROM_hash_`N`(`*key*, *hashinput1*, …, *hashinputN*, *hashoutput*, *hash*, *hashoracle*, *qH*`).`
  `expand ROM_hash_large_`N`(`*key*, *hashinput1*, …, *hashinputN*, *hashoutput*, *hash*, *hashoracle*, *qH*`).`
  `expand CollisionResistant_hash_`N`(`*key*, *hashinput1*, …, *hashinputN*, *hashoutput*, *hash*,

hashoracle, *Phash*).
expand HiddenKeyCollisionResistant_hash_N(*key*, *hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *hashoracle*, *qH*, *Phash*).
expand SecondPreimageResistant_hash_N(*key*, *hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *hashoracle*, *Phash*).
expand HiddenKeySecondPreimageResistant_hash_N(*key*, *hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *hashoracle*, *qH*, *Phash*).
expand FixedSecondPreimageResistant_hash_N(*hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *Phash*).
expand PreimageResistant_hash_N(*key*, *hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *hashoracle*, *Phash*).
expand PreimageResistant_hash_all_args_N(*key*, *hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *hash'*, *hashoracle*, *Phash*).
expand HiddenKeyPreimageResistant_hash_N(*key*, *hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *hashoracle*, *qH*, *Phash*).
expand HiddenKeyPreimageResistant_hash_all_args_N(*key*, *hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *hash'*, *hashoracle*, *qH*, *Phash*).
expand FixedPreimageResistant_hash_N(*hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *Phash*).
expand FixedPreimageResistant_hash_all_args_N(*hashinput1*, ..., *hashinputN*, *hashoutput*, *hash*, *hash'*, *Phash*).
define hash functions with N arguments, with the same properties as above.

hashinput1, ..., *hashinputN* are the types of the inputs of the hash function and *hash*(*key*, *hashinput1*, ..., *hashinputN*) : *hashoutput* is the hash function, except for FixedSecondPreimageResistant_hash_N and FixedPreimageResistant_hash_N, where *hash*(*hashinput1*, ..., *hashinputN*) : *hashoutput* is the hash function.

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

  seed is the type of key seeds, must be bounded (to be able to generate random numbers from it, and to talk about the runtime of *pkgen* without mentioning the length of the key), typically fixed and large.

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

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

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

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

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

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

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

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

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

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

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

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

  $seed$ is the type of key seeds, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of $pkgen$ without mentioning the length of the key), typically `fixed` and `large`.

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

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

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

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

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

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

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

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

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

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

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

- `expand OW_trapdoor_perm_all_args(`$seed, pkey, skey, D, pkgen, pkgen', skgen, f, f', invf, POW$`).`
  `expand OW_trapdoor_perm_RSR_all_args(`$seed, pkey, skey, D, pkgen, pkgen', skgen, f, f', invf, POW$`).`
  `expand set_PD_OW_trapdoor_perm_all_args(`$seed$, $pkey$, $skey$, D, Dow, Dr, $pkgen$, $pkgen'$, $skgen$, f, f', $invf$, $concat$, P_PD_OW`).` are similar to `OW_trapdoor_perm`, `OW_trapdoor_perm_RSR`, and `set_PD_OW_trapdoor_perm_all_args` respectively, with two additional arguments.

  $pkgen'$ and f' are the symbols that replace $pkgen$ and f respectively after game transformation.

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

  key is the type of keys, must be `bounded` (to be able to generate random numbers from it, and to talk about the runtime of f without mentioned the length of the key), typically `fixed` and `large`.

  $input$ is the type of the input of the PRF.

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

  $f(key, input) : output$ 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 key, $input$, $output$ and the probability $Pprf$ must be declared before this macro is expanded. The function f is declared by this macro. It must not be declared elsewhere, and it can be used only after expanding the macro.

  This macro defines equivalences named `prf(`f`)` and `prf_partial(`f`)` for use in the `crypto` command (see Section 7). These equivalences are generated via `equiv` ... `special`; the `crypto` command therefore supports special arguments *collisions_LHS* and, for `prf_partial`(f), collision matrix. See the explanation of the *collisions_LHS* argument, the collision matrix, and the oracles present in these equivalences in the documentation of `equiv` ... `special`.

- `expand PRF_large(`*key*, *input*, *output*, f, *Pprf*`).` defines a pseudo-random function with a large output, that is, it optimizes the definition by eliminating collisions between random output elements. Its interface is the same as the one of `PRF` above.
- Similarly, for N from 1 to 10, the macros
`expand PRF_`N`(`*key*, *input1*, ..., *inputN*, *output*, f, *Pprf*`).`
`expand PRF_large_`N`(`*key*, *input1*, ..., *inputN*, *output*, f, *Pprf*`).`
define pseudo-random functions with N arguments, similarly to `PRF` and `PRF_large` above. *input1*, ..., *inputN* are the types of the inputs of the PRF and $f(key, input1, \ldots, inputN) : output$ is the PRF.
- The specification of Diffie-Hellman key agreements is typically composed of two or three macro expansions:
  - One from the following set of macros, which defines properties of the group:
    * `expand DH_basic(`$G, Z, g, exp, exp', mult$`).` defines a Diffie-Hellman structure G.
    G: type of group elements (must be `bounded` and `large`).
    Z: type of exponents (must be `bounded` and `large`).
    g: an element of the group G.
    $exp(G, Z) : G$: the exponentiation function.
    $exp'(G, Z) : G$: symbol used to replace *exp* after game transformations.
    $mult(Z, Z) : Z$: the multiplication function for exponents, commutative.
    The equation $exp(exp(a, x), y) = exp(a, mult(x, y))$ must be satisfied.
    The private Diffie-Hellman keys are generated by choosing an element randomly in Z, according to its default distribution (which is not necessarily uniform). The public Diffie-Hellman keys are generated as $X = exp(g, x)$, where x is a private Diffie-Hellman key, and similarly $Y = exp(g, y)$. The Diffie-Hellman shared secret is $exp(X, y) = exp(Y, x) = exp(g, mult(x, y))$. This macro makes no other assumption. In particular, it allows G to contain elements other than those generated by g.
    The types G and Z must be declared before this macro. The functions g, *exp*, and *mult* are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.
    * `expand DH_basic_with_is_neutral(`$G, Z, g, exp, exp', mult, is\_neutral$`).` defines a Diffie-Hellman structure like `expand DH_basic(`$G, Z, g, exp, exp', mult$`).` with additionally $is\_neutral(g)$ is false and for X in G, $is\_neutral(exp(X, y))$ if and only if $is\_neutral(X)$.
    $is\_neutral(G) : bool$ is defined by this macro. It must not be declared elsewhere, and can be used only after expanding the macro.
    Prime-order groups with the neutral element included satisfy this assumption, for instance, where $is\_neutral(X)$ is true if and only if X is the neutral element. Prime-order groups without the neutral element also satisfy this assumption, with $is\_neutral(X) = false$.
    * `expand DH_subgroup(`G, Z, g, *exp*, *mult*, *subG*, g_k, exp_div_k, exp_div_k', pow_k, *subGtoG*`).` defines a Diffie-Hellman structure that satisfies the following properties:
    G: type of elements (must be `bounded` and `large`).
    Z: type of exponents, a set of integers multiple of k, prime to n (possibly modulo kn); k is prime to n; Z must be `bounded` and `large`.
    g: an element of G.
    There is an exponentiation function such that for X in G and y integer, we have X^y in G with the following properties:
      1. $(X^x)^y = X^(xy)$;
      2. $subG = \{X^k \mid X \in G\}$ is a subset of G;

3. for X, X' in $subG$, for any x prime to n, $X^x = X'^x \Rightarrow X = X'$;
4. exponentiation yields the same results for exponents equal modulo kn.

$exp(G, Z) : G$ is defined by $exp(X, y) = X^y$.
$mult(Z, Z) : Z$ is the product of integers (possibly modulo kn), commutative.
$subG = \{X^k \mid X \in G\}$ is a subset of G as mentioned above. The type $subG$ must be `bounded` and `large`.
$g\_k = g^k \in subG$.
$exp\_div\_k(subG, Z) : subG$ is defined by $exp\_div\_k(X, y) = X^{y/k}$.
exp_div_k' is defined like exp_div_k; it replaces exp_div_k after games transformations.
$pow\_k(G) : subG$ is defined by $pow\_k(X) = X^k$.
$subGtoG(subG) : G$ is the injection from $subG$ to G.
The types G, Z, and $subG$ must be declared before expanding this macro. The constants g and g_k, and the functions exp, $mult$, exp_div_k, exp_div_k', pow_k, $subGtoG$ are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.
When this macro is used, the other Diffie-Hellman macros (detailed below) except `DH_exclude_weak_keys` should be applied to the subgroup, that is, `expand` $assumption$`(`$subG$, Z, g_k, exp_div_k, exp_div_k', $mult$, ...`).`
Curve25519 satisfies these properties with $k = 8$, $n = pp'$ where the curve has order kp and the quadratic twist has order $k'p'$ ($k' = 4$, p and p' are large primes). See `https://hal.inria.fr/hal-02100345`.
Curve448 satisfies these properties with $k = 4$, $n = pp'$ after removing the weak private key kp which is not prime to $n = pp'$, where the curve has order kp and the quadratic twist has order $k'p'$ ($k = k' = 4$, p and p' are large primes). This can be done as follows using `DH_exclude_weak_keys` defined below:

> `expand DH_subgroup(`$G, Znw, g, expnw, mult, subG, g\_k, exp\_div\_k, exp\_div\_k',$
> $pow\_k, subGtoG$`).`
> `letproba` $Pweak\_key = 2\hat{}-445$.
> `expand DH_exclude_weak_keys(`$G, Z, Znw, ZnwtoZ, exp, expnw, Pweak\_key$`).`

Groups of prime order q also satisfy these properties, with $k = 1$, $n = q$, $subG = G$, $g\_k = g$, pow_k and $subGtoG$ are the identity (assuming private keys are chosen in $\{1, \ldots, q-1\}$).

* `expand DH_subgroup_with_is_neutral(`G, Z, g, exp, $mult$, $subG$, g_k, exp_div_k, exp_div_k', pow_k, $subGtoG$, $is\_neutral\_G$, $is\_neutral\_subG$`).` defines a Diffie-Hellman structure like `expand DH_subgroup(`G, Z, g, exp, $mult$, $subG$, g_k, exp_div_k, exp_div_k', pow_k, $subGtoG$`).` with additionally $is\_neutral(g^k)$ is false and for X in $subG$, $is\_neutral(X^y)$ if and only if $is\_neutral(X)$.
$is\_neutral\_G(G) : bool$ and $is\_neutral\_subG(subG) : bool$ correspond to the function $is\_neutral$, respectively on G and on $subG$. These functions are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.
Curve22519 satisfies these properties with $is\_neutral(X) = (X = 0)$.
Curve448 also satisfies these properties with $is\_neutral(X) = (X = 0)$, after removing the weak private key as follows:

> `expand DH_subgroup_with_is_neutral(`$G, Znw, g, expnw, mult, subG, g\_k,$
> $exp\_div\_k, exp\_div\_k', pow\_k, subGtoG, is\_neutral\_G, is\_neutral\_subG$`).`
> `letproba` $Pweak\_key = 2\hat{}-445$.
> `expand DH_exclude_weak_keys(`$G, Z, Znw, ZnwtoZ, exp, expnw, Pweak\_key$`).`

Prime order groups (without the neutral element) satisfy these properties with $is\_neutral(X) = false$.

* `expand DH_good_group(`$G, Z, g, exp, exp', mult$`).` defines a group G like `DH_basic`, with the following additional properties: G is a group of prime order q, with the neutral element excluded, the set of exponents Z is $\{1, \dots, q-1\}$, g is a generator of G, $mult$ is the product modulo q in $\{1, \dots, q-1\}$, i.e. in the group $(\mathbb{Z}/q\mathbb{Z})*$, the distributions of random choices in Z and G are uniform.
  It may not be obvious when an element is received on the network whether it really belongs to the group G generated by g. That should be checked for the properties assumed in this macro to hold.
  This macro defines the following equivalences for use in the `crypto` command (see Section 7):
  - `group_to_exp_strict`(exp) which allows to replace a random $X \in G$ with $exp(g, x)$ for a random $x \in Z$, provided $exp(X, \_)$ occurs in the game.
  - `group_to_exp`(exp) which allows to replace a random $X \in G$ with $exp(g, x)$ for a random $x \in Z$ in any case. (This transformation is applied only manually.)
  - `exp_to_group`(exp) which allows to replace $exp(g, x)$ for a random $x \in Z$ with a random $X \in G$.
  - `exp'_to_group`(exp) which allows to replace $exp'(g, x)$ for a random $x \in Z$ with a random $X \in G$.
* `expand DH_single_coord_ladder(`G, Z, g, exp, $mult$, $subG$, Znw, $ZnwtoZ$, g_k, exp_div_k, exp_div_k', pow_k, $subGtoG$, is_zero_G, is_zero_subG`).` models an elliptic curve defined by the equation $Y^2 = X^3 + AX^2 + X$ in the field $\mathbb{F}_p$ of non-zero integers modulo the large prime p, where A^2-4 is not a square modulo p. This curve must form a commutative group of order kq where k is a small integer and q is a large prime. Its quadratic twist must form a commutative group of order $k'q'$ where k' is a small integer and q' is a large prime. k must be a multiple of k'. We must use a single coordinate ladder defined as follows: we consider the elliptic curve $E(\mathbb{F}_{p^2})$ defined by the equation $Y^2 = X^3 + AX^2 + X$ in a quadratic extension $\mathbb{F}_{p^2}$ of $\mathbb{F}_p$, we define $X_0 : E(\mathbb{F}_{p^2}) \to \mathbb{F}_{p^2}$ by $X_0(\infty) = 0$ and $X_0(X, Y) = X$, and for $X \in \mathbb{F}_p$ and y an integer, we define $y \cdot X \in \mathbb{F}_p$ as $y \cdot X = X_0(yQ)$ for all $Q \in E(\mathbb{F}_{p^2})$ such that $X_0(Q) = X$. The value $g = X_0(g_0)$ represents the base point g_0, which must have order q. The public keys (bitstrings) are mapped to elements of $\mathbb{F}_p$ by the function red and conversely, elements of $\mathbb{F}_p$ are mapped to public keys by the function repr, such that red$\circ$repr is the identity. The Diffie-Hellman "exponentiation" is defined by
  $$\exp(X, y) = \mathrm{repr}(y \cdot \mathrm{red}(X))$$
  The secret keys are chosen uniformly in $\{kn \mid n \in [n_{min}, n_{max}]\}$ where $n_{min} < n_{max}$, $n_{max} - n_{min} < q$ and $n_{max} - n_{min} < q'$. Therefore the set of secret keys may contain a multiple of q (resp. q'). Such keys are weak, in the sense that they yield 0 for all public keys on the curve (resp. on the twist). We exclude them as a first step in the proof, by applying the equivalence `exclude_weak_keys`(Z) defined by this macro, automatically or with the `crypto` command (see Section 7).
  This model is justified in [9].
  G: type of public keys (must be `bounded` and `large`).
  $subG$: type of $\{k \cdot X \mid X \in F_p\}$ (must be `bounded`, `nonuniform`, and `large`). Random choices in $subG$ are done by choosing uniformly in $\{x \cdot g \mid x \in \{1, \dots, q-1\}\}$. (This set is not the whole $subG$, since $subG$ also contains elements of the twist.) This is important when the DDH assumption or the square DDH assumption is used.
  Z, Znw: type of exponents (must be `bounded`, `nonuniform`, and `large`). Znw is the set of integers multiple of k, prime to qq' modulo kqq', that is, exponents without weak keys. Random choices in Znw are done by choosing uniformly in $\{kn \mid n \in [n_{min}, n_{max}], n$ prime to $qq'\}$. Z is the set of integers multiple of k modulo kqq', that is, exponents with weak keys. Random choices in Z are done by choosing uniformly in $\{kn \mid n \in [n_{min}, n_{max}]\}$, hence `Pcoll1rand`$(Z) = 1/(n_{max} - n_{min} + 1)$.
  $ZnwtoZ(Znw) : Z$: injection from Znw to Z.

$g : G$: represents the base point.
$exp(G, Z) : G$: the exponentiation function.
$mult(Znw, Znw) : Znw$: the multiplication function for exponents, defined as $mult(x, y) = x.y \mod kqq'$. (It remains in Znw.)
$g\_k = k \cdot \mathrm{red}(g)$. It is an element of $subG$.
$exp\_div\_k(subG, Znw) : subG$ is defined by $exp\_div\_k(X, y) = (y/k) \cdot X$.
exp_div_k': symbol that replaces exp_div_k after game transformation, with the same definition as exp_div_k.
$pow\_k(G) : subG$, defined by $pow\_k(x) = k \cdot \mathrm{red}(x)$.
$subGtoG(subG) : G$ is repr restricted to $subG$.
$is\_zero\_G(G) : bool$ is defined by: $is\_zero\_G(X)$ is true when X is the public key 0.
$is\_zero\_subG(subG) : bool$ is defined by: $is\_zero\_subG(X)$ is true when X is the public key 0.
The types G, $subG$, Z, and Znw must be declared before this macro. The functions g, exp, $mult$, $ZnwtoZ$, g_k, exp_div_k, exp_div_k', pow_k, $subGtoG$, is_zero_G, is_zero_subG are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.
When this macro is used, the other Diffie-Hellman macros (detailed below) should be applied to the subgroup, that is, `expand` *assumption*`(`$subG$, Znw, g_k, exp_div_k, exp_div_k', $mult$, `...).`

    * `expand DH_X25519(`G, Z, g, exp, $mult$, $subG$, g_k, exp_div_k, exp_div_k', pow_k, $subGtoG$, is_zero_G, is_zero_subG`).` models Curve25519 as defined in RFC 7748 (`https://tools.ietf.org/html/rfc7748`). It is justified in detail in [9]. More generally, it supports the same curves as `DH_single_coord_ladder` with the additional assumption that all secret keys are prime to qq'. Therefore, we do not need to exclude weak secret keys, so the parameters Znw and $ZnwtoZ$ are removed, and we use Z instead of Znw.
    Curve25519 satisfies these assumptions with $p = 2^{255} - 19$, $k = 8$, $k' = 4$, $q = 2^{252} + \delta$ with $0 < \delta < 2^{128}$, $q' = 2^{253} - 9 - 2\delta$, $\mathrm{red}(X) = (X \bmod 2^{255}) \bmod p$, $\mathrm{repr}(X)$ is the representation of X as an element of $\{0, \dots, p-1\}$, $n_{min} = 2^{251}$, and $n_{max} = 2^{252} - 1$, so $\mathtt{Pcoll1rand}(Z) = 2^{-251}$. (For simple examples that use Curve25519, using the macro `DH_proba_collision`, possibly with `DH_subgroup` or `DH_subgroup_with_is_neutral`, may also work.)
    * `expand DH_X448(`G, Z, g, exp, $mult$, $subG$, Znw, $ZnwtoZ$, g_k, exp_div_k, exp_div_k', pow_k, $subGtoG$, is_zero_G, is_zero_subG`).` models Curve448 as defined in RFC 7748 (`https://tools.ietf.org/html/rfc7748`). More generally, it supports the same curves as `DH_single_coord_ladder` with the additional assumptions that there is at most one secret key multiple of q or q', and that $q = -1 \mod 4$, so -1 is not a square modulo q. That allows to reduce some probabilities. This model is justified in [9].

– Optionally, one or more of the following macros:
    * `expand DH_exclude_weak_keys(`G, Z, Znw, $ZnwtoZ$, exp, $expnw$, $Pweak\_key$`).` allows excluding weak private keys.
    Z is a set of Diffie-Hellman private keys (exponents), possibly containing weak private keys. The type Z must be `bounded` and `large`.
    Znw is the subset of Z obtained by removing weak keys. The type Znw must be `bounded` and `large`.
    $ZnwtoZ(Znw : Z$ is the injection from Znw to Z.
    $exp(G, Z) : G$ and $expnw(G, Znw) : G$ are exponentiation functions.
    $Pweak\_key$ is the probability that a weak private key is chosen.
    This macro defines an equivalence `exclude_weak_keys(`Z`)`, for use with the `crypto` command (see Section 7), which replaces the random choice of private keys in Z with a choice in Znw, so that there are no weak private keys. It should be applied early in the proof, before applying Diffie-Hellman properties.
    The types G, Z, Znw, the function $expnw$, and the probability $Pweak\_key$ must be declared before expanding this macro. (The function $expnw$ should be defined by expanding one of the macros `DH_basic`, `DH_basic_with_is_neutral`, `DH_subgroup`,

`DH_subgroup_with_is_neutral`, or `DH_good_group` with *Znw* instead of *Z* and *expnw* instead of *exp*.) The functions *ZnwtoZ* and *exp* are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.
When this macro is used, the other Diffie-Hellman macros must use *Znw* instead of *Z* and *expnw* instead of *exp*. This macro should not be used with `DH_single_coord_ladder`, `DH_X25519`, `DH_X448`. There are no weak keys for `DH_X25519` (if the specification of the choice of exponents for Curve25519 is followed). `DH_single_coord_ladder` and `DH_X448` already include the needed removal of weak keys.
This macro is useful for Curve448 (defined using `DH_basic`, `DH_basic_with_is_neutral`, `DH_subgroup`, or `DH_subgroup_with_is_neutral`), which has a weak key kp, with $k = 4$ where the curve has order kp, so $Znw = Z \setminus \{kp\}$, $Pweak\_key = 2^-445$. It is also useful for groups of prime order q in case private keys are chosen in $\{0, \dots, q-1\}$: one should eliminate the weak private key 0, so $Z = \{0, \dots, q-1\}$, $Znw = \{1, \dots, p-1\}$, $Pweak\_key = 1/q$.

* `expand DH_proba_collision(`$G, Z, g, exp, exp', mult, PCollKey1, PCollKey2$`).` adds the following properties: the probability that $exp(g, x) = Y$ where x is random and Y is independent of x is at most $PCollKey1$, and the probability that $exp(g, mult(x, y)) = Y$ where x and y are independent random private keys and Y is independent of x or y is at most $PCollKey2$. These probabilities are negligible in most Diffie-Hellman groups, but need to be evaluated more precisely for using this property.
The types G and Z and the probabilities $PCollKey1$ and $PCollKey2$ must be declared before this macro. The functions g, exp, and $mult$ are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.
`DH_proba_collision` should be used with `DH_basic`, `DH_basic_with_is_neutral`, `DH_subgroup`, or `DH_subgroup_with_is_neutral`; with the latter two, it should be applied to the subgroup. (The macros `DH_good_group`, `DH_single_coord_ladder`, `DH_X25519`, `DH_X448` already include such collision information.) It should not be used with `square_DH_proba_collision` or `is_neutral_DH_proba_collision`: they include information provided by `DH_proba_collision`.
* `expand square_DH_proba_collision(`$G, Z, g, exp, exp', mult, PCollKey1, PCollKey2, PCollKey3$`).` is similar to `DH_proba_collision`, but additionally says that the probability that $exp(g, mult(x, x)) = Y$ where x is random and Y is independent of x is at most $PCollKey3$, with $PCollKey3 \geq PCollKey2$.
The types G and Z and the probabilities $PCollKey1$, $PCollKey2$, and $PCollKey3$ must be declared before this macro. The functions g, exp, and $mult$ are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.
`square_DH_proba_collision` should be used with `DH_basic`, `DH_basic_with_is_neutral`, `DH_subgroup`, or `DH_subgroup_with_is_neutral`; with the latter two, it should be applied to the subgroup. (The macros `DH_good_group`, `DH_single_coord_ladder`, `DH_X25519`, `DH_X448` already include such collision information.) It should not be used with `DH_proba_collision` or `is_neutral_DH_proba_collision`: it includes information provided by `DH_proba_collision` and `is_neutral_DH_proba_collision` includes information provided by `square_DH_proba_collision`.
* `expand is_neutral_DH_proba_collision(`$G, Z, g, exp, exp', mult, is\_neutral, PCollKey2, PCollKey3, PCollKey4$`).` should be used with either `DH_basic_with_is_neutral` or `DH_subgroup_with_is_neutral`; with the latter, it should be applied to the subgroup, as follows:

  `expand DH_subgroup_with_is_neutral(`$G, Z, g, exp, mult, subG, g\_k, exp\_div\_k, exp\_div\_k', pow\_k, subGtoG, is\_neutral\_G, is\_neutral\_subG$`).`
  `expand is_neutral_DH_proba_collision(`$subG, Z, g\_k, exp\_div\_k, exp\_div\_k', mult, is\_neutral\_subG, PCollKey2, PCollKey3, PCollKey4$`).`

In addition to the information provided by `DH_basic_with_is_neutral` or `DH_subgroup_with_is_neutral`, it assumes that

- if $is\_neutral(X)$ and $is\_neutral(Y)$ then $X = Y$ (in other words, there is 0 or 1 neutral element in G);
- the probability that $exp(X, x) = Y$ where x is random and X and Y are independent of x and not neutral is at most $PCollKey2$;
- the probability that $exp(g, mult(x, x)) = Y$ with random x and Y independent of x is at most $PCollKey3$, with $PCollKey3 \geq PCollKey2$.
- the probability that $exp(X, y) = exp(X, z)$ with y, z random independent of each other and X is not neutral is at most $PCollKey4$.

It states these collisions and others that can be inferred from them.

It should not be used with `DH_proba_collision` or `square_DH_proba_collision`: it includes the information provided by these two macros.

* `expand DH_dist_random_group_element_vs_exponent(`$G, Z, g, exp, exp', mult,$ $PDist$`).` This macro says that the probability of distinguishing a random group element from an exponentiation $exp(g, x)$ with a random exponent x is at most $PDist$. The other arguments are as in `DH_basic` and all arguments must be defined before expanding the macro.

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

  - `group_to_exp_strict(`exp`)` which allows to replace a random $X \in G$ with $exp(g, x)$ for a random $x \in Z$, provided $exp(X, \_)$ occurs in the game.
  - `group_to_exp(`exp`)` which allows to replace a random $X \in G$ with $exp(g, x)$ for a random $x \in Z$ in any case. (This transformation is applied only manually.)
  - `exp_to_group(`exp`)` which allows to replace $exp(g, x)$ for a random $x \in Z$ with a random $X \in G$.
  - `exp'_to_group(`exp`)` which allows to replace $exp'(g, x)$ for a random $x \in Z$ with a random $X \in G$.

  This macro can be used with any of the previous macros, except that it is useless with the macro `DH_good_group`, because this macro already includes these properties with $PDist = 0$. When the macro `DH_subgroup`, `DH_subgroup_with_is_neutral`, `DH_single_coord_ladder`, `DH_X25519`, or `DH_X448` is used, this macro should be applied to the subgroup. For instance, with `expand DH_single_coord_ladder(`G, Z, g, exp, $mult$, $subG$, Znw, $ZnwtoZ$, g_k, exp_div_k, exp_div_k', pow_k, $subGtoG$, $zero$, sub_zero`).`, it should be `expand DH_dist_random_group_element_vs_exponent(`$subG$, Znw, g_k, exp_div_k, exp_div_k', $mult$, $Pdist$`).`

– One from the following set of macros, which defines the Diffie-Hellman assumption itself:

* `expand CDH(`$G, Z, g, exp, exp', mult, p$`).` says that the group G satisfies the computational Diffie-Hellman assumption; $p(t)$ is the probability of breaking the CDH assumption, for one pair of exponents, in time t. This macro defines the equivalence `cdh(`exp`)`, whichs corresponds to the CDH property, for use in the `crypto` command (see Section 7).
* `expand CDH_RSR(`$G, Z, g, exp, exp', mult, p, p_d$`).` is similar to `CDH`, but uses random self reducibility. p_d is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key. We have $p_d = 0$ when the exponents are chosen uniformly in $(\mathbb{Z}/q\mathbb{Z})^*$ in a group of prime order q (as assumed by the macro `DH_good_group`), $p_d = 2^{-126}$ for Curve25519, and $p_d = 2^{-221}$ for Curve448.
* `expand DDH(`$G, Z, g, exp, exp', mult, p$`).` says that the group G satisfies the decisional Diffie-Hellman assumption; $p(t)$ is the probability of breaking the DDH assumption, for one pair of exponents, in time t. This macro defines the equivalence `ddh(`exp`)`, whichs corresponds to the DDH property, for use in the `crypto` command (see Section 7).
* `expand DDH_RSR(`$G, Z, g, exp, exp', mult, p, p_d$`).` is similar to `DDH`, but uses random self reducibility. p_d is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key (see `CDH_RSR`). This macro is much more limited than the

macro DDH. It does not support corruption; corrupted keys must be in variables different from the ones containing honest keys. It supports only a single Diffie-Hellman query for each exponent a_i, associated with an arbitrary b_j and no Diffie-Hellman queries for b_j. The default distribution on G must be as follows: There is an underlying prime-order group (the Diffie-Hellman group itself when it has prime order; the prime-order subgroup of the curve generated by the base point for Curve25519/Curve448). The default distribution on G is obtained by choosing uniformly an element in that group minus its neutral element and taking the associated public key in G (the group element itself for prime-order Diffie-Hellman groups; the encoding of its X coordinate for Curve25519/Curve448). CDH and GDH with random self reducibility do not have such limitations.

* `expand GDH(`$G, Z, g, exp, exp', mult, p, p_d$`).` says that the group G satisfies the gap Diffie-Hellman assumption (GDH). The probability $p(t, n)$ is the probability of breaking the GDH assumption for one pair of exponents in time t with at most n calls to the decisional Diffie-Hellman oracle. p_d is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key (see `CDH_RSR`). It is needed because, for Curve25519/448, to make the Diffie-Hellman decision oracle unambiguous, we generate secret keys in $[(p + 1)/2, p - 1]$ instead of the set used for generating secret keys in the Curve25519/448 implementation. (The latter set yields equivalent secret keys with small probability.) We make the same change of distribution for rerandomization. This macro defines the equivalence `gdh(`exp`)`, whichs corresponds to the GDH property, for use in the `crypto` command (see Section 7).
* `expand GDH_RSR(`$G, Z, g, exp, exp', mult, p, p_d$`).` is similar to `GDH`, but uses random self reducibility.
* `expand square_CDH(`$G, Z, g, exp, exp', mult, p, sqp$`).` says that the group G satisfies the computational Diffie-Hellman assumption and the square computational Diffie-Hellman assumption; $p(t)$ is the probability of breaking the CDH assumption, for one pair of exponents, in time t and $sqp(t)$ is the probability of breaking the square CDH assumption, for one pair of exponents, in time t. This macro defines the equivalence `cdh(`exp`)`, whichs corresponds to the (square) CDH property, for use in the `crypto` command (see Section 7). When the group has prime order, the computational Diffie-Hellman assumption is equivalent to the square variant, but CryptoVerif can do more proofs using the square variant. (It allows transforming $exp(g, mult(x, x))$.)
* `expand square_CDH_RSR(`$G, Z, g, exp, exp', mult, sqp$`).` says that the group G satisfies the square computational Diffie-Hellman assumption; $sqp(t)$ is the probability of breaking the square CDH assumption, for one pair of exponents, in time t; this macro uses random self reducibility, and p_d is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key (see `CDH_RSR`). This macro defines the equivalence `cdh(`exp`)`, whichs corresponds to the square CDH property, for use in the `crypto` command (see Section 7).
* `expand square_DDH(`$G, Z, g, exp, exp', mult, p, sqp$`).` says that the group G satisfies the decisional Diffie-Hellman assumption and the square decisional Diffie-Hellman assumption; $p(t)$ is the probability of breaking the DDH assumption, for one pair of exponents, in time t and $sqp(t)$ is the probability of breaking the square DDH assumption, for one pair of exponents, in time t. This macro defines the equivalence `ddh(`exp`)`, whichs corresponds to the (square) DDH property, for use in the `crypto` command (see Section 7).
* `expand square_GDH(`$G, Z, g, exp, exp', mult, p, sqp, p_d$`).` says that the group G satisfies the gap Diffie-Hellman (GDH) assumption and the square gap Diffie-Hellman assumption; $p(t, n)$ is the probability of breaking the GDH assumption, for one pair of exponents, in time t with at most n calls to the decisional Diffie-Hellman oracle and $sqp(t, n)$ is the probability of breaking the square GDH assumption, for one pair of exponents, in time t with at most n calls to the decisional Diffie-Hellman oracle. p_d is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key (see `CDH_RSR`). It is needed because, for Curve25519/448, to make the Diffie-Hellman decision oracle unambiguous, we generate secret keys in $[(p + 1)/2, p - 1]$ instead of the set used for generating secret keys in the Curve25519/448 implementation. (The latter set yields

equivalent secret keys with small probability.) We make the same change of distribution for rerandomization. This macro defines the equivalence `gdh`(exp), whichs corresponds to the (square) GDH property, for use in the `crypto` command (see Section 7).

* `expand square_GDH_RSR(`$G, Z, g, exp, exp', mult, sqp, p_d$`).` says that the group G satisfies the square gap Diffie-Hellman assumption; $sqp(t, n)$ is the probability of breaking the square GDH assumption, for one pair of exponents, in time t with at most n calls to the decisional Diffie-Hellman oracle; this macro uses random self reducibility, and p_d is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key (see `CDH_RSR`). This macro defines the equivalence `gdh`(exp), whichs corresponds to the square GDH property, for use in the `crypto` command (see Section 7).

* `expand PRF_ODH1(`$G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p$`).` says that the group G satisfies the PRF-ODH1 (pseudo-random function oracle Diffie-Hellman) assumption, which corresponds to PRF-ODHnn in [6]. The pseudo-random function $prf(G, prf\_in) : prf\_out$ takes as argument a group element in G and an element in prf_in, and produces a result in prf_out. The type prf_out must be `bounded` or `nonuniform`. This assumption means that an adversary that has 2 public Diffie-Hellman keys $exp(g, a)$ and $exp(g, b)$ for random a, b cannot distinguish $x \mapsto prf(exp(g, mult(a, b)), x)$ from a random function. A random function returns a fresh random value when it is called with a new argument and the previous result when it is called with the same argument as a previous call. The probability $p(t, n)$ is the probability of breaking the PRF-ODH1 assumption in time t with n queries to $prf(exp(g, mult(a, b)), x)$. This macro defines the equivalence `prf_odh`(prf), whichs corresponds to the PRF-ODH1 property, for use in the `crypto` command (see Section 7).

  When this assumption is used with `DH_subgroup`, `DH_subgroup_with_is_neutral`, `DH_X25519`, `DH_X448`, or `DH_single_coord_ladder`, it must be applied to the subgroup, which can be done as follows:

  `expand DH_single_coord_ladder(`$G, Z, g, exp, mult, subG, Znw, ZnwtoZ, g\_k,$
  $exp\_div\_k, exp\_div\_k', pow\_k, subGtoG, zero, sub\_zero$`).`
  `expand PRF_ODH1(`$subG, Znw, prf\_in, prf\_out, g\_k, exp\_div\_k, exp\_div\_k', mult,$
  prf_subG, p`).`
  `fun` $prf(G, prf\_in) : prf\_out.$
  `equation forall` $x : subG, y : prf\_in; prf(subGtoG(x), y) = prf\_subG(x, y).$

* `expand PRF_ODH2(`$G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p, PCollKey1$`).` says that the group G satisfies the PRF-ODH2 assumption, which corresponds to PRF-ODHmm in [6]. The types prf_in and prf_out and the pseudo-random function prf are defined as for `PRF_ODH1`. This assumption means that an adversary that has 2 public Diffie-Hellman keys $exp(g, a)$ and $exp(g, b)$ for random a, b and has access to the oracles $(Y, x) \mapsto prf(exp(Y, a), x)$ and $(X, x) \mapsto prf(exp(X, b), x)$ cannot distinguish $x \mapsto prf(exp(g, mult(a, b)), x)$ from a random function. The probability $p(t, n, n')$ is the probability of breaking the PRF-ODH2 assumption in time t with n queries to $prf(exp(g, mult(a, b)), x)$ and n' queries to $(Y, x) \mapsto prf(exp(Y, a), x)$ and $(X, x) \mapsto prf(exp(X, b), x)$ in total. This macro defines the equivalence `prf_odh`(prf), which corresponds to the PRF-ODH2 property, for use in the `crypto` command (see Section 7).

  When this assumption is used with `DH_subgroup`, `DH_subgroup_with_is_neutral`, `DH_X25519`, `DH_X448`, or `DH_single_coord_ladder`, it must be applied to the subgroup, which can be done as for `PRF_ODH1`.

  If G is Curve448, the weak private key kp must be excluded, which can be done using `DH_exclude_weak_keys`, `DH_X448`, or `DH_single_coord_ladder`.

  Additionally, this assumption requires that it is possible to test efficiently whether $\exp(Y, a) = \exp(g, ab)$ knowing just Y and $B = \exp(g, b)$ (so the result does not depend on a). This is possible for prime-order groups as well as Curve25519 and Curve448 when the weak private key is excluded. When this is true, we say that the keys Y and B are equivalent. The probability that two honestly generated random keys are equivalent is bounded by $PCollKey1$.

* expand square_PRF_ODH1($G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p, sqp$). says that the group G satisfies the square PRF-ODH1 assumption and the PRF-ODH1 assumption. The types prf_in and prf_out and the pseudo-random function prf are defined as for PRF_ODH1. The square PRF-ODH1 assumption means that an adversary that has a public Diffie-Hellman key $exp(g, a)$ for random a cannot distinguish $x \mapsto prf(exp(g, mult(a, a)), x)$ from a random function. The probability $sqp(t, n)$ is the probability of breaking the square PRF-ODH1 assumption in time t with n queries to $prf(exp(g, mult(a, a)), x)$. The probability $p(t, n)$ is the probability of breaking the PRF-ODH1 assumption in time t with n queries to $prf(exp(g, mult(a, b)), x)$. This macro defines the equivalence prf_odh(prf), whichs corresponds to the square PRF-ODH1 and PRF-ODH1 properties, for use in the crypto command (see Section 7).
When this assumption is used with DH_subgroup, DH_subgroup_with_is_neutral, DH_X25519, DH_X448, or DH_single_coord_ladder, it must be applied to the subgroup, which can be done as for PRF_ODH1.
* expand square_PRF_ODH2($G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p, sqp$). says that the group G satisfies the square PRF-ODH2 assumption and the PRF-ODH2 assumption. The types prf_in and prf_out and the pseudo-random function prf are defined as for PRF_ODH1. The square PRF-ODH2 assumption means that an adversary that has a public Diffie-Hellman key $exp(g, a)$ for random a and has access to the oracle $(Y, x) \mapsto prf(exp(Y, a), x)$ cannot distinguish $x \mapsto prf(exp(g, mult(a, a)), x)$ from a random function. The probability $sqp(t, n, n')$ is the probability of breaking the square PRF-ODH2 assumption in time t with n queries to $prf(exp(g, mult(a, a)), x)$ and n' queries to $(Y, x) \mapsto prf(exp(Y, a), x)$. The probability $p(t, n, n')$ is the probability of breaking the PRF-ODH2 assumption in time t with n queries to $prf(exp(g, mult(a, b)), x)$ and n' queries to $(Y, x) \mapsto prf(exp(Y, a), x)$ and $(X, x) \mapsto prf(exp(X, b), x)$ in total. This macro defines the equivalence prf_odh(prf), whichs corresponds to the square PRF-ODH2 and PRF-ODH2 properties, for use in the crypto command (see Section 7).
When this assumption is used with DH_subgroup, DH_subgroup_with_is_neutral, DH_X25519, DH_X448, or DH_single_coord_ladder, it must be applied to the subgroup, which can be done as for PRF_ODH1.
If G is Curve448, the weak private key kp must be excluded, which can be done using DH_exclude_weak_keys, DH_X448, or DH_single_coord_ladder.
Additionally, this assumption requires that it is possible to test efficiently whether $\exp(Y, a) = \exp(g, ab)$ knowing just Y and $B = \exp(g, b)$ (so the result does not depend on a). This is possible for prime-order groups as well as Curve25519 and Curve448 when the weak private key is excluded.

The argument prf of the PRF-ODH macros is defined by these macros. It must not be declared elsewhere, and it can be used only after expanding the macro. All other arguments of these macros must be defined before expanding the macro.

– expand CDH_single($G, Z, g, exp, exp', mult, p$).
expand CDH_RSR_single($G, Z, g, exp, exp', mult, p, p_d$).
expand DDH_single($G, Z, g, exp, exp', mult, p$).
expand GDH_single($G, Z, g, exp, exp', mult, p, p_d$).
expand GDH_RSR_single($G, Z, g, exp, exp', mult, p, p_d$).
expand PRF_ODH1_single($G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p$).
expand PRF_ODH2_single($G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p, PCollKey1$).
are similar to macros with the same name without _single, except that they use a single family of exponents, a_i, instead of two, a_i and b_j. Obviously, they make no security claims on Diffie-Hellman between a_i and itself (because that is the square Diffie-Hellman property), but they guarantee security for Diffie-Hellman between a_i and a_j for any $i \neq j$. That is more powerful than the properties without _single, because it allows proving protocols that rely

on Diffie-Hellman computations between exponents in a single family, but may lead to larger probability bounds.

- `expand Xor(`$D, xor, zero$`).` defines the function symbol xor to be exclusive or on the set of bitstrings D, where $zero$ is the bitstring consisting only of zeroes in D. D should be `fixed`.

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

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

- `expand keygen(`$keyseed, key, kgen$`).` defines a key generation function $kgen$. It can be used to add a key generation function to symmetric cryptographic primitives, if needed.

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

  key type of keys, must be `bounded`.

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

  The types $keyseed$ and key must be declared before this macro is expanded. The function $kgen$ is declared by this macro. It must not be declared elsewhere, and it can be used only after expanding the macro.

  This macro defines the equivalence named `keygen`($kgen$) for use in the `crypto` command (see Section 7).

- `expand Auth_Enc_from_Enc_then_MAC(`$key, cleartext, ciphertext, enc, dec, injbot, Z, Penc, Pmac$`).` defines an authenticated encryption scheme, built by encrypt-then-MAC from an IND-CPA encryption scheme and an SUF-CMA deterministic MAC.

  The arguments are the same as for `IND_CPA_INT_CTXT_sym_enc` except that $Penc(t, N, l)$ is the probability of breaking the IND-CPA property of the underlying encryption scheme in time t for one key and N encryption queries with cleartexts of length at most l, and $Pmac(t, N, N', Nu', l)$ is the probability of breaking the SUF-CMA property of the underlying MAC scheme in time t for one key, N MAC queries, N' verification queries modified by the transformation and Nu verification queries left unchanged by the transformation for messages of length at most l.

- `expand Auth_Enc_from_AEAD(`$key, cleartext, ciphertext, enc, dec, injbot, Z, Penc, Pencctxt$`).` defines an authenticated encryption scheme, built from an AEAD scheme using empty additional data.

  The arguments are the same as for `IND_CPA_INT_CTXT_sym_enc` except that $Penc(t, N, l)$ is the probability of breaking the IND-CPA property of the underlying AEAD scheme in time t for one key and N encryption queries with cleartexts of length at most l, and $Pencctxt(t, N, N', l, l', ld, ld')$ is the probability of breaking the INT-CTXT property of the underlying AEAD scheme 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', additional data for encryption of length at most ld, and additional data for decryption of length at most ld'.

- `expand Auth_Enc_from_AEAD_nonce(`$key, cleartext, ciphertext, enc, dec, injbot, Z, Penc, Pencctxt$`).` defines an authenticated encryption scheme, built from an AEAD scheme with a nonce by choosing the nonce randomly at each encryption and using empty additional data.

  The arguments are the same as for `IND_CPA_INT_CTXT_sym_enc` except that $Penc(t, N, l)$ is the probability of breaking the IND-CPA property of the underlying AEAD scheme in time t for one key and N encryption queries with cleartexts of length at most l, and $Pencctxt(t, N, N', l, l', ld, ld')$ is the probability of breaking the INT-CTXT property of the underlying AEAD scheme 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', additional data for encryption of length at most ld, and additional data for decryption of length at most ld'.

- `expand AEAD_from_Enc_then_MAC(`key, $cleartext$, $ciphertext$, add_data, enc, dec, $injbot$, Z, $Penc$, $Pmac$`).` defines an authenticated encryption scheme with additional data built by encrypt-then-MAC from an IND-CPA encryption scheme and an SUF-CMA deterministic MAC.

  The arguments are the same as for `AEAD` except that $Penc(t, N, l)$ is the probability of breaking the IND-CPA property of the underlying encryption scheme in time t for one key and N encryption queries with cleartexts of length at most l, and $Pmac(t, N, N', Nu', l)$ is the probability of breaking the SUF-CMA property of the underlying MAC scheme in time t for one key, N MAC queries, N' verification queries modified by the transformation and Nu verification queries left unchanged by the transformation for messages of length at most l.

- `expand AEAD_from_AEAD_nonce(`key, $cleartext$, $ciphertext$, add_data, enc, dec, $injbot$, Z, $Penc$, $Pencctxt$`).` defines an authenticated encryption scheme with additional data, built from an AEAD scheme with a nonce by choosing the nonce randomly at each encryption.

  The arguments are the same as for `AEAD` except that $Penc(t, N, l)$ is the probability of breaking the IND-CPA property of the underlying AEAD scheme in time t for one key and N encryption queries with cleartexts of length at most l, and $Pencctxt(t, N, N', l, l', ld, ld')$ is the probability of breaking the INT-CTXT property of the underlying AEAD scheme 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', additional data for encryption of length at most ld, and additional data for decryption of length at most ld'.

- `expand random_split_`N`(`$input\_t, part1\_t, \ldots, partN\_t, tuple\_t, tuple, split$`).` defines allows to split a random value into N values, for $N \leq 10$.

  $input\_t$: type of the input value

  $part1\_t, \ldots, partN\_t$: types of the output parts.

  $tuple\_t$: type of a tuple of the output parts

  $tuple(part1\_t, \ldots, partN\_t) : tuple\_t$: builds a tuple from N parts.

  $split(input\_t) : tuple\_t$ splits the input into N parts and returns a tuple of these parts. The macro says that if y is a random value in $input\_t$, then $split(y)$ is a tuple $tuple(x1, \ldots, xN)$ of N independent random values in $part1\_t, \ldots, partN\_t$.

  To split a value y of type $input\_t$ into N parts of types $part1\_t, \ldots, partN\_t$, write:

  $$\texttt{let}\ tuple(x1, \ldots, xN) = split(y)\ \texttt{in}\ \ldots$$

  Note that a priori, CryptoVerif thinks that the pattern-matching with $tuple(x1, \ldots, xN)$ may fail, and thus requires an `else` branch when the `let` occurs in a term. CryptoVerif realizes that the pattern-matching never fails when it expands the definition of $split$.

  This macro defines the equivalence named `splitter(`$split$`)` which replaces the splitting of a random number generation in $input\_t$ with N independent random number generations in $part1\_t, \ldots, partN\_t$.

  $input\_t$, $part1\_t, \ldots, partN\_t$, $tuple\_t$ must be defined before expanding this macro. $tuple$ and $split$ are defined by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

7 Interactive Mode

In interactive mode, the user specifies transformations to perform. Some of the instructions take a program point (or occurrence) in the current game as argument. One should use the command `show_game occ` or `out_game` f `occ` (mentioned below) to display the game with an occurrence number at each program point. The program points can then be specified as follows:

- an integer designates the program point labeled by that integer in the displayed game.

- `before` "*regexp*" designates the program point at the beginning of the line that matches the regular expression *regexp*. Regular expressions follow the syntax of regular expressions in the OCaml Str module, see `https://ocaml.org/releases/4.14/api/Str.html`. In regular expressions, backslash (\) must be escaped as \\, as in OCaml string literals. There must be a single line that matches this regular expression, otherwise CryptoVerif shows an error message.

- `after` "*regexp*" designates the program point at the beginning of the first line that has an occurrence number after the line that matches the regular expression *regexp*. There must be a single line that matches this regular expression, otherwise CryptoVerif shows an error message.

- `before_nth` n "*regexp*" designates the program point at the beginning of the n-th line that matches the regular expression *regexp*.

- `after_nth` n "*regexp*" designates the program point at the beginning of the first line that has an occurrence number after the n-th line that matches the regular expression *regexp*.

- `at` n' "*regexp*" designates the program point at the n'-th occurrence number that occurs inside the string that matches the regular expression *regexp* in the displayed game. There must be a single match for this regular expression, otherwise CryptoVerif shows an error message. (With `at`, if the same line matches the regular expression several times, it counts as several matches.)

- `at_nth` n n' "*regexp*" designates the program point at the n'-th occurrence number that occurs inside the string corresponding to the n-th match of the regular expression *regexp* in the displayed game. (With `at_nth`, if the same line matches the regular expression several times, it counts as several matches.)

With `before`, `after`, `before_nth`, and `after_nth`, the match is performed on a game in which only occurrences of processes are displayed, at the beginning of lines. Therefore, the occurrence numbers typically do not appear in the regular expression given by the user, provided the regular expression does not require explicitly matching at the beginning of the line (i.e., the regular expression should not use `^`). In contrast, with `at` and `at_nth`, the match is performed on a game in which all occurrence numbers are displayed. The regular expression needs to match at least one occurrence number. Any occurrence number can be matched by the regular expression `{[0-9]+}`.

Using an explicit integer to designate a program point is very unstable: it changes if the verified protocol is slightly modified, or if a new version of CryptoVerif itself is used, which may transform games in a slightly different way. The other ways of designating program points are therefore preferable when possible.

When an identifier is expected in an instruction, it is possible to put it between quotes. This is useful in particular for identifiers that clash with proof keywords.

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 findcond`: removes useless assignments, as above, as well as assignments `let` $x = M$ `in` inside conditions of `find`.

- `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` $x_1 \ldots x_n$: remove assignments to $x_1, \ldots, x_n$ by replacing x_i with its value. When x_i becomes unused, its definition is removed. When x_i is used only in `defined` tests after transformation, its definition is replaced with a constant. The variables x_i may also be regular expressions, following the syntax of regular expressions in the OCaml Str module, see `https://ocaml.org/releases/4.14/api/Str.html`. In this case, they designate all variables that match the regular expression. This is particularly helpful to designate all variables that come from the same initial name but have different numbers: "*name*`_[0-9]*`". Regular expressions need to be put between quotes because they use characters that do not belong to ordinary identifiers. Blackslash (\) must then be escaped as \\, as in OCaml string literals.

- `use_variable` $x_1 \ldots x_n$: when x_i is defined by an assignment `let` $x_i = M_i$, replace all occurrences of term M_i at which x_i is guaranteed to be defined by x_i. The replacement is also performed with array accesses, that is, $M_i\{\widetilde{M}/\widetilde{i}\}$ is replaced with $x_i[\widetilde{M}]$, when $x_i[\widetilde{M}]$ is guaranteed to be defined, where $\widetilde{i}$ are the current replication indices at the definition of x_i. As in the command `remove_assign binder` $x_1 \ldots x_n$, the variables x_i may also be regular expressions, designating all variables that match the regular expression.

- `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` $x_1 \ldots x_n$: move random number generations and assignments that define $x_1, \ldots, x_n$ (even when this is not beneficial). The variables x_i may also be regular expressions.
  - `array` x [`"`*exp*`", ..., "`*exp*`"`]: move random number generations that define x when x is of a `bounded` or `nonuniform` 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.)

    The expressions `"`*exp*`"` allow the user to specify expressions that do no require the generation of x when it has not been generated before, because the expression always yields the same result when x is a fresh random value, up to negligible probability. More precisely, these expressions must be of the form

    [`forall` seq⟨vartype⟩`;`] `new` ⟨ident⟩`:`⟨ident⟩`;` ⟨simpleterm⟩

    The expression can be `forall` $y_1 : T_1, \ldots, y_n : T_n$`;` `new` $x' : T$`;` M, where T is the type of x and $y_1, \ldots, y_n, x'$ are the variables of M. CryptoVerif tries to simplify M into a term M' that does not contain x', assuming that x' is random and $y_1, \ldots, y_n$ are independent of x'. If it fails, the `move array` transformation fails. If it succeeds, the transformation can be performed, replacing $M\{x/x'\}$ with M' instead of generating a fresh x.

    When no expression `"`*exp*`"` is mentioned, the expressions that do no require the generation of x are equality tests with x, and the type T of x must be large enough, so that collisions between x and a value independent of x can be eliminated ($\texttt{Pcoll1rand}(T) \leq 2^{-n'}$, that is, T has option $\texttt{pcoll}n$ with $n \geq n'$ where n' is set by `set minAutoCollElim` $=$ `pest`n'; the default is $n' = 80$).

The variables x may also be a regular expression. In this case, it designates all variables that match the regular expression; all these variables must have the same type.

- `simplify`: simplify the game.

- `simplify coll_elim(variables:` $x_1, \dots, x_n$`; types:`$t_1, \dots, t_{n'}$`; terms:`$occ_1, \dots, occ_{n''}$`)`: simplify the game, additionally allowing elimination of collisions on data at all occurrences of variables $x_1, \dots, x_n$, at all data of types $t_1, \dots, t_{n'}$, and at the program points $occ_1, \dots, occ_{n''}$. See above for how to specify the program points occ_i. Some of the lists of variables, types, or terms may be omitted. In this case, the separating semi-colon `;` is obviously omitted as well. It is also possible to reorder or repeat these lists; the lists add up. (The probability of the collision must still satisfy the condition given by `allowed_collisions`.)

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

  `global_dep_anal` x `coll_elim(variables:` $x_1, \dots, x_n$`; types:`$t_1, \dots, t_{n'}$`; terms:`$occ_1, \dots, occ_{n''}$`)` performs global dependency analysis on x, additionally allowing elimination of collisions on data at all occurrences of variables $x_1, \dots, x_n$, at all data of types $t_1, \dots, t_{n'}$, and at the program points $occ_1, \dots, occ_{n''}$. See above for how to specify the program points occ_i. Some of the lists of variables, types, or terms may be omitted. In this case, the separating semi-colon `;` is obviously omitted as well. It is also possible to reorder or repeat these lists; the lists add up. (The probability of the collision must still satisfy the condition given by `allowed_collisions`.)

  One must allow elimination on x independently of the program point, so if x is not large, x must be mentioned in $x_1, \dots, x_n$ or its type must be mentioned in $t_1, \dots, t_n$; mentioning the occurrences of x in $occ_1, \dots, occ_{n''}$ is not sufficient.

  The variable x may also be a regular expression. In this case, it designates all variables that match the regular expression, and the command `global_dep_anal` is executed for each of these variables in turn.

- `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. The variable x may also be a regular expression. In this case, it designates all variables that match the regular expression, and the command `SArename` is executed for each of these variables in turn.

- `all_simplify`: perform several simplifications on the game, as if

  – `simplify`,

  – `move all` if `autoMove = true`,

  – `remove_assign useless` if `autoRemoveAssignFindCond = false`,
  `remove_assign findcond` if `autoRemoveAssignFindCond = true`,

  – and `merge_branches` if `autoMergeBranches = true`

  had been called.

- `expand`: expand `if`, `let`, `find`, `event`, `event_abort`, `new` terms into processes. That leads to distinguishing the branches until the end of the process, which may help the proof by distinguishing more cases, but may lead to very large games. This is also needed because some game transformations of CryptoVerif do not support non-expanded games. When `autoExpand = true` (the default), this expansion is performed automatically in case a game transformation results in a non-expanded game.

- `crypto` ⟨crypto_args⟩: applies a cryptographic transformation that comes from a statement `equiv`. This command can have two forms:

  `crypto`: list all available `equiv` statements, and ask the user to choose which one should be applied, with which special arguments when the chosen equivalence is generated by `equiv` ... `special`, and with which ⟨info⟩ as described below.

  `crypto` ⟨name⟩ [`special(`seq⟨specialarg⟩`)`] ⟨info⟩: apply a cryptographic transformation determined by the name ⟨name⟩, where:

  – ⟨name⟩ can be either an identifier *id* or *id*(f), and corresponds to the name given at the declaration of the cryptographic transformation by `equiv`. In case the name is not found, CryptoVerif reverts to the old way of designating cryptographic transformations, in which ⟨name⟩ can be either a function symbol that occurs in the terms in the left-hand side of the `equiv` statement, or a probability function that occurs in the probability formula of the `equiv` statement. When several equivalences correspond, the user is prompted for choice.

  – `special(`seq⟨specialarg⟩`)`, when present, passes the given special arguments to the generation of the chosen equivalence, which must be defined by `equiv` ... `special`. The exact meaning of the arguments depend on the considered special equivalence.

    1. For the special equivalences `rom`, `prf`, `prp`, `sprp`, and `icm` there must be at most one special argument, and when one argument is present, it overrides the *collisions_LHS* argument of the special equivalence.
    2. For the special equivalences `rom_partial`, `prf_partial`, `prp_partial`, `sprp_partial`, and `icm_partial` there must be at most two special arguments, and when such arguments are present, one of them overrides the *collisions_LHS* argument of the special equivalence, and the other one determines the collision matrix between oracles. CryptoVerif determines which argument is which based on their type (tuple of strings for *collisions_LHS*, one string for the collision matrix).

    See `equiv` ... `special` for more information.

  – ⟨info⟩ can be

    1. `*`: The transformation is applied as many times as possible. (In this case, the advised transformations are applied automatically even when `set autoAdvice = false`.)
    2. `**`: Similar to `*`, but the game is simplified only after the last cryptographic transformation instead of simplifying it after each transformation, for faster execution. This is recommended only for very simple cryptographic transformations.
    3. $x_1 \ \dots \ x_n$: apply the cryptographic transformation, where $x_1, \dots, x_n$ are variable names of the game corresponding to random number generations in the left-hand side of the equivalence. (CryptoVerif may automatically add variables to the list $x_1, \dots, x_n$ if needed, except when a dot is added at the end of the list $x_1, \dots, x_n$. The transformation is applied only once. If several disjoint lists $x_1, \dots, x_n$ are possible and no variable name is mentioned, CryptoVerif makes a choice. It is better to mention at least one variable name when the left-hand side of the equivalence contains a random number generation, to make explicit which transformation should be applied.)
       In case the command ends with a dot (`.`), CryptoVerif never adds other variable names to those already listed. If the dot is absent, CryptoVerif may add other variable names if that seems necessary to perform the transformation.
       The variables x_i may also be regular expressions. In this case, they designate all variables that match the regular expression.
    4. `[variables:` x_1`->`$y_1, \dots, x_n$`->`y_n`; terms:` occ_1`->`$O_1, \dots, occ_m$`->`O_m`]`: apply the cryptographic transformation, where
       (a) $x_1, \dots, x_n$ are variable names of the game which correspond to random number generations $y_1, \dots, y_n$ respectively in the left-hand side of the equivalence. (CryptoVerif may automatically add variables to the list x_1`->`$y_1, \dots, x_n$`->`y_n if needed, except when a dot is added at the end of this list.)

The variables x_i may also be regular expressions. In this case, they designate all variables that match the regular expression, and they are mapped to the same variable y_i in the equivalence.

(b) $occ_1, \dots, occ_m$ are program points at which terms will be transformed using oracles $O_1, \dots, O_m$ respectively of the equivalence. See above for how to specify the program points occ_i. (CryptoVerif may automatically add elements to the list occ_1->$O_1, \dots,$ occ_m->O_m if needed, except when a dot is added at the end of this list.)

When the considered equivalence is defined inside a macro, macro expansion may add an integer suffix `_`k to the variable and oracle names of the equivalence (or may modify that suffix if they already have one). This suffix must be included in the variable and oracle names used in this command. This happens in particular for primitives defined in the library of primitives of CryptoVerif. The right value of k in the suffix can be determined by issuing a command `crypto` without further indication. This command will display the equivalences as they are stored by CryptoVerif after macro expansion. It can also be determined using the commands `show_equiv` and `out_equiv`.
One of the lists of variables or terms may be omitted. In this case, the separating semicolon `;` is obviously omitted as well. It is also possible to reorder or repeat the `variables` and /or `terms` lists; the lists add up.

- `insert_event` *e occ* replaces the subprocess or term at program point *occ* with the event `event_abort` *e*. The games may be distinguished if and only if the event *e* is executed, and CryptoVerif then tries to find a bound for the probability of executing that event. See above for how to specify the program point *occ*. The program point *occ* must correspond to an output process (resp. oracle body in the oracles front-end) or to a term not in a condition of `find` nor in the channel of an input.

  When the setting `autoExpand` is true and the occurrence *occ* corresponds to a term, the game is automatically expanded after inserting the event, so that after expansion the event occurs in a process, not in a term.

- `insert` *occ* `"`⟨ins⟩`"` inserts instruction ⟨ins⟩ at program point *occ*. The instruction ⟨ins⟩ can be

  `new` ⟨ident⟩`:`⟨ident⟩[`;`⟨ins⟩]
  ⟨ident⟩ `<-R` ⟨ident⟩[`;`⟨ins⟩]
  `event_abort` ⟨ident⟩
  `if` ⟨cond⟩ `then` ⟨ins⟩ [`else` ⟨ins⟩]
  `let` ⟨pattern⟩ `=` ⟨term⟩ `in` ⟨ins⟩ [`else` ⟨ins⟩]
  ⟨basicpat⟩ `<-` ⟨term⟩[`;`⟨ins⟩]
  `find`[`[unique]`] ⟨findbranch⟩ (`orfind` ⟨findbranch⟩)* [`else` ⟨ins⟩]
  `(`⟨ins⟩`)`
  (empty)

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

  The empty instruction is replaced by the code that follows the insertion point. In all cases except `event_abort` *e*, the code that follows the insertion point is executed after the inserted instruction. The probability of distinguishing the game before insertion from the game after insertion is then at most of the probability of the events `event_abort` *e* in the inserted instruction. CryptoVerif then tries to bound this probability. This is similar to `insert_event` *e occ*. However, the `insert` command does not subsume `insert_event`, because `insert` allows inserting only at process points while `insert_event` allows inserting events at term points as well.

  The instruction `(`⟨ins⟩`)` is equivalent to ⟨ins⟩. The parentheses just help resolving ambiguities of the grammar, for instance to specify to which process `else` branches are attached.

  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 autoMergeBranches = false` when a case distinction is inserted, so that the operation is not immediately undone at the next simplification.

In contrast to the initial game, the terms `event`, `event_abort`, `get`, `insert`, `new`, `if`, `find`, or `let` are not expanded, so terms `if`, `find`, `let` and its synonym `<-` can occur only in conditions of `find`; `event`, `event_abort`, `get`, `insert`, `new` and its synonym `<-R` 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 $x[M_1, \dots, M_n]$ under a `defined` condition that contains $x[M_1, \dots, M_n]$ as a subterm). In case the inserted instruction is not appropriate, an error message explaining the problem is displayed.

See above for how to specify the program point *occ*. The program point *occ* must correspond to an output process (resp. oracle body in the oracles front-end).

- `replace` *occ* "*term*" replaces the term at program point *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`, `event`, `event_abort`, `insert`, `get`. See above for how to specify the program point *occ*. The program point *occ* must correspond to a term not containing `if`, `let`, `find`, `new`, `event`, `event_abort`, `insert`, `get`.
- `assume replace` *occ* "*term*" does the same thing as `replace` *occ* "*term*", but does not check that the term at program point *occ* is equal the term *term*, hence the replacement may not be correct. This command is present only to allow testing whether a proof would succeed if some replacement could be done. If you make a proof by relying on this command, CryptoVerif still considers that the query is not proved.
- `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` $x_{11} \dots x_{1n}$ `,` $\dots$ `,` $x_{k1} \dots x_{kn}$ takes as argument k lists of n variables separated by commas. It merges the variables $x_{i1}, \dots, x_{in}$ 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.
- `guess` ⟨guessspec⟩ guesses the value of ⟨guessspec⟩, where ⟨guessspec⟩ can be one of the following:
  1. i, where i is a replication index: one guesses the value of the replication index i in the tested session. (The other sessions still exist, but one does not try to prove queries for them.) There must be a single replication with index i in the game.

2. *occ*, where *occ* is the occurrence of a replication; then one guesses the value of the replication index i of that replication in the tested session.
3. i `&& above`, where i is a replication index; then, just under the replication with index i, one guesses the value of the whole sequence of replication indices above the replication with index i in the tested session. There must be a single replication with index i in the game.
4. *occ* `&& above`, where *occ* is the occurrence of a replication; then, just under that replication, one guesses the value of the whole sequence of replication indices above that replication in the tested session.
5. "$x[i_1, \dots, i_n]$", where $x[i_1, \dots, i_n]$ is the variable to be guessed, defined under n replications, and $i_1, \dots, i_n$ are constant replication indices (typically produced by a previous guess of the replication indices). CryptoVerif must be able to determine, at each definition of x, whether its indices will be equal to $i_1, \dots, i_n$ or not. Otherwise, the transformation fails. When x is defined under no replication, one writes x instead of "$x[i_1, \dots, i_n]$".

CryptoVerif distinguishes whether the element specified by ⟨guessspec⟩ is equal to a constant value v_{tested} by introducing a test under the definition of that element, and tries to prove the security properties for ⟨guessspec⟩ $= v_{tested}$. The queries are adjusted accordingly. The probability of breaking the initial query is typically the size of the guessed element ⟨guessspec⟩ (e.g. N when we guess a replication index i in $[1, N]$; $\#O$ when we guess the whole sequence of indices above a replication just above oracle O; $|T|$ when we guess a variable x of type T) times the probability of breaking the query for ⟨guessspec⟩ $= v_{tested}$. The size of the guessed element must be estimated less than `maxGuess` (see the command `set maxGuess`).

Guessing the value of replication indices cannot apply to injective correspondences. When we guess replication indices and injective correspondences are present, CryptoVerif tries to prove injectivity, so that only a non-injective correspondence remains to prove. It that fails, injective correspondences are left unchanged and proved on all sessions. If all queries are injective correspondences for which that fails, the transformation fails.

This transformation does not apply to `equivalence` and `query_equiv` proofs. The transformation fails if such a proof is still required.

- `guess_branch` *occ* guesses which branch is taken at program point *occ*. The instruction at program point *occ* must be a test (`if`, `let`, or `find`) with at least two branches. This instruction must be executed at most once (either because it is not under replication, and because it is under replication and the replication indices are fixed, for instance because they have previously been guessed by the `guess` instruction above). If this instruction has n branches, CryptoVerif generates n games G_i $(0 \leq i < n)$ in which branch i is kept and all other branches are replaced with an event `bad_guess`. (Branch 0 is the `else` branch.) The desired queries must then be proved in all games G_i and the probability of breaking the queries in the initial game is bounded by the sum of the probabililities of breaking them in all games G_i. That allows the user to split cases depending on which branch is taken at program point *occ*.

  The user first needs to prove the queries in game G_0. Once all queries are proved in this game, CryptoVerif automatically goes back and asks the user to prove the queries in next game, until they are proved in all games G_i.

  In case the user is unable to prove some queries in a game G_i, it is blocked: it cannot consider the following games. In that case, the best is probably to remove the queries that cannot be proved and restart the proof. One can also undo the transformations until before `guess_branch`, use `focus` to limit oneself to the queries that can be proved and redo the proof from `guess_branch`.

- `start_from_other_end`: for proofs of indistinguishability only (`equivalence`), instruct CryptoVerif to start transforming from the other game. When your input file contains `equivalence` Q_1 Q_2, CryptoVerif initally works on the first process Q_1. When you issue the command `start_from_other_end`, CryptoVerif stores your current state, and starts working from Q_2. If you issue `start_from_other_end` again, it will store what you did from Q_2, and will restart working from the end of the sequence that you built from Q_1. This command allows you to alternate between the sequence that starts from Q_1 and the one that starts from Q_2. The property is proved when both sequences end with the same game (which you can check with the command `success`, as usual).

- `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.
- `success simplify`: run `success` then `simplify`, with the following addition. The command `success` collects information that is known to be true when the adversary manages to break at least one of the desired properties. The first iteration of `simplify` removes parts of the game that contradict this information and replaces them with `event adv_loses`.
- `show_game`: display the current game.
- `show_game occ`: display the current game with occurrence numbers. Useful for some commands that require specifying a program point; see above for how program point are specified.
- `show_state`: display the whole sequence of games until the current game.
- `show_facts` *occ*: show the facts that are proved by CryptoVerif in the current game, at the program point *occ*. See above for how to specify the program point *occ*. This command is mainly helpful for debugging.
- `show_equiv` ⟨crypto_args⟩: display the game equivalence corresponding to the cryptographic transformation specified by ⟨crypto_args⟩. These arguments are the same as for the `crypto` command. This command is useful to determine the exact variable and oracle names of an equivalence and to examine and possibly modify equivalences generated by `equiv` ... `special`.
- `show_commands`: display the interactive commands executed so far (or from the last change of output file by `out_commands`).
- `out_game` *f*: output the current game to file *f*. By default, *f* is output in the current directory. If the command-line option `-o` *directory* was given, *f* is output in the given directory. Only the digits, ascii letters, and `%+-.=@_~` are allowed in the filename *f*. The dot (`.`) is not allowed as first character. (Be careful: file *f* will be overwritten if it already exists.)
- `out_game` *f* `occ`: output the current game with occurrence numbers to file *f*. Useful for some commands that require specifying a program point; see above for how occurrences are specified. (See command `out_game` for details on the filename *f*. Be careful: file *f* will be overwritten if it already exists.)
- `out_state` *f*: output the whole sequence of games until the current game to file *f*. (See command `out_game` for details on the filename *f*. Be careful: file *f* will be overwritten if it already exists.)
- `out_facts` *f occ*: output the facts that are proved by CryptoVerif in the current game, at the program point *occ*, to file *f*. See above for how to specify the program point *occ*. This command is mainly helpful for debugging. (See command `out_game` for details on the filename *f*. Be careful: file *f* will be overwritten if it already exists.)
- `out_equiv` *f* ⟨crypto_args⟩: output the game equivalence corresponding to the cryptographic transformation specified by ⟨crypto_args⟩ to the file *f*. The arguments ⟨crypto_args⟩ are the same as for the `crypto` command. This command is useful to determine the exact variable and oracle names of an equivalence and to examine and possibly modify equivalences generated by `equiv` ... `special`. (See command `out_game` for details on the filename *f*. Be careful: file *f* will be overwritten if it already exists.)
- `out_commands` *f*: output the executed interactive commands to file *f*. If no output file was specified before, outputs both the previous and future interactive commands to *f*. If an output file was specified before (by the command-line setting `-ocommands` or by a previous command

out_commands), the previous commands are output to the previously specified file, and the future commands are output to f. If f is the empty string "", stops outputting interactive commands. (See command out_game for details on the filename f. Be careful: file f will be overwritten if it already exists.)

- 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 determines when to eliminate collisions. This command has two variants:
  - allowed_collisions ⟨formulas⟩: ⟨formulas⟩ is a comma-separated list of formulas of the form ⟨psize⟩$_1$^n_1 * ... *⟨psize⟩$_k$^n_k/⟨pest⟩, where the exponents n_i can be omitted when equal to 1, writing ⟨psize⟩$_i$ instead of ⟨psize⟩$_i$^1, and the whole factor ⟨psize⟩$_1$^n_1 * ... *⟨psize⟩$_k$^n_k is replaced with 1 when there is no ⟨psize⟩$_i$ factor at all; ⟨psize⟩$_i$ is an identifier that determines the size of a parameter: size$n$ for parameters of size n, meaning that the parameter is at most 2^n, small for size 2, passive or default for size 30, noninteractive for size 80; ⟨pest⟩ (Probability ESTimate) is an identifier such that 1/⟨pest⟩ estimates a probability. It can take the following values: pest$n$ means that the probability 1/⟨pest⟩ is at most 2^{-n}; password is equivalent to pest20, i.e. the probability 1/⟨pest⟩ is at most 2^{-20}; large is equivalent to pest160, i.e. the probability 1/⟨pest⟩ is at most 2^{-160}. (See also the declarations param, proba, and type for explanations of these estimates.)

    Collisions are eliminated when, for some formula ⟨psize⟩$_1$^n_1 * ... *⟨psize⟩$_k$^n_k/⟨pest⟩ in the list ⟨formulas⟩, their probability is at most of the form $\mathit{constant} \times p_1^{n_1} \times \cdots \times p_k^{n_k} \times \mathit{proba}_0$, where p_i is a parameter of size at most ⟨psize⟩$_i$ and the estimate of the probability proba_0 is at most 1/⟨pest⟩. This condition is applied both for collisions between elements of a type T, in which case $\mathit{proba}_0 = \texttt{Pcoll1rand}(T)$, and for collision statements, in which case proba_0 is the probability given in the collision statement. When the list of formulas ⟨formulas⟩ is empty, elimination of collisions is entirely disabled.

    By default, collisions are eliminated for $\mathit{anything} \times \mathit{proba}_0$ when $\mathit{proba}_0 \leq 2^{-155}$ (where large would mean 2^{-160}, to allow collisions that have probability a small factor times collisions between elements of a large type), and for $p \times \mathit{proba}_0$ when $p \leq 2^2$ (small) and $\mathit{proba}_0 \leq 2^{-20}$ (password). The default behavior is inspired by what happens in asymptotic security: large means that the probability of collision is asymptotically negligible, while the parameters are always polynomial, so $\mathit{constant} \times p_1^{n_1} \times \cdots \times p_k^{n_k} \times \texttt{Pcoll1rand}(T)$ is negligible when T is large. Similarly, probabilities given in collision statements are always negligible (probabilities are declared large by default), while the parameters are always polynomial, so the probability obtained by applying $\mathit{constant} \times p_1^{n_1} \times \cdots \times p_k^{n_k}$ times a collision statement remains negligible.
  - allowed_collisions ⟨pest⟩, where ⟨pest⟩ estimates a probability: pest$n$ means that the probability is at most 2^{-n}; password is equivalent to pest20, i.e. probability at most 2^{-20}; large is equivalent to pest160, i.e. probability at most 2^{-160}. Collisions are eliminated when their probability, taking into account how many times they are applied, is at most the probability specified by ⟨pest⟩. This behavior fits the exact security framework nicely: we eliminate collisions when they have a small enough probability.
- focus "⟨querydecl⟩", ..., "⟨querydecl⟩" where ⟨querydecl⟩ ::= **query** [seq⟨vartypeb⟩;]⟨query⟩(; ⟨query⟩)* follows the syntax of query declarations given in Section 3 without the final dot: tell CryptoVerif to try to prove only the mentioned queries, ignoring all other queries. That sometimes allows to simplify the game further (e.g. remove events that are not used in the queries on which we focus), and may allow to prove the mentioned queries. The queries are considered equal modulo renaming of variables declared in seq⟨vartypeb⟩. When there is no ambiguity, the public variables of the queries can be omitted. When the queries on which we focus are all proved, CryptoVerif goes back to the state before the last focus command, to try to prove the other queries. undo focus also goes back to the state before the last focus command, to try to prove remaining queries.
- tag t: tag the current state with tag t (which can be an identifier or a string). This is useful to mark the current state and be able to go back to that state with the command undo t.

- `undo`: undo the last transformation.
- `undo` n: undo the last n transformations.
- `undo focus`: go back to the state before the last `focus` command.
- `undo` t: undo the transformations until the last state tagged t.
- `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.
- `forget_old_games`: removes games before the current one from memory. That allows to save some memory, but prevents `undo` and `undo` n. However, tagged states are not removed from memory, so that the command `undo` t where t is a tag still works. Similarly, states before `focus` commands are not removed from memory, so that the command `undo focus` still works. The display of the games is saved into a temporary file to allow displaying the games at the end of the proof. You can save more memory by applying this command systematically with the setting `set forgetOldGames = true`.

Ctrl-C allows to interrupt a command in interactive mode, and go back to the state before the beginning of this command. This feature can be helpful when a command is very slow, to be able to try another command without waiting for the current command to terminate. It may not work under Windows.

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.

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

8 Output of the system

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

- When a step of the transformations transforms the same part of the game as a previous step, the program point in the second step actually refers to the code generated by the previous step, so it is not found in the previously displayed game.
- When a step transforms part of the game that was duplicated by a previous step of the transformation, the displayed program point is in fact ambiguous: one does not know which of the copies is actually transformed.

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

Lines that begin with `RESULT` give the proved results. They may indicate that a property is proved and give an upper bound of the probability that the adversary breaks the property. These probabilities use the same notations are probabilities given as input for CryptoVerif, with the following addition: in the channel front-end, `#`c designates the total number of inputs performed on channel c. This notation mimics the notation #O for the number of oracles calls in the oracle front-end. The notation #c is used only when the channel names satisfy the same constraints as those required for oracle names: inputs on the same channel c must not occur on both sides of a parallel composition $Q \mid Q'$; an input on channel c must not occur under an input on the same channel c in the syntax tree; and inputs in different branches of `if`, `find`, `let`, `get` must have compatible structures (inputs on the same channel must occur at the same place in the sequence of possible inputs). These constraints make sure that the usage of #c does not mix unrelated inputs. When they are not satisfied, a warning is displayed and CryptoVerif uses only replication bounds for counting inputs.

In the end, they may also list the properties that could not be proved, if any.

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

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

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

9 Implementation

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

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

- For the cryptographic primitives, one can specify which OCaml function corresponds to which CryptoVerif function as explained in Section 9.3 below. For the security guarantees to hold, the OCaml implementation must satisfy the security assumptions mentioned in the CryptoVerif specification. The subdirectory `implementation` provides a basic implementation for some cryptographic primitives, in the module `Crypto`. This module has two implementations:
  - `crypto_real.ml` corresponds to real cryptographic primitives, implemented by relying on the OCaml cryptographic library `Cryptokit` (`http://forge.ocamlcore.org/projects/cryptokit/`). You need to install this library in order to run the protocol implementations generated by CryptoVerif. (It is used at least for random number generation even if you implement the cryptographic primitives by other means.)
  - `crypto_dbg.ml` is a debugging implementation, which constructs terms instead of applying the real cryptographic primitives.

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

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

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

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

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

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

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

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

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

9.1 Restrictions on the processes for implementation

The following two constraints must be satisfied:

- `find` must not be used. You can obtain a similar result using `insert` and `get`, which are supported.
- Let us name "oracles" the parts of the process that are between an `in`/⟨ident⟩`(`seq⟨pattern⟩`) := ...` and an `out`/`return` statement, because in the oracle frontend, they correspond exactly to that.

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

  - the type $T_1 \times \ldots \times T_k \rightarrow T'_1 \times \ldots \times T'_n$, where $T_1 \times \ldots \times T_k$ are the types of the arguments expected in the `in`/⟨ident⟩`(`seq⟨pattern⟩`) :=` statement, and $T'_1 \times \ldots \times T'_n$ are the types of the result given in the `out`/`return` statements, and

$\langle \text{mod\_opt} \rangle ::= \langle \text{ident} \rangle (\texttt{<} \mid \texttt{>}) \langle \text{string} \rangle$

$\langle \text{odef} \rangle :: += \langle \text{ident} \rangle [\texttt{[}\ \text{seq}^+ \langle \text{mod\_opt} \rangle\ \texttt{]}]\ \texttt{\{}\ \langle \text{odef} \rangle$

If channel frontend, $\langle \text{obody} \rangle :: += \texttt{out(}\langle \text{channel} \rangle\texttt{,}\ \langle \text{term} \rangle\texttt{)}[\texttt{\}}][\texttt{;}\ \langle \text{odef} \rangle]$

If oracle frontend, $\langle \text{obody} \rangle :: += \texttt{return(}\text{seq}\langle \text{term} \rangle\texttt{)}[\texttt{\}}][\texttt{;}\ \langle \text{odef} \rangle]$

Figure 9: Extensions to the syntax

- the list containing for each of the following oracles, its name and whether it is under a replication or not.

An oracle can have multiple `out`/`return` statements. To be able to implement it, we must be able to define the signature above for each oracle, that is, all `out`/`return` must return the same type of elements, and the oracles present after each `out`/`return` statement must be the same. Moreover, if an oracle with the same name is defined at several places, all its definitions must have the same signature.

9.2 Defining modules

The syntax of the processes is extended to add annotations, described in Figure 9. The symbol $:: +=$ means that we add the rule at the right-hand side to the non-terminal symbol at the left-hand side.

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

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

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

9.3 Implementation options

The implementation options declares how the implementation should translate functions, tables and types, and one must declare them after the declaration of the element it modifies and before use. The syntax is described in Figure 10.

The available implementation options are described hereafter:

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

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

seq;$^+$⟨N⟩ ::= N | N;seq;$^+$⟨N⟩

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

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

⟨fun_opt⟩ ::= ⟨ident⟩=⟨string⟩

⟨impl_opt⟩ ::= `type` ⟨ident⟩=⟨string⟩ [[seq;$^+$⟨type_opt⟩]]
| `type` ⟨ident⟩=⟨integer⟩ [[seq;$^+$⟨type_opt⟩]]
| `table` ⟨ident⟩=⟨string⟩
| `fun` ⟨ident⟩=⟨string⟩ [[seq;$^+$⟨fun_opt⟩]]
| `const` ⟨ident⟩=⟨string⟩

Figure 10: Grammar for implementation options

- `serial="s","d"`: Sets the serialization/deserialization of the type. There is no default, and this is required when a variable of type T is written or read to a file/table, or when it is contained in a tuple. The serialization function `s` must be of type `ty` $\rightarrow$ `string`, the deserialization function `d` must be of type `string` $\rightarrow$ `ty`. When deserialization fails, it must raise exception `Match_fail`.
- `pred="p"`: Sets the predicate function, this function must be an OCaml function of type `ty` $\rightarrow$ `bool`. It returns whether an element is of type T or not. The default predicate function is a function that accepts every element.
- `random="f"`: Sets the random generation function. This function must be an OCaml function of type `unit` $\rightarrow$ `ty`, and must return uniformly a random element of type `ty`. In particular, if a predicate function has been defined, the predicate function must return `true` on every element returned by the random generation function.

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

CryptoVerif allows one to define macros by `letfun`. Specifying an OCaml implementation for these macros is optional. When the OCaml implementation is not specified, CryptoVerif generates code according to the `letfun` macro. When the OCaml implementation is specified, it is used when generating the OCaml code, while the CryptoVerif macro defined by `letfun` is used for proving the protocol. This feature can be used, for instance, to define probabilistic functions: the OCaml implementation generates the random choices inside the function, while the CryptoVerif definition by `letfun` first makes the random choices, then calls a deterministic function.

- `const` f=`"s"`: Sets the implementation of the function f that has no arguments to an OCaml constant. If the constant is a string, one can write, for example, `const` f=`"\"constant\""`.

10 Additional programs

10.1 `test`

Usage:

`test` [`-timeout` ⟨n⟩] ⟨mode⟩ ⟨test_set⟩

where `-timeout` ⟨n⟩ sets the timeout for each execution of the tested program to n seconds (by default, there is no timeout), ⟨mode⟩ can be:

- `test`: test the mentioned scripts
- `test_add`: test the mentioned scripts and add the expected result in the script when it is missing
- `add`: add the expected result in the script when it is missing, do not test scripts that already have an expected result
- `update`: test the mentioned scripts and update the expected result in the script

and ⟨test_set⟩ can be:

- `basic` runs basic CryptoVerif tests
- `big` runs bigger CryptoVerif examples
- `proverif` runs ProVerif on tests suitable for it
- `converted` runs CryptoVerif on examples converted from CryptoVerif 1.28
- `impl` runs tests of the generation of OCaml implementations
- `all` runs all tests included in `basic`, `proverif`, `converted`, `big`, and `impl`
- `dir` ⟨prefix⟩ ⟨list_of_directories⟩ analyzes the mentioned directories using CryptoVerif, using ⟨prefix⟩ as prefix for the output files.

⟨test_set⟩ can be omitted when it is `basic`, and ⟨mode⟩ ⟨test_set⟩ can both be omitted when they are `test basic`.

The script `test` is a bash shell script, so you must have bash installed. On Windows, the best is to install Cygwin and run `test` from a Cygwin terminal.

The script `test` must be run in the CryptoVerif main directory; the programs `analyze` and `cryptoverif` must be present in that directory.

For CryptoVerif tests, the programs first runs the script `prepare` in each directory when it is present. That allows for instance to generate the CryptoVerif scripts to run. Then it runs the program `analyze` described below.

10.2 `analyze`

The program `analyze` is mainly meant to be called from `test`, but it can also be called directly.

Usage:

`analyze` [⟨options⟩] ⟨prog⟩ ⟨mode⟩ ⟨tmp_directory⟩ ⟨prefix_for_output_files⟩ `dirs` ⟨directories⟩

`analyze` [⟨options⟩] ⟨prog⟩ ⟨mode⟩ ⟨tmp_directory⟩ ⟨prefix_for_output_files⟩ `file` ⟨directory⟩ ⟨filename⟩

where ⟨options⟩ can be

- `-timeout` ⟨n⟩ sets the timeout for each execution of the tested program to n seconds (by default, there is no timeout);

- `-progopt` ⟨command-line options⟩ `-endprogopt` passes the additional ⟨command-line options⟩ to the tested program (ProVerif or CryptoVerif);

⟨prog⟩ is either `CV` for CryptoVerif or `PV` for ProVerif and ⟨mode⟩ is as for the `test` program above. Temporary files are stored in directory ⟨tmp_directory⟩, and the output files are:

- full output of the test: `tests/`⟨prefix_for_output_files⟩⟨date⟩,
- summary of the results: `tests/sum-`⟨prefix_for_output_files⟩⟨date⟩,
- comparison with expected results: `tests/res-`⟨prefix_for_output_files⟩⟨date⟩.

This program analyzes a series of scripts using the program specified by ⟨prog⟩.

- In the first command line, it analyzes scripts in the mentioned directories and in their subdirectories. The files whose name contains `.m4.` or `.out.` are excluded. (The first ones are supposed to be files to preprocess by `m4` before actually analyzing them; the second ones are supposed to be output files.) When the program is CryptoVerif, the files whose name ends with `.cv`, `.ocv`, or `.pcv` are analyzed. When the program is ProVerif, the files whose name ends with `.pcv`, `.pv`, `.pi`, `.horn`, or `.horntype` are analyzed.
- In the second command line, the specified file in the specified directory is analyzed, provided it has one of the extensions above. (The directory and the file are mentioned separately because the directory may be used to locate the library `mylib.*`, see below.)

The executable for CryptoVerif is searched in the current directory, in `$HOME/CryptoVerif`, and in the `PATH`. The executable for ProVerif is searched in the current directory, in `$HOME/proverif/proverif`, and in the `PATH`.

When `mylib.cvl` is present in a directory, its files with extension `.cv` or `.pcv` are analyzed using that library of primitives for CryptoVerif. Otherwise, the default library is used.

When `mylib.ocvl` is present in a directory, its files with extension `.ocv` are analyzed using that library of primitives for CryptoVerif. Otherwise, the default library is used.

When `mylib.pvl` is present in a directory, its files with extension `.pcv` or `.pv` are analyzed using that library of primitives for ProVerif. Otherwise, the library `cryptoverif.pvl` is used for `.pcv` files and no library for `.pv` files. The file `cryptoverif.pvl` is searched in the current directory, `$HOME/CryptoVerif` and `$HOME/proverif/proverif`. If it is not found and `mylib.pvl` is not present in the directory, .pcv files are not analyzed using ProVerif.

The result of running each script is compared to the expected result. The expected result is found in the script itself in a comment that starts with `EXPECTED` for CryptoVerif and `EXPECTPV` for ProVerif, and ends with `END`. (The entire lines that contain `EXPECTED`, resp. `EXPECTPV` and `END` do not belong to the expected result.) For CryptoVerif, the expected result consists of the line `RESULT Could not prove ...` or `All queries proved` in the output of CryptoVerif. For ProVerif, it consists of the lines that start with `RESULT` in the output of ProVerif. It also includes a runtime of the script or an error message `xtime: ...` if the execution terminates with an error.

In the modes `update` (resp. `test_add` or `add`), the expected result is updated (resp. added if it is absent or empty). To deal with generated files, the `EXPECTED`, resp. `EXPECTPV` line may contain the indications

`FILENAME:` *name of the file* `TAG:` *distinct tag*

In this case, the expected result is not updated in the script itself, but in the file whose name is mentioned after `FILENAME:`, and inside this file after an exact copy of the line that contains `EXPECTED`, resp. `EXPECTPV`. (This line is unique thanks to the tag.) The idea is that this file is the file from which the script was generated. Hence regenerating the script from this file with an updated expected result will update the expected result in the script.

10.3 addexpectedtags

Usage:

`addexpectedtags` ⟨directories⟩

For each mentioned directory, for each file in that directory or its subdirectories that contains `.m4.` in its name and ends with `.cv`, `.ocv`, `.pcv`, `.pv`, `.pi`, `.horntype`, `.horn`, this program adds at the end of each line that contains `EXPECTED` or `EXPECTPV` the indications

`FILENAME:` *name of the file* `TAG:` *distinct integer*

These files are supposed to be initial models used to generate CryptoVerif or ProVerif scripts by the `m4` preprocessor. The additional indications will propagate to the generated scripts, and will allow the `analyze` program above to find from which `m4` file the script was generated (indicated after `FILENAME:`) and inside this `m4` file, which expected result indication ended up in the considered script (identified by the integer after `TAG:`). It can then update the expected results in the mode `update`, `add`, or `test_add` (the last two when the expected result was initially empty).

Acknowledgments

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

We warmly thank David Pointcheval for his advice and explanations of the computational proofs of protocols. This project would not have been possible without him.

This project was partly supported by ARA SSIA Formacrypt and the ANR projects ProSe (decision ANR-2010-VERS-004-01) and TECAP (decision ANR-17-CE39-0004-03).

References

[1] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Computer and Communications Security (CCS'93)*, pages 62–73, New York, NY, 1993. ACM Press.

[2] 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`.

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

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

[5] 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`.

[6] J. Brendel, M. Fischlin, F. Günther, and C. Janson. PRF-ODH: Relations, instantiations, and impossibility results. Cryptology ePrint Archive, Report 2017/517, 2017. `https://eprint.iacr.org/2017/517`.

[7] S. Goldwasser and M. Bellare. Lecture notes on cryptography. Available at `http://cseweb.ucsd.edu/~mihir/papers/gb.pdf`, July 2008.

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

[9] B. Lipp, B. Blanchet, and K. Bhargavan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. Research report RR-9269, Inria, Apr. 2019. Available at `https://hal.inria.fr/hal-02100345`.

[10] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications and separations for preimage resistance, second-preimage resistance, and collision resistance. Cryptology ePrint Archive, Report 2004/035, 2004. `https://eprint.iacr.org/2004/035`.