1 Introduction

This manual describes the input syntax and output of CryptoVerif. It does not describe the internal algorithms used in the system. These algorithms have been described in the research report [5] and in research papers [3, 2, 6, 7] that can be downloaded at https://bblanche.gitlabpages.inria.fr/publications/index.html.

The goal of CryptoVerif 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 oracles or channels. The oracles frontend uses a calculus closer to cryptographic games, described in Section 3 and in [3, 5, 7]. The channels frontend uses a calculus inspired by the pi calculus, described in Section 4 in [7, Section 2.6] and in [3, 5, 4]. 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 oracles front-end, it is (filename).ocvl; in the channels front-end, the loaded file is (filename).cvl. (The extension .ocvl or .cvl 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.ocvl in the oracles front-end, default.cvl in the channels 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** (language): Instead of proving the protocol, generate an implementation in the chosen language corresponding to the modules defined in the input file. The language can be OCaml or FStar.
• \([M]\) means that \(M\) is optional; \((M)^*\) means that \(M\) occurs 0 or any number of times.

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

• seq\(^+\)(\(X)\) is a non-empty sequence of \(X\): seq\(^+\)(\(X)\) = \([(X)]\)(\(X)\) = \(X\), ..., \(X\). (It can be one or several elements of \(X\) separated by commas.)

Figure 1: Grammar notations

• - 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.

• -EasyCryptConvert (filename) : (equiv name): Converts the equiv statement named (equiv name) to EasyCrypt, so that it can be proved in EasyCrypt. The result is output in (filename). When the filename is omitted (-EasyCryptConvert (equiv name)), the result is output to the standard output.

• -EasyCryptRemoveTables: This option tells CryptoVerif to convert insert/get into find before translating to EasyCrypt (see previous option). By default, CryptoVerif tables are translated to lists in EasyCrypt.

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 \(\equiv\) with CryptoVerif defined. Similarly, when ProVerif analyzes such a file, it first preprocesses it with \(\equiv\) with ProVerif defined. That allows you to conditionally include parts of the file depending on whether CryptoVerif or ProVerif analyzes it.

3 oracles 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, collision, const, def, defined, diff, do, else, eps_find, eps_rand, equation, equiv, equivalence, event, event_abort, exists, expand, find, forall, foreach, fun, get, if, implementation, in, independent-of, inf, inj-event, insert, is-cst, length, let, letfun, letproba, max, maxlen, min, new, newOracle, number, optim-if, orfind, paras, Pcoll1rand, Pcoll2rand, proba, process, proof, public_vars, query, query_equiv, return, run, secret, set, special, suchthat, table, then, time, type, yield.

Strings (string) start and end with \". Inside the string, \" 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 declaration\rangle^* process (o def)

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 declaration\rangle^* equivalence (o def) (o def) [public_vars seq(identify)]
Figure 2: Grammar for terms and patterns
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 \(x_1, \ldots, x_n\). When events are present in \(Q_1\) and/or \(Q_2\), they are considered as visible by the adversary at the end of the game, so the executed events must also be indistinguishable.

Finally, the input may also be:

\[
(\text{declaration})^* \text{ query\_equiv}([\text{public\_vars seq}\{\text{ident}\}] [\text{Seq}^+\{\text{ident}\}] [\text{forall seq}^+\{\text{vartype}\};\langle\text{queryterm}\rangle [\text{public\_vars seq}\{\text{ident}\}] [\text{forall seq}^+\{\text{vartype}\};]\langle\text{queryterm}\rangle [\text{public\_vars seq}\{\text{ident}\}]
\]

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 \([\text{computational}]\) is absent, CryptoVerif then converts this assumption into an equivalence between two processes and tries to prove it.
- When the option \([\text{computational}]\) 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 \([\text{unchanged}]\) 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.

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 \(-\text{lib}\) 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.
(proba ::= (proba))
| (proba) + (proba)
| (proba) - (proba)
| (proba) * (proba)
| (proba) / (proba)
| (proba)^{int}
| max (seq^+(proba))
| min (seq^+(proba))
| (ident)[seq(proba)]
| (ident)
| maxlength (simpleterm)
| length (ident], seq^{+}(proba])
| length (seq(ident)]) [seq^+(proba)])
| n
| #(ident) foreach seq^{+}(ident)
| eps_find
| eps_rand(T)
| Pcoll1rand(T)
| Pcoll2rand(T)

(optimcond ::= (optimcond))
| is-cst (proba)
| (proba) = (proba)
| (proba) <= (proba)
| (proba) >= (proba)
| (proba) < (proba)
| (proba) > (proba)
| (optimcond) && (optimcond)
| (optimcond) || (optimcond)

Figure 4: Grammar for probabilities
\(\text{(repl)} := !\text{[ident]} \leq \text{[ident]}\)
| \(\text{foreach \text{[ident]} \leq \text{[ident]} do}\)
\(\text{(res)} := \text{[ident]} \leftarrow \text{R \text{[ident]}};\)
| \(\text{new \text{[ident]}: \text{[ident]}};\)
\(\text{(obody-equiv)} := ((\text{obody-equiv})\)
| \(\text{event_abort \text{[ident]}}\)
| \(\text{[res] (obody-equiv)}\)
| \(\text{[basicpat]} \leftarrow \text{[term]}; \text{(obody-equiv)}\)
| \(\text{let \text{[pattern]} = \text{[term]} in (obody-equiv) [else (obody-equiv)]}\)
| \(\text{if \text{[cond]} then \text{(obody-equiv) else (obody-equiv)}}\)
| \(\text{find[unique] (findbranch) (orfind (findbranch))* else (obody-equiv)}\)
| \(\text{insert \text{[ident]}(seq[term]); \text{(obody-equiv)}\)
| \(\text{get[unique] \text{[ident]}(seq[pattern]) [suchthat \text{[term]} in (obody-equiv) else (obody-equiv)}\)
| \(\text{return((term))}\)
\(\text{(findbranch)} := \text{seq[identbound] suchthat \text{[cond]} then (obody-equiv)}\)
\(\text{(ogroup)} := \text{[ident]}(\text{seq[vartypeb]}) [\text{Dl}] \text{[useful_change]} := \text{(obody-equiv)}\)
| \(\text{[repl]} \text{[res]}* \text{(ogroup)}\)
| \(\text{[repl]} \text{[res]}* \text{(ogroup)} [\text{...]} \text{(ogroup)}\)
\(\text{(omode)} := \text{(ogroup)} [\text{exist}]\)
| \(\text{(ogroup)} [\text{all}]\)
\(\text{(specialarg)} := \text{[ident]}\)
| \(\text{[string]}\)
| \(\text{[seq(specialarg)]}\)

Figure 5: Grammar for equivalences

\(\text{[dim]} := \text{time[^{\text{[int]}}}\)
| \(\text{length[^{\text{[int]}}}\)
| \(\text{number}\)
| \(\text{[dim]} \ast \text{[dim]}\)
| \(\text{[dim]} \div \text{[dim]}\)
\(\text{[vardim]} := \text{seq[^{\text{[ident]}}} \text{[dim]}\)

Figure 6: Grammar for dimensions
\langle \text{body} \rangle ::= \text{run} \ \langle \text{id} \rangle (\text{seq} \langle \text{term} \rangle) \\
\quad | (\langle \text{body} \rangle) \\
\quad | \text{yield} \\
\quad | \text{event} \ \langle \text{id} \rangle (\text{seq} \langle \text{term} \rangle) \ [; \ \langle \text{body} \rangle] \\
\quad | \text{event\_abort} \ \langle \text{id} \rangle \\
\quad | \langle \text{id} \rangle \leftarrow \text{R} \ \langle \text{id} \rangle; \ \langle \text{body} \rangle \\
\quad | \text{new} \ \langle \text{id} \rangle : \langle \text{id} \rangle; \ \langle \text{body} \rangle \\
\quad | \langle \text{basicpat} \rangle \leftarrow \langle \text{term} \rangle; \ \langle \text{body} \rangle \\
\quad | \text{let} \ \langle \text{pattern} \rangle = \langle \text{term} \rangle \ [\text{in} \ \langle \text{body} \rangle \ [\text{else} \ \langle \text{body} \rangle]] \\
\quad | \text{if} \ \langle \text{cond} \rangle \ \text{then} \ \langle \text{body} \rangle \ [\text{else} \ \langle \text{body} \rangle] \\
\quad | \text{find} [\text{unique}] \ \langle \text{findbranch} \rangle \ (\text{orf} \ \text{findbranch})^* \ [\text{else} \ \langle \text{body} \rangle] \\
\quad | \text{insert} \ \langle \text{id} \rangle (\text{seq} \langle \text{term} \rangle); \ \langle \text{body} \rangle \\
\quad | \text{get} [\text{unique}] \ \langle \text{id} \rangle (\text{seq} \langle \text{pattern} \rangle) \ [\text{suchthat} \ \langle \text{term} \rangle \text{ in} \ \langle \text{body} \rangle \ [\text{else} \ \langle \text{body} \rangle]] \\
\quad | \text{return} (\text{seq} \langle \text{term} \rangle); \ \langle \text{odef} \rangle \\
\langle \text{findbranch} \rangle ::= \text{seq} (\text{id} \text{bound}) \ [\text{suchthat} \ \langle \text{cond} \rangle \ \text{then} \ \langle \text{body} \rangle] \\
\langle \text{odef} \rangle ::= \text{run} \ \langle \text{id} \rangle (\text{seq} \langle \text{term} \rangle) \\
\quad | (\langle \text{odef} \rangle) \\
\quad | 0 \\
\quad | (\langle \text{odef} \rangle) | (\langle \text{odef} \rangle) \\
\quad | \langle \text{id} \rangle \leftarrow \langle \text{id} \rangle \ \langle \text{odef} \rangle \\
\quad | \langle \text{ident} \rangle \leftarrow \langle \text{ident} \rangle \ \text{do} \ \langle \text{odef} \rangle \\
\quad | \langle \text{id} \rangle (\text{seq} \langle \text{pattern} \rangle) ::= \langle \text{body} \rangle \\

\text{Figure 7: Grammar for processes (oracles front-end)}
• An identifier can be either a constant symbol \( f \) (declared by \texttt{const} or \texttt{fun} without argument) or a variable identifier.

• The function application \( f(M_1, \ldots, M_n) \) applies function \( f \) to the result of \( M_1, \ldots, M_n \).

• The tuple application \( (M_1, \ldots, M_n) \) builds a tuple from \( M_1, \ldots, M_n \) (corresponds to the concatenation of \( M_1, \ldots, M_n \) with length and type indications so that \( M_1, \ldots, M_n \) can be recovered without ambiguity). This is allowed only for \( n \neq 1 \), so that it is distinguished from parenthesizing.

• The array access \( x[M_1, \ldots, M_n] \) returns the cell of indices \( M_1, \ldots, M_n \) of array \( x \).

• \( =, \neq, \|, \& \) 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 \(<-R, <-, \text{event}, \text{event_abort}, \text{if}, \text{find}, \text{let}, \text{new}, \text{insert}, \text{get}, \text{and} \) diff which are similar to the corresponding constructs of oracles bodies but return a bitstring instead of executing a process. They are not allowed to occur in defined conditions of \text{find}. The constructs \text{event} and \text{insert} are not allowed to occur in conditions of \text{find} or \text{get}. We refer the reader to the description of processes below for a fully detailed explanation.

• \( x \leftarrow-R \; T; M \) chooses a new random number in type \( T \), stores it in \( x \), and returns the result of \( M \).

• \texttt{new} \( x:T; M \) is equivalent to \( x \leftarrow-R \; T; M \).

• \texttt{let} \( p = M \) in \( M' \) else \( M'' \) tries to decompose the term \( M \) according to pattern \( p \). In case of success, returns the result of \( M' \), otherwise the result of \( M'' \).

The pattern \( p \) can be:

- \( x: [T] \) variable, possibly with its type. Matches any bitstring (in type \( T \)), and stores it in \( x \).

- \( \_ : [T] \) underscore, possibly with a type. Matches any bitstring (in type \( T \)), and ignores its value.

- \( f(p_1, \ldots, p_n) \) where the function symbol \( f \) is declared \texttt{[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 \texttt{[data]} functions encoding unambiguously the values of \( p_1, \ldots, p_n \) and their type.

- \( = \) \texttt{M'} matches a bitstring equal to \( M' \).

When \( p \) is a variable, the \texttt{else} branch can be omitted (it cannot be executed).

• \( x : [T] \leftarrow M ; M' \) stores the result of \( M \) in \( x \) and returns the result of \( M' \). This is equivalent to the construct \texttt{let} \( x : [T] = M \) in \( M' \). \_ is allowed instead of \( x \), and in this case the value of \( M \) is simply ignored.

• \texttt{if} \texttt{cond then} \( M \) \texttt{else} \( M' \) is syntactic sugar for \texttt{find} \texttt{suchthat} \texttt{cond} \texttt{then} \( M \) \texttt{else} \( M' \). It returns the result of \( M \) if the condition \texttt{cond} evaluates to \texttt{true} and of \( M' \) if \texttt{cond} evaluates to \texttt{false}.

• \texttt{find} \texttt{FB}_1 \texttt{orf} \texttt{find} \ldots \texttt{orf} \texttt{find} \ldots \texttt{orf} \texttt{find} \ldots \texttt{else} \( M \) where \( \texttt{FB}_j = u_{i_{j1}} = n_{j1}, \ldots, u_{i_{jm}} = i_{jm} \leftarrow n_{jm}, \texttt{suchthat} \texttt{cond}_j \texttt{then} M_j \texttt{evaluates} \texttt{the} \texttt{conditions} \texttt{cond}_j \texttt{for} \texttt{each} \texttt{j} \texttt{and} \texttt{each} \texttt{value} \texttt{of} \texttt{i}_{j1}, \ldots, \texttt{i}_{jm} \texttt{in} [1, n_{j1}] \times \ldots \times [1, n_{jm}]. \) If none of these conditions is \texttt{true}, it returns the result of \( M \). Otherwise, it chooses randomly with (almost) uniform probability one \( j \) and one value of \( i_{j1}, \ldots, i_{jm} \) such that the corresponding condition is \texttt{true}, stores it in \( u_{i_{j1}}, \ldots, u_{i_{jm}} \), and returns the result of \( M_j \). See the explanation of the \texttt{find} process below for more details.

• \texttt{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 \texttt{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.

• **diff [M_1, M_2]** allows the user to define two processes: in the first process, the first argument of diff is always used, while in the second process, the second argument of diff is always used. CryptoVerif shows indistinguishability between the obtained two processes.

  The calculus distinguishes two kinds of processes: oracle definitions (odef) define new oracles; oracle bodies (obody) return a result after executing some internal computations. Processes allow parenthesizing for disambiguation.

  Let us first describe oracle definitions:

• **run 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 oracle definition. The terms M_1, \ldots, M_n must contain only variables, replication indices, and function applications.

  0 does nothing.

• **Q \sqcup Q'** is the parallel composition of Q and Q'.

• **foreach i \Leftarrow N** do Q represents N copies of Q in parallel each with a different value of i ∈ [1, N]. The identifier N must have been declared by param N. The identifier i is most often used implicitly as array index of variables defined under the replication foreach i \Leftarrow N. It can also be used as argument of events, tables, and tuples (but not as argument of other functions).

  When a program point is under replications foreach i \Leftarrow N_1, \ldots, foreach i_n \Leftarrow N_n, the current replication indices at that point are i_1, \ldots, i_n.

  !i \Leftarrow N Q is equivalent to foreach i \Leftarrow N do Q. The replication !i \Leftarrow N Q can be abbreviated !N Q.

• **O(p_1, \ldots, p_n) := P** defines an oracle O taking arguments p_1, \ldots, p_n, and returning the result of the oracle body P. The patterns p_1, \ldots, p_n are as in the let construct above, except that variables in p that are not under a function symbol f(...) must be declared with their type. When an oracle O is defined under foreach i_1 \Leftarrow N_1, \ldots, foreach i_n \Leftarrow N_n, it implicitly defines O[i_1, \ldots, i_n].

Note that the construct newOracle e; 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, \ldots, M_n)** is replaced with let x_1 = M_1 in \ldots let x_n = M_n in P when proc is declared by let proc(x_1 : T_1, \ldots, x_n : T_n) = P, where P is an oracle body.
• **yield** terminates the oracle, returning control to the caller. Intuitively, it represents termination with an error.

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

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

• \(x \leftarrow R T\) or \(\text{new } x; T\) chooses a new random number in type \(T\), stores it in \(x\), and executes \(P\). \(T\) must be declared with option \textit{fixed}, \textit{bounded}, or \textit{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 \texttt{time(<-R \(T\))} or equivalently \texttt{time(new \(\text{\texttt{T}}\))}.

  - When the type \(T\) is \textit{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 chosen uniformly among a finite set of bitstrings, chosen in another type.)

  - When the type \(T\) is \textit{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 \textit{nonuniform}, the default probability distribution \(D_T\) for \(T\) is the uniform distribution.

  - For other \textit{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 \textit{nonuniform}, the default probability distribution \(D_T\) for \(T\) is such that its distance to the uniform distribution is at most \(\text{eps\_rand}(T)\). The distance between two probability distributions \(D_1\) and \(D_2\) for type \(T\) is

\[
d(D_1, D_2) = \frac{1}{2} \sum_{a \in T} |\Pr[X_1 = a] - \Pr[X_2 = a]|\]

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 \(x' \mod m\). By euclidian division, we have \(2^k = qm + r\) with \(r \in [0, m-1]\). With this algorithm

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

so

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

Therefore

\[
d(D_T, \text{uniform}) = \frac{1}{2} \sum_{a \in T} \left| \Pr[x = a] - \frac{1}{m} \right| = \frac{1}{2} r \left( \frac{q+1}{2^k} - \frac{1}{m} \right) - \frac{1}{2} (m-r) \left( \frac{1}{m} - \frac{q}{2^k} \right)
\]

\[
= \frac{r(m-r)}{m \cdot 2^k} \leq \frac{m}{2^k+1}
\]

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

When \texttt{IgnoreSmallTimes} is set to a value greater than 0 (which is the default), the time for random number generations and the probability \(\text{eps\_rand}(T)\) 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 \( x \).
- \( _{\_} : [T] \) underscore, possibly with a type. Matches any bitstring (in type \( T \)), and ignores its value.
- \( f(p_1, \ldots, p_n) \) 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 branch is never executed when the pattern is simply a variable or underscore. When else \( P' \) is omitted, it is equivalent to \( \text{else yield} \). Similarly, when in \( P \) is omitted, it is equivalent to in yield.

- \( x: [T] \leftarrow M \); \( P \) 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 \( P \). _\_ is allowed instead of \( x \), and in this case the value of \( M \) is simply ignored.

- if \( \text{cond} \) then \( P \) else \( P' \) is syntactic sugar for find such that \( \text{cond} \) then \( P \) else \( P' \). It executes \( P \) if the condition \( \text{cond} \) evaluates to true and \( P' \) if \( \text{cond} \) evaluates to false. When the else branch is omitted, it is implicitly \( \text{else yield} \). (else 0 would not be syntactically correct.)

Next, we explain the process find \( FB_1 \) orfind ... orfind \( FB_m \) else \( P \) where each branch \( FB_i \) is \( FB_j = u_j = i_j = n_j, \ldots, u_{jm} = i_{jm} = n_{jm} \) such that \( \text{cond}_j \) then \( P_j \).

A simple example is the following: find \( u = i \Leftarrow n \) such that \( \text{defined}(x[i]) \land x[i] = a \) 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 \( u_i = i_1 \Leftarrow n_1, \ldots, u_m = i_m \Leftarrow n_m \) such that \( \text{defined}(M_1, \ldots, M_j) \land M \) then \( P' \) else \( P \) tries to find values of \( i_1, \ldots, i_m \) for which \( M_1, \ldots, M_j \) are defined and \( M \) is true. In case of success, it stores the values of \( i_1, \ldots, i_m \) in \( u_1, \ldots, u_m \) 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 \( FB_j = u_j = i_j = n_j, \ldots, u_{jm} = i_{jm} = n_{jm} \) such that \( \text{defined}(M_1, \ldots, M_j) \land M \) then \( P_j \) tries to find a branch \( j \) in \( [1, m] \) such that there are values of \( i_{j1}, \ldots, i_{jm} \) for which \( M_{j1}, \ldots, M_{jm} \) are defined and \( M \) is true. In case of success, it stores the value of \( i_{j1}, \ldots, i_{jm} \) in \( u_{j1}, \ldots, u_{jm} \) and executes \( P_j \). In case of failure, it executes \( P \). More formally, it evaluates the conditions \( \text{cond}_j = \text{defined}(M_1, \ldots, M_j) \land M \) for each \( j \) and each value of \( i_{j1}, \ldots, i_{jm} \) in \( [1, n_{j1}] \times \cdots \times [1, n_{jm}] \). 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 \( i_{j1}, \ldots, i_{jm} \) such that the corresponding condition is true, stores that value in \( u_{j1}, \ldots, u_{jm} \), and executes \( P_j \).

In the general case, the conditions \( \text{cond}_j \) are of the form \( \text{defined}(M_1, \ldots, M_j) \land M \) or simply \( M \). The condition \( \text{defined}(M_1, \ldots, M_j) \) means that \( M_1, \ldots, M_j \) are defined. At least one of the two conditions \( \text{defined} \) or \( M \) must be present. Omitted \( \text{defined} \) conditions are considered empty; when \( M \) is omitted, it is considered true.

\(^1\)Precisely, the distance between the distribution actually used for choosing \( j, i_{j1}, \ldots, i_{jm} \) and the uniform distribution is at most \( \varepsilon \frac{\text{eps}_{\text{find}}}{2} \). See the explanation of \( x \Leftarrow T \) for details on how to achieve this.
foreach abbreviation ! 

\[ \text{foreach } i \]

\[ \text{foreach } M \]

\[ \text{foreach } CryptoV erif accept two synchronies for replication (} \hat{\text{diff}}[\hat{\text{return}}(\hat{\text{get}}] \text{is always used, while in the second process, the second argument of equivalence by } CryptoV erif \text{ shows indistinguishability between the obtained two processes. In this case, the same identifier } u_{jk} \text{ is used for both the variable and the associated replication index } i_{jk}. \]

As an abbreviation, one may write \( FB_j = u_{j1} \Leftarrow n_{j1}, \ldots, u_{jm_j} \Leftarrow n_{jm_j} \text{ such that defined}(M_{j1}, \ldots, M_{jm_j}) \&\& M_j \text{ then } P_j. \) In this case, the same identifier \( u_{jk} \) is used for both the variable and the associated replication index \( i_{jk}. \)

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

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

- \text{get} \( \text{tbl}(p_1, \ldots, p_n) \) such that \( M \in P \) else \( P' \) tries to find an element of the table \( \text{tbl} \) that matches the patterns \( p_1, \ldots, p_n \) and such that \( M \) is true. If it succeeds, it executes \( P \) 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 executes \( P'. \) When else \( P' \) is omitted, it is equivalent to else yield. When such that \( M \) is omitted, it is equivalent to such that \( \text{true}. \)

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

Internally, \text{get} is converted into \text{find} by CryptoV erif.

- \text{return} \( (N_1, \ldots, N_n); Q \) terminates the oracle, returning the result of the terms \( N_1, \ldots, N_n. \) Then, it makes available the oracles defined in \( Q \) and gives back control to the attacker.

- \text{diff}[P_1, P_2] \text{ allows the user to define two processes: in the first process, the first argument of } \text{diff} \text{ is always used, while in the second process, the second argument of } \text{diff} \text{ is always used. CryptoV erif shows indistinguishability between the obtained two processes. In this proof, events are considered as invisible for the adversary. (That differs from proofs of indistinguishability made by equivalence.) Internally, this is encoded by choosing a random bit } \text{diff_bit}, \text{ running the first argument of } \text{diff} \text{ when } \text{diff_bit} \text{ is true and the second one when } \text{diff_bit} \text{ is false, and showing the secrecy of } \text{diff_bit}. \text{ The variables that are public due to other queries (variables included in } \text{public_vars in queries as well as variables on which there is a query secret except query secret ... [reachability,onesession]} \text{ are considered public when proving the secrecy of } \text{diff_bit}. \text{ The random bit } \text{diff_bit} \text{ is considered public in the other queries, so that the adversary knows which side of } \text{diff} \text{ is executed.}

CryptoV erif accept two syntaxes for replication (\text{foreach } i \Leftarrow N \text{ do } Q \text{ and } \text{let } i \Leftarrow N \text{ with the abbreviation } !N \text{ Q}) \text{, generation of random numbers (} x \Leftarrow \text{R } T; P \text{ and new } x ; T; P \text{, and assignments (} x \Leftarrow M; P \text{ and let } x = M \text{ in } P \text{ when the pattern is a variable). For the display, the notations } \text{foreach } i \Leftarrow N \text{ do } Q \text{, } x \Leftarrow \text{R } T; P \text{, and } x \Leftarrow M; P \text{ are always used.}

In this calculus, all variables are implicitly arrays, where values can be stored but not overwritten.

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

In the initial game, several variables may be defined with the same name, but they are immediately renamed to different names, so that after renaming, each variable is defined once. When several variables are defined with the same name, they can be referenced only under their definition without explicit array 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, ..., M_n]` in `M` in a find branch ... such that defined(`M'_1, ..., M'_l`) & & `M` then ..., such that `x[M_1, ..., M_n]` is a subterm of `M'_1, ..., M'_l`.
- `x[M_1, ..., M_n]` in `P` in a find branch `u_1 = i_1 <= n_1, ..., u_m = i_m <= n_m` such that defined(`M'_1, ..., M'_l`) & & ... then `P`, such that `x[M_1, ..., M_n] = M{u_1/i_1, ..., u_m/i_m}` and `M` is a subterm of `M'_1, ..., M'_l`.
- `x[M_1, ..., M_n]` in `M''` in a find branch `u_1 = i_1 <= n_1, ..., u_m = i_m <= n_m` such that defined(`M'_1, ..., M'_l`) & & ... then `M''`, such that `x[M_1, ..., M_n] = M{u_1/i_1, ..., u_m/i_m}` and `M` is a subterm of `M'_1, ..., 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 `foreach i_1 <= N_1, ..., foreach i_n <= N_n`, `x` has type `[1, N_1] \times \ldots \times [1, N_n] \rightarrow T`. When the type of `x` is not explicitly given in its declaration (in `let` or in oracle definitions), its type is left undefined in this pass, and `x` cannot be used outside its scope.

In the second pass, the type system checks the following requirements: In `x[M_1, ..., M_n]`, `M_1, ..., M_n` 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, ..., M_n)`, if `f` has been declared by `fun f(T_1, ..., T_m) : T`, `M_j` must be of type `T_j`, and `f(M_1, ..., M_n)` is then of type `T`. In `(M_1, ..., 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 \neq M_2`, `M_1` and `M_2` must be of the same type, and the result is of type `bool`. In `M_1 \land M_2` and `M_1 \lor M_2`, `M_1` and `M_2` must be of `bool` type and the result is of `bool` type. The type system requires each subterm to be well-typed. Furthermore, in `event c(M_1, ..., M_n)`, if `c` has been declared by `event c(T_1, ..., T_n)`, `M_j` must be of type `T_j`, in `x <=b T` or `new x : T`, `T` must be declared with option `bounded` (or `fixed`).

In `if M then ... else ...`, `M` must be of type `bool`. Similarly, for

```
find ... or find ... such that defined(...) & & M then ...
```

`M` must be of `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
A declaration can be:

- set \( \text{parameter} = \langle \text{value} \rangle \).

This declaration sets the value of configuration parameters. We provide next a list of the parameters and values supported, where 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.

- set allowUndefinedVar = false.
  set allowUndefinedVar = true.

  By default \( \text{allowUndefinedVar = false} \), variables in defined conditions must be defined somewhere in the game. The setting \( \text{allowUndefinedVar = true} \) 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 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 \( \text{let } x = y \text{ or } x \leftarrow y \) where \( x \) and \( y \) are variables by substituting \( y \) for \( x \) (in the transformation \text{remove_assign useless}). When false, this transformation is not performed as part of \text{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 \text{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 guessRemoveUnique = false.
  set guessRemoveUnique = true.
When we use a guess transformation (guess or guess_branch) and a (one-session) secrecy
query is present, we must reprove that all find[unique] in the game are really unique. When
guessRemoveUnique = false (the default), we reprove them. When guessRemoveUnique =
true, we instead remove the [unique] annotations to avoid having to reprove them.

- set autoSARename = true.
  set autoSARename = false.
When true, and a variable is defined several times and used only in the scope of its defi-
nition 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 transforma-
tion all_simplify and by the simplification after the crypto and SARename transformations.
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 assign-
ments on variables x defined by let x = M in ... inside a condition of find. When false,
this removal 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 (<-R or new) down-
  wards 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, <-R, 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 correspond-
 ence queries when the arguments of the considered events contain if, let, <-, find, event,
  event_abort, <-R, 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 automati-
  cally. When autoAdvice = false, display the advised transformations, but do not execute
  them. Users may then give them as instructions if they wish.

- 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 gen-
  erating advice. Useful when the global dependency analysis generates bad advice.

- set simplifyAfterSARename = true.
  set simplifyAfterSARename = false.

  When simplifyAfterSARename = true, apply simplification (that is, remove_assign
  useless if autoRemoveAssignFindCond = false, remove_assign findcond if
  autoRemoveAssignFindCond = true, SArename new if autoSARename = true, simplify)
  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 an 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 setting 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 $x[M_1, \ldots, M_l]$, 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:

$$\forall u_1, \ldots, u_k, \neg (\text{defined}(y_1[M_{11}, \ldots, M_{1l}], \ldots, y_k[M_{k1}, \ldots, M_{kl}]) \land t)$$

We try to infer $\neg(t)$ from this fact.

* if each variable $y_j[M_{j1}, \ldots, M_{jl}]$ is defined before $x[M_1, \ldots, M_l]$, then $\neg(t)$ indeed holds by the fact above;
* for each $y_j[M_{j1}, \ldots, M_{jl}]$, we assume that $y_j[M_{j1}, \ldots, M_{jl}]$ is defined after or at the same time as $x[M_1, \ldots, M_l]$ and try to prove $\neg(t)$.

If this proof succeeds, we can infer that $\neg(t)$ 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 elseifAdditionalDisjunct = true.
  set elseifAdditionalDisjunct = false.
When elseifAdditionalDisjunct = true, the procedure that infers facts from false conditions of find (see set elsefindFactsInReplace) is enriched: in case \( y_j[M_1, \ldots, M_{jl}] \) may be defined at the same time as \( x[M_1, \ldots, M_l] \), we additionally assume that they have different indices, that is, \( (M_1, \ldots, M_{jl}) \neq (M_1, \ldots, M_l) \) to eliminate this case. Therefore, we infer \( (M_1, \ldots, M_{jl}) \neq (M_1, \ldots, M_l) \Rightarrow \neg t(1) \) or equivalently \( (M_1, \ldots, M_{jl}) = (M_1, \ldots, M_l) \vee \neg t(1) \). This is typically more costly and more precise than the basic procedure that just infers \( \neg t(1) \) 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 elseif 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 = n. (default 3)
When 0, the evaluation of the runtime is very precise, but the formulas are often too complicated to read.

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

When 2, the system ignores even more details, including application of boolean operations \( \& \), \( \| \), \( \neg \), constants generated by the system, \( () \) and matching on \( () \). It ignores the creation and decomposition of tuples in oracle calls and returns.

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

- set maxIterSimplf = 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 \( n = 0 \), 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 \( n = 0 \), 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 \( n = 0 \), puts no limit on this depth of recursion. Putting a limit avoids an infinite loop in some rare cases.
bounding the number of calls to random oracles.

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].

- 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 maxIsIndepDepth = n. (default 1)
  Sets the maximum depth of recursion of the replacement of already rewritten variables with their values in proofs that a term is independent of some value. When n = 0, puts no limit on this depth of recursion. Putting a limit avoids an infinite loop in some rare cases.

- set maxIterRemoveUselessAssign = n. (default 10)
  Sets the maximum number of repetitions of the removal of useless assignments for each remove_assign useless instruction. A greater value slows down the system but may enable it to obtain simpler games, and therefore increase its chances of success. When n = 0, repeats removal of useless assignments until a fixpoint is reached.

- set maxAdvicePossibilitiesBeginning = n1. (default 50)
  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 n1 first possibilities (with highest priority) and the n2 last possibilities (with lowest priority but fewer advised transformations). When n1 or n2 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^{10}), or sizen (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.

• param seq^+(ident) [[noninteractive] | [passive] | [default] | [small] | [sizen]].
  param n_1, . . . , n_m. declares parameters n_1, . . . , 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 [sizen] indicate to CryptoVerif an order of magnitude of the parameter. The option [sizen] (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.
The absence of option, the option \([\text{default}]\), and the option \([\text{passive}]\) 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 \([\text{passive}]\)). Therefore, for such runs, the adversary is undetected.

No option, \([\text{default}]\), and \([\text{passive}]\) are equivalent to \([\text{size30}]\).
The option \([\text{small}]\) 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). \([\text{small}]\) is equivalent to \([\text{size2}]\).

- **proba** \([\text{ident}][\text{(seq}(\text{dim}))[\text{([pest]})]]\).
  proba\(p(d_1, \ldots, d_n)\), declares a probability function \(p\) taking \(n\) arguments of dimensions \(d_1, \ldots, d_n\) respectively. The syntax of dimensions is given in Figure 6 where \(*\), \(/\), and \(^\wedge\) are the usual product, division, and exponentiation. After reduction, dimensions are of the form \(\text{time}^l \times \text{length}^t\), where \(t\) and \(l\) are integers. The dimension number corresponds to \(\text{time}^l \times \text{length}^t\).
  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 \([\text{pest}]\) (probability estimate) is present, it gives an estimate of the value of the probability: \(\text{pest} n\), where \(n\) is an integer, means that the probability is at most \(2^{-n}\); \(\text{password}\) is equivalent to \(\text{pest} 20\), i.e. probability at most \(2^{-20}\); \(\text{large}\) is equivalent to \(\text{pest} 160\), i.e. probability at most \(2^{-160}\).
  When \([\text{pest}]\) is absent, \(\text{large}\) is the default. When the probability \(p\) appears in a collision statement and the command \text{allowed collisions} \(\text{pest} n\) 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.

- **letproa** \([\text{ident}][\text{(seq}(+\text{(var dim))})] = (\text{proba})\).
  \(\text{letproa}\(p(x_1 : d_1, \ldots, x_n : d_n) = \text{prob}\), declares a probability function \(p\) with \(n\) arguments \(x_i\) of dimension \(d_i\), equal to the probability formula \(\text{prob}\). See proba above for an explanation of dimensions. The formula \(\text{prob}\) must represent a probability. It may refer to \(x_1, \ldots, x_n\). It is instantiated with the appropriate values of \(x_1, \ldots, x_n\) every time the probability function \(p\) is applied.

- **type** \([\text{ident}][\text{([seq}(+\text{(option)}))}]\).
  type\(T\), declares a type \(T\). Types correspond to sets of bitstrings or a special symbol \(\bot\) (used for failed decryptions, for instance). Optionally, the declaration of a type may be followed by options between brackets. These options can be:

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

`sizemin_max` means that \(2^{\min} \leq |T| \leq 2^{\max}\), where `min` and `max` are integers.

- `pcoll` (probability of collision) means that `Pcoll1rand(T) \leq 2^{-n}`. By default, `Pcoll1rand(T)` is the probability of collision between a random element chosen according to the default probability distribution \(DT\) 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)` is equivalent to \(\frac{1}{n}\), so CryptoVerif estimates the probability of collision from the cardinal of the type and conversely, so mentioning one of `sizex` or `pcoll` 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` with \(n \geq n'\), where `n'` is set by `set minAutoCollElim = pestn` (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` with \(n \geq 160\) or option `large`) 
   - or `Pcoll1rand(T) \leq 2^{-20} (T` has option `pcoll` 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 `sizex160_10000000000, 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 `sizex20_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 about one million \((2^{20})\) and about one trillion \((2^{40})\) elements.

- `small` is equivalent to `sizex0_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(T1, . . . , Tn) : T`, declares a function that takes `n` arguments, of types `T1, . . . , Tn`, 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: \(f(x_1, . . . , x_n) = y\) implies for \(i \in \{1, . . . , m\}\), \(x_i = f^{-1}_i(y)\) for some functions \(f^{-1}_i\). (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.
- [autoSwapIf] tells CryptoVerif to rewrite terms of the form $f(..., if\_fun(M_1, M_2, M_3), ...) \rightarrow
 if\_fun(M_1, f(..., M_2, ...), f(..., M_3, ...))$, where $if\_fun$ is a predefined test function that satisfies $if\_fun(true, x, y) = x$ and $if\_fun(false, x, y) = y$.

- **letfun** (letfun term) \rightarrow (term).
  letfun $f(x_1:T_1, ..., x_n:T_n) = M$, declares a function $f$ that takes $n$ arguments named $x_1, ..., x_n$ of types $T_1, ..., T_n$, respectively. The subsequent calls to this function are replaced by the term $M$ in which we replace $x_1, ..., x_n$ with the arguments given by the caller. (We use $x_i \leftarrow N_i$ 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) + (ident)).

- **const** $c_1, ..., c_n : T$. declares constants $c_1, ..., c_n$ 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, ..., T_n)` declares the table `tbl`, whose elements are tuples of type $T_1, ..., T_n$. 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, ..., M_n)` and the table can be read using `get`.

- **event** (seq (ident)).
  `event e(T_1, ..., T_n)`, declares an event $e$ that takes arguments of types $T_1, ..., T_n$. 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) [(letterm)] = (body).
- **let** (ident) [(letterm)] = (def).
  `let proc(x_1 : T_1, ..., 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 $x_i \leftarrow N_i$ 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, run `proc(M_1, ..., M_n)` will be replaced with $P(M_1/x_1, ..., M_n/x_n)$ when $P$ is an oracle definition. In this case, the terms $M_1, ..., M_n$ must contain only variables, replication indices, and function applications and the variables $x_1, ..., x_n$ cannot have array accesses. The process run `proc(M_1, ..., M_n)` will be replaced with let $x_1 = M_1$ in ... let $x_n = M_n$ in $P$ when $P$ is an oracle body.

- **equation** ([ident]) [(ident)] | [forall seq (ident);] | [letterm] | [if (letterm)] | [manual].
  `equation forall x_1 : T_1, ..., x_n : T_n ; M`, says that for all values of $x_1, ..., x_n$ in types $T_1, ..., T_n$ respectively, $M$ is true. The term $M$ must be a simple term without array accesses. All bound variables $x_1, ..., x_n$ 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 $x_1, ..., x_n$ 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, $M_1 \not= M_2$, CryptoVerif rewrites $M_1 = M_2$ to false and $M_1 \not< M_2$ 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, ..., x_n : T_n ; M if M'`, says that for all values of $x_1, ..., x_n$ in types $T_1, ..., T_n$, 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.

Equations can be named by writing `equation(name) forall x_1 : T_1, ..., x_n : T_n ; M if M'`, where the name `name` can be either an identifier `id`, or `id(f)`, where `id` is an identifier and $f$ a
second identifier. Names of the form $\text{id}(f)$ are most useful when the equivalence is defined inside a macro definition (def). In this case, the identifier $\text{id}$ 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 is useful to designate the equation in the proof command use, which can activate or deactivate equations (see Section 7). All equations are initially active, except when they have the option [manual], in which case they are initially inactive. Active equations are used by other commands (e.g. simplify) in order to reason on the game.

Equations must be named when they have no universally quantified variable and the term $M$ starts with a parenthesis, because otherwise the beginning of $M$ would erroneously be interpreted as the beginning of the name of the equation.

- **equation builtin (eq_name)(eq_name+eq_name).**
  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 $f(x, y) \rightarrow f(y, x)$ does not terminate.)

  - **equation builtin assoc($f$).** indicates that the function $f$ is associative, that is, $f(f(x, y), z) = f(x, f(y, z))$ 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 assoc($f$, $n$).** 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 a group with inverse $\text{inv}$ and neutral element $n$, that is, the function $f$ is associative, $n$ is a neutral element for $f$, and $\text{inv}(x)$ is the inverse of $x$, that is, $f(\text{inv}(x), x) = f(x, \text{inv}(x)) = n$. In this case, the function $f$ must be a binary function with both arguments and the result of the same type $T$, $\text{inv}$ 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 $f$ forms a commutative group with inverse $\text{inv}$ and neutral element $n$, that is, the function $f$ is associative and commutative, $n$ is a neutral element for $f$, and $\text{inv}(x)$ 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 $T$, $\text{inv}$ must be a unary function that takes and returns a value of type $T$, and $n$ must be a constant of type $T$.

- **collision([\text{ident}](\text{[ident]})) | \{res\}*[\text{random_choices_may_be_equal}] | \forall seq(vartype); return(\text{letterm}) \leq (\text{proba}) \Rightarrow \{\text{manual}\} return(\text{letterm}) | \text{if (cond)}.**
where

\[(\text{cond}) ::= (\text{ident}) \mid \text{(letterm)} \mid (\text{ident}) \text{ independent-of } (\text{ident}) \mid (\text{cond}) \& (\text{cond}) \mid (\text{cond}) \text{ || } (\text{cond}) \mid (\text{basicpat}) \leftarrow (\text{letterm}) ; (\text{cond}) \mid \text{let } (\text{basicpat}) = (\text{letterm}) \text{ in } (\text{cond})\]

collision \(x_1 \leftarrow R \ T_1; \ldots x_n \leftarrow R \ T_n \); forall \(y_1 : T'_1, \ldots, y_m : T'_m\);
\[\text{return } (M_1) \lhd (p) \Rightarrow \text{return } (M_2)\]

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

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

collision \(x_1 \leftarrow R \ T_1; \ldots x_n \leftarrow R \ T_n \); forall \(y_1 : T'_1, \ldots, y_m : T'_m\);
\[\text{return } (M_1) \lhd (p) \Rightarrow \text{return } (M_2) \text{ if } c\]

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 \(\leftarrow R\) or new. (However, disjunctions cannot mix terms and independence conditions.)

The option \(\text{random_choices_may_be_equal}\), when it is present, allows several random number generations among \(x_1, \ldots, x_n\) to be the same, instead of being independent. One can then group, in a single collision statement, situations in which \(x_1, \ldots, x_n\) are the same or they are independent. The indices of the variables corresponding to \(x_1, \ldots, x_n\) in the game are still made independent of \(x_1, \ldots, x_n\). 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 \(\text{random_choices_may_be_equal}\), the independence conditions can also be "\(x_i\) independent-of \(x_j\)\", where \(x_i\) and \(x_j\) are both bound by \(\leftarrow R\) or 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.

Collisions can be named by writing

collision \(\text{name} \ x_1 \leftarrow R \ T_1; \ldots x_n \leftarrow R \ T_n \); forall \(y_1 : T'_1, \ldots, y_m : T'_m\);
\[\text{return } (M_1) \lhd (p) \Rightarrow \text{return } (M_2) \text{ if } c\]

where the syntax of names \(\text{name}\) is explained in the first equation declaration above. The name is useful to designate the collision in the proof command use, which can activate or deactivate collisions (see Section 7). All collisions are initially active, except when they have the option \(\text{manual}\), in which case they are initially inactive. Active collisions are used by other commands (e.g. \(\text{simplify}\)) in order to reason on the game.

- \(\text{equiv}((\text{ident})|(\text{ident}))\]
- \((\text{omode}) [1 \ldots 1 (\text{omode})] \lhd (\text{proba}) \Rightarrow [\text{let}] [\text{seq}^+(\text{option})] \langle\text{group}\rangle [1 \ldots 1 (\text{group})]\]
- \(\text{equiv}((\text{name}) \ L \lhd (p) \Rightarrow R\]

means that the probability that a probabilistic Turing machine that runs in time \(t\) and \(d\) respectively, and returns the result computed by \(R\). The oracle body \(R\) is similar to term, but terminates with a return as shown in the grammar of \(\langle\text{body_equiv}\rangle\) (Figure 5).

O(\(x_1 : T_1, \ldots, x_n : T_n\) := FP represents an oracle \(O\) that takes arguments \(x_1, \ldots, x_n\) of types \(T_1, \ldots, T_n\) 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 \(\langle\text{body_equiv}\rangle\) (Figure 5).
foreach $i \leftarrow N$ do $y_1 \leftarrow R T'_1; \ldots; y_m \leftarrow R T'_m; (FG_1 | \ldots | FG_n)$ represents $N$ copies of a process that picks fresh random numbers $y_1, \ldots, y_m$ of types $T'_1, \ldots, T'_m$ respectively, and makes available the functions described in $FG_1, \ldots, FG_n$. Each copy has a different value of $i \in [1, N]$. The identifier $i$ is most often used implicitly as array index of variables defined under foreach $i \leftarrow N$. It can also be used as argument of tables and tuples (but not as argument of other functions). The replication foreach $i \leftarrow N$ do can also be written $!i \leftarrow N$ and can be abbreviated $!N$. (In these definitions, foreach $i \leftarrow N$ do new $x_1:T_1; \ldots; new \ x_m:T_m; \ return; Q$ in fact stands for foreach $i \leftarrow N$ do $O() := new \ x_1:T_1; \ldots; new \ x_m:T_m; \ return; Q$, where $O$ is a fresh oracle name. The same oracle names are used in both sides of the equivalence.)

The replication foreach $i \leftarrow N$ 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.

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

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

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

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

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 be 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 manual option can be cancelled using the use 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 $O(\ldots) := \ return(M)$ where $M$ is a simple term. Oracle definitions in $R$ are of the form $O(\ldots) := \ (\text{body}_\text{equiv})$. 

26
- $L$ 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
  \[ \text{find}[\text{[unique]}] \ldots \]
  \[ \text{orfind } u_1 \Leftarrow N_1, \ldots, u_m \Leftarrow N_m \text{ such that defined}(z_1[\hat{u}_1], \ldots, z_l[\hat{u}_l]) \& M \text{ then } FP \]
  \[ \ldots \text{ else } FP' \]

where $\hat{u}_k$ is a non-empty suffix of $u_1, \ldots, u_m$, at least one $\hat{u}_k$ for $1 \leq k \leq l$ is the whole sequence $u_1, \ldots, u_m$ optionally followed by a sequence of indices $\hat{u}_0$, 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
  \[ \text{foreach } i_1 \Leftarrow N_1, \ldots, \text{foreach } i_n \Leftarrow N_n, \ z \text{ is always an array with } n \text{ dimensions, so it}
  \]
  \[ \text{expects } n \text{ indices}, \text{but the first } n' < n \text{ indices are left implicit when they are equal to the}
  \]
  \[ \text{current indices of the top-most } n' \text{ replications above the usage of } z \text{—which must also be the}
  \]
  \[ \text{top-most } n' \text{ replications above the definition of } z. \text{ We require the implicit indices to be the}
  \]
  \[ \text{same for all variables } z_1, \ldots, z_l.) \text{ Furthermore, there must exist } k \in \{1, \ldots, l_j\} \text{ such that for all}
  \]
  \[ k' \neq k, \ z_{k'} \text{ is defined syntactically above all definitions of } z_k \text{ and } \hat{u}_{k'} \text{ is a suffix of } \hat{u}_k.
  \]

When $\hat{u}_0$ is not empty, the find is automatically transformed into

\[ \text{find}[\text{[unique]}] \ldots \]
\[ \text{orfind } u_1 \Leftarrow N_1, \ldots, u_m \Leftarrow N_m, \hat{u}_0' \Leftarrow \hat{N}_0' \text{ such that defined}(z_1[\hat{u}_1'], \ldots, z_l[\hat{u}_l']) \& \]
\[ \text{(} \hat{u}_0', \hat{i} \text{)} = (\hat{u}_0, \hat{i}) \& M' \text{ then } FP'' \]
\[ \ldots \text{ else } FP' \]

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

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 $(\ldots)$ 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 9 or copy-pasting declarations from examples.
equiv((ident)|(ident)) | special (ident)(seq(specialarg)) [manual] | [v]n,

equiv(name) special specialname(a1,...,an) 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(a1,...,an).

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) = a1,...,an:

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, \( p_{coll}(T) \leq 2^{-n} \), where \( n' \) is set by \( \text{set minAutoCollElim} = \text{pest} n' \); 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 \( p(t,N,l_1,...,l_m) \) 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 \( l_1,...,l_m \). 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 : r \text{ < } T; r' \text{ < } T; \forall a_1 : T_1;...,a_n : T_n; M^n" (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;...,a_n : T_n; r_1 \text{ < } T; M^n" when T is the type of the result of f and the simple term M uses the variables a1,...,an,r1. In this case, CryptoVerif tries to simplify M assuming r1 is a random value and a1,...,an do not depend on r1. If it rewrites M into a term N that does not contain r1, then it uses this information to transform terms M\{f(...)\} into N when the result of f(...) 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: \(r_1 \leftarrow T; \text{forall } a_1: T_1, \ldots, a_n: T_n; M^n\) where \(T\) is the type of the result of \(f\) and the simple term \(M\) that uses the variables \(a_1, \ldots, a_n, r_1\). In this case, CryptoVerif tries to simplify \(M\) assuming \(r_1\) is a random value (\(a_1, \ldots, a_n\) 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 \(M\{f(...)/r_1\}\) into \(N\), in the generated cryptographic transformation.

- "Ocoll: \(r_1 \leftarrow T; r_2 \leftarrow T; \text{forall } a_1: T_1, \ldots, a_n: T_n; M^n\) where \(T\) is the type of the result of \(f\) and the simple term \(M\) that uses the variables \(a_1, \ldots, a_n, r_1, r_2\). 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 \(n = 0\), for all \(a_1: T_1, \ldots, a_n: T_n\); 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 \(0\) 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 \(\text{pcoll1}\) with \(n \geq n'\) where \(n'\) is set by \(\text{set minAutoCollElim = pest'nv};\) the default is \(n' = 80\). In this case, this is equivalent to \(collisions\_LHS\) containing:

- "Eq: for all \(a_1: T_1; r_1 \leftarrow T; r_1 = a_1^n\). Assuming \(a_1\) does not depend on \(r_1\), \(r_1 = a_1\) simplifies into \(false\), so \(f(...) = a_1\) is transformed into \(false\) in the generated cryptographic transformation, when the result of \(f(...)\) is a fresh random value.

- When \(\text{specialname}\) is not \(\text{prp}\) nor \(\text{prp}\_partial\), "Ocoll: \(r_1 \leftarrow T; r_2 \leftarrow T; r_1 = r_2\). The term \(r_1 = r_2\) simplifies into \(false\) when \(r_1\) and \(r_2\) are independent random values and into \(true\) when \(r_1\) and \(r_2\) are equal random values. If it succeeds, then it \(f(args) = f(args)\) is transformed into \(args = args\) in the generated cryptographic transformation.

The argument \(collisions\_LHS\) can be overridden when the equivalence is used in a \(\text{crypto}\) command, by passing the desired \(collisions\_LHS\) as special argument to the \(\text{crypto}\) command.

The last or the last two arguments may be omitted.

When \(\text{specialname}\) is \(\text{rom}\), \(\text{prf}\), or \(\text{prp}\), the generated equivalence provides the following oracles:

- Oracle \(0\) evaluates \(f\) on its arguments in the left-hand side, and performs a lookup into previous arguments of \(0\) 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 \(\text{specialname}\) is \(\text{rom}\_partial\), \(\text{prf}\_partial\), or \(\text{prp}\_partial\), the generated equivalence provides oracles named \(Ocoll\) for each element "Ocoll: \(r_1 \leftarrow T; \text{forall } a_1: T_1, \ldots, a_n: T_n; M^n\) or "Ocoll: \(r_1 \leftarrow T; r_2 \leftarrow T; \text{forall } a_1: T_1, \ldots, a_n: T_n; M^n\" of \(collisions\_LHS\). These oracles act like the oracle of the same name when \(\text{specialname}\) is \(\text{rom}\) (resp. \(\text{prf}\)—there are no such oracles for \(\text{prp}\_partial\)).

It also provides oracles named \(\text{prefix}\_\text{suffix}\) where \(\text{prefix}\) is \(0\) or an identifier \(Ocoll\) from a collision "Ocoll: \(\text{forall } a_1: T_1, \ldots, a_n: T_n; r_1 \leftarrow T; M^n\" in \(collisions\_LHS\) and \(\text{suffix}\) is arbitrary. These oracles act like the oracle \(\text{prefix}\) when \(\text{specialname}\) is \(\text{rom}\) (resp. \(\text{prf}\) or \(\text{prp}\)), except that:

- When \(\text{suffix}\) starts with \(\text{leave}\), the right-hand side still uses calls to \(f\); it does not replace them with fresh random values. However, it still performs lookups 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 command, which can be either "no collisions" or statements such as "seq^+ (suffix) may collide with previous seq^+ (suffix)" separated by semi-colons (;).

1. A tuple of strings arg, 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, Pcoll_rand(T) ≤ 2^{-n}, that is, T has option pcoll1 with n ≥ n' where n' is set by set minAutoCol1Elim = pestn'; 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:

\[ \text{"Ocoll: forall } a_1 : T_1, \ldots, a_n : T_n ; r_1 \text{-}\text{R } T; M" \]

where \( T \) is the type of the result of \texttt{enc} and the simple term \( M \) uses the variables \( a_1, \ldots, a_n, r_1 \). In this case, CryptoVerif tries to simplify \( M \) assuming \( r_1 \) is a random value and \( a_1, \ldots, a_n \) 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 \( M\{f(\ldots)/r_1\} \) into \( N \) when the result of \( f(\ldots) \) is a fresh random value, in the generated cryptographic transformation. Obviously, when \( n = 0 \), \texttt{forall } a_1 : T_1, \ldots, a_n : T_n ; r_1 \text{-}\text{is omitted. The identifiers } O\text{coll are used to form the oracle names in the generated equivalence (see below); they must not contain } _, and must be different from 0 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:

\[ \text{"Oeq: forall } a_1 : T; r_1 \text{-}\text{R } T; r_1 = a_1" \]

Assuming \( a_1 \) does not depend on \( r_1 \), \( r_1 = a_1 \) simplifies into \texttt{false}, so \( f(\ldots) = a_1 \) is transformed into \texttt{false} in the generated cryptographic transformation, when the result of \( f(\ldots) \) 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 \textit{specialname} is \texttt{icm} or \texttt{sprp}, the generated equivalence provides the following oracles:

- Oracle 0\texttt{enc} evaluates \texttt{enc} on its arguments in the left-hand side, and performs a lookup into previous cleartexts (and local keys for \texttt{icm}) of calls to 0\texttt{enc} and 0\texttt{dec} in the right-hand side: it returns the previous ciphertext when the current arguments are equal to previous cleartexts (and local keys for \texttt{icm}) and otherwise it returns a fresh random value.

- Oracle 0\texttt{dec} evaluates \texttt{dec} on its arguments in the left-hand side, and performs a lookup into previous ciphertexts (and local keys for \texttt{icm}) of calls to 0\texttt{enc} and 0\texttt{dec} in the right-hand side: it returns the previous cleartext when the current arguments are equal to previous ciphertexts (and local keys for \texttt{icm}) and otherwise it returns a fresh random value.

- For each element of collisions\_LHS, oracles \texttt{Ocoll\_enc} and \texttt{Ocoll\_dec} evaluate \( M \) with \( r_i \) replaced with a call to \texttt{enc} (resp. \texttt{dec}) in left-hand side and uses the simplified form of \( M \) in the right-hand side.

When \textit{specialname} is \texttt{icm\_partial} or \texttt{sprp\_partial}, the generated equivalence provides oracles named \texttt{prefix\_middle\_suffix} where \texttt{prefix} is 0 or an identifier \texttt{Ocoll} from a collision in collisions\_LHS, \texttt{middle} is \texttt{enc} or \texttt{dec}, and \texttt{suffix} is arbitrary. These oracles act like the oracle \texttt{prefix\_middle} when \textit{specialname} is \texttt{icm} (resp. \texttt{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 \texttt{seq\_i\_suffix} may collide with previous \texttt{seq\_j\_suffix} separated by semi-colons (;). "no collisions" says that the arguments of two oracle calls are never allowed to collide. \texttt{suffix\_i} may collide with previous \texttt{suffix\_j}, \texttt{suffix\_n} says that the arguments of oracles with suffix \texttt{suffix\_i} (\( i \in \{1, \ldots, n\} \)) are allowed to collide with arguments of previous calls to oracles with suffix \texttt{suffix\_j} (\( j \in \{1, \ldots, n'\} \)). In the right-hand side, the oracles execute \texttt{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 \texttt{ev\_coll} does not happen, so the proof will fail.

The oracles with suffix \texttt{default} are generated by default. The other oracles are generated on demand when they are present in the terms: information of the \texttt{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 \texttt{default}.

The [\texttt{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 \texttt{crypto} command. Alternatively, an integer between brackets [\texttt{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 \[
\texttt{seq}([\texttt{vartype}]);\] \texttt{|(query)}\{\texttt{(query)}\}^*.

The query declaration indicates which security properties we would like to prove. CryptoVerif allows two variants of the syntax for queries:

- In the recommended syntax, the part \texttt{seq(\texttt{vartype});} is omitted, and the query declaration is of the form query \(Q_1; \ldots ; Q_n\). The variables in correspondence queries are explicitly quantified inside each query \(Q_i\), as explained below.

- In the other syntax, the query declaration is of the form query \(x_1:T_1, \ldots , x_n:T_n; Q_1; \ldots ; Q_n\). First, we declare the types of all variables \(x_1, \ldots , x_n\) that occur in correspondence queries that follow. (We use \(x_i \equiv N_i\) instead of \(x_i:T_i\) when \(x_i\) is of type \([1, N_i]\), where \(N_i\) is a parameter, declared by \texttt{param} \(N_i\).) Second, we give the queries \(Q_1, \ldots , Q_n\) themselves, without any \texttt{forall} \(x_1:T_1, \ldots , x_n:T_n\) and \texttt{exists} \(y_1:T_1' , \ldots , y_k:T_k'\). This form is not recommended because the quantifiers applied to the variables are less obvious to the user: variables that occur before \(\Rightarrow\) are universally quantified and variables that occur after \(\Rightarrow\) but not before \(\Rightarrow\) are existentially quantified just after \(\Rightarrow\).

The available queries \(Q_i\) are as follows:

- \texttt{secret} \(x\) \texttt{|[public\_varts l]} or \texttt{secret} \(x\) \texttt{|[public\_varts l]} \texttt{|[real\_or\_random]}: 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 \cite{2}, this is secrecy.

- \texttt{secret} \(x\) \texttt{|[public\_varts l]} \texttt{|[onesession]} or \texttt{secret} \(x\) \texttt{|[public\_varts l]} \texttt{|[real\_or\_random,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 \cite{2}, this is one-session secrecy.

- \texttt{secret} \(x\) \texttt{|[public\_varts l]} \texttt{|[reachability]}: shows that the adversary cannot compute any element of the array \(x\), even when it has access to the other elements of \(x\) and to the variables in \(l\). This query is allowed only when \(x\) is of a large type (or type declared with \texttt{pcollin} for \(n \geq n'\) with set \texttt{minAutoCollElim} = \texttt{pestn'}), that is, collisions can be eliminated between random values of that type. Otherwise, the adversary would have a non-negligible probability of finding the value of \(x\) just by guessing.

- \texttt{secret} \(x\) \texttt{|[public\_varts l]} \texttt{|[reachability,onesession]}: shows that the adversary cannot compute any element of the array \(x\), even when it has access to the variables in \(l\). This query is allowed only when \(x\) is of a large type (or type declared with \texttt{pcollin} for \(n \geq n'\) with set \texttt{minAutoCollElim} = \texttt{pestn'}), that is, collisions can be eliminated between random values of that type. Otherwise, the adversary would have a non-negligible probability of finding the value of \(x\) just by guessing.
This notion of secrecy as "the adversary cannot compute" is less common in the computational model than "the adversary cannot distinguish from random", but it is still used (e.g. in the property of one-wayness or in the computational Diffie-Hellman assumption).

- `secret x [public vars l] [cv_bit]`: shows that the adversary cannot guess the value of the boolean variable `x` significantly better than at random. It applies only when `x` is a boolean variable defined under no replication. When the process chooses `x` randomly and executes different code depending on whether `x` is true or false, this property also shows that the process with `x` true is indistinguishable from the process with `x` false.

The options `cv_omsession`, `cv_real_or_random`, `cv_reachability` are also allowed, and synonym of the similar options without `cv_` prefix. The only difference is that options that start with `cv_` apply to CryptoVerif only, while options that start neither with `cv_` nor with `pv_` apply to both CryptoVerif and ProVerif. All options starting with `pv_` are also allowed, but ignored: they are for ProVerif.

- `[forall x1:T1, . . . , xj:Tj;] M0 [public vars l]`, where in `M0`, `==>` is not allowed under `&&, ||`, or `exists`, and `||` and `exists` are allowed only after `==>`, so that after replacing variables bound by assignments by their value and moving existential quantifiers just after `==>`, `M0` is of form `M ==>[exists y1:T′1, . . . , yk:T′k] M′ or M`, where `M` and `M′` do not contain `==>`. The query `M` is an abbreviation for `M ==>[exists y1:T′1, . . . , yk:T′k] M′`, so we only need to consider `M ==>[exists y1:T1, . . . , yk:Tk] M′`. (We use `x1 <= N1` instead of `x1:T1` when `x1` is of type `[1, N1]`, where `N1` is a parameter, declared by `param N1`. We use the same notation for `yi`.)

The variables `x1, . . . , xj` are the variables that occur in `M`, and the variables `y1, . . . , yk` are the variables of `M′` that do not occur in `M`. (In particular, a universally quantified variable `x1` is not allowed to occur in `M′` and not in `M`.) CryptoVerif shows that, for all values of `x1, . . . , xj`, if `M` is true then there exist values of `y1, . . . , yk` 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(M1, . . . , Mn))`, or `inj-event(e(M1, . . . , Mn))` where `e` is an event declared by `event` and the `Mi` 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(M1, . . . , Mn)), inj-event(e(M1, . . . , Mn))`, 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(e1(M11, . . . , M1m1)) && . . . && inj-event(en(Mn1, . . . , Mnmn)) && . . . ==>` means that for each tuple of executed events `e1(M11, . . . , M1m1)` (executed `N1` times), . . . , `en(Mn1, . . . , Mnmn)` (executed `Nn` times), `M′` holds, considering that an event `inj-event(e′(M11, . . . , M1m1))` in `M′` holds when it has been executed at least `N1` times . . . `Nn` times. The `inj-event` marker must occur either 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 `cryptocommand` 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(x1, . . . , xn) {d1, . . . , dk}` defines a macro named `m`, with arguments `x1, . . . , xn`. This macro expands to the declarations `d1, . . . , dk`, 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, . . . , y_n) , expands the macro m by applying it to the arguments y_1, . . . , y_n. If the definition of the macro m is def m(x_1, . . . , x_n) {d_1, . . . , d_k}, then it generates d_1, . . . , d_k in which y_1, . . . , y_n are substituted for x_1, . . . , 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 ⊥.
- 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 ⊥. (The special element of bitstringbot that is not a bitstring.)
- The function if_fun takes three arguments, one of type bool and two arguments of the same type. It satisfies if_fun(true, x, y) = x and if_fun(false, x, y) = y.

The syntax of probability formulas allows parenthesesing 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, . . . , p_n) where P has been declared by proba P and p_1, . . . , p_n are probability formulas; this formula represents an unspecified probability depending on p_1, . . . , 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, . . . , i_n), where O is an oracle and i_1, . . . , 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, . . . , 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 every 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, . . . , p_n) designates the maximal length of the result of a call to f, where p_1, . . . , 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', . . . ), 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).
Finally, time(\(\hat{\text{Pcoll}}_2\text{rand}(T)\)) designates the maximum probability of collision between a random value \(X\) of type \(T\) chosen according to the default distribution \(D_T\) for type \(T\) (when \(T\) is nonuniform). This is also the maximum probability of choosing any given element of \(T\) in the default distribution for that type:

\[
\text{Pcoll}_2\text{rand}(T) = \max_{a \in T} \Pr[X = a]
\]

where \(X\) is chosen according to 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(\(\ldots\)) designates the runtime time of each elementary action of a game:

- \(\text{time}(f, p_1, \ldots, p_n)\) designates the maximal runtime of one call to function symbol \(f\), where \(p_1, \ldots, p_n\) represent the maximum length of the non-bounded arguments of \(f\).

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

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

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

- \(\text{time}(\text{=}(T, p_1, p_2))\) 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.
• \text{time(!)} or \text{time(foreach)} is the maximum time of an access to a replication index.
• \text{time([a])} is the maximum time of an array access with \text{n} indices.
• \text{time(&&)} is the maximum time of a boolean and.
• \text{time(!)} is the maximum time of a boolean or.
• \text{time(new T)} or \text{time(<-R T)} is the maximum time needed to choose a random number of type \text{T} according to the default distribution for type \text{T}.
• \text{time(newOracle)} is the maximum time to create a new private oracle.
• \text{time(if)} is the maximum time to perform a boolean test.
• \text{time(find n)} is the maximum time to perform one condition test of a find with \text{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.)

CryptoVerif checks the dimension of probability formulas.

4 channels Front-end

CryptoVerif also as an a channel frontend, that enables cross-compatibility with ProVerif. This frontend is only for the input file, the input process will be translated as an oracle definition and all subsequent games will be displayed in the oracle syntax.

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

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

\begin{align*}
\langle \text{declaration} \rangle \ast \text{process } & \langle \text{iprocess} \rangle \\
\langle \text{declaration} \rangle \ast \text{equivalence } & \langle \text{iprocess} \rangle \langle \text{iprocess} \rangle \text{ [public_vars seq (ident)]} \\
\langle \text{declaration} \rangle \ast \text{query_equiv } & \langle \text{ident} \rangle \langle (\text{ident}) \rangle \\
\langle \text{option} \rangle & \langle \text{group} \rangle \langle 1 \ldots 1 \langle \text{option} \rangle \langle \text{group} \rangle \rangle
\end{align*}

In addition to the declarations of the oracles front-end, the channels front-end allows the declaration \text{channel e\_1, \ldots, e\_n}, which declares communication channels \text{e\_1, \ldots, e\_n}.

The syntax of processes is given in Figure 8.

The calculus distinguishes two kinds of processes: input processes (\text{iprocess}) are ready to receive a message on a channel, output processes (\text{iprocess}) 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. The input process \text{proc}(M_1, \ldots, M_n) is replaced with \text{Q(M_1/x_1, \ldots, M_n/x_n)} when \text{proc} is declared by let \text{proc(x_1 : T_1, \ldots, x_n : T_n) = Q}, where \text{Q} is an input process. The terms \text{M_1, \ldots, M_n} must contain only variables, replication indices, and function applications. The output process \text{proc(M_1, \ldots, M_n)} is replaced with let \text{x_1 = M_1 in \ldots let x_n = M_n in P} when \text{proc} is declared by let \text{proc(x_1 : T_1, \ldots, x_n : T_n) = P}, where \text{P} is an output process.

In this front-end, communications are made over channels.

The input process \text{in(c[i\_1, \ldots, i\_n], p); P} declares a new process expecting an input from the attacker over the channel \text{c[i\_1, \ldots, i\_n]} where \text{c} is declared by \text{channel c} and \text{i\_1, \ldots, i\_n} are the current replication indices at the considered input. When the received message matches the given pattern \text{p}, the output process \text{P} is executed with the variables in the pattern \text{p} bound according to the received message. Otherwise, the input fails and controls returns to the attacker immediately, as if \text{yield} had been executed. Patterns \text{p} are as in the let process, except that variables in \text{p} that are not under a function symbol \text{f(\ldots)} must be declared with their type.

The output process \text{out(c[i\_1, \ldots, i\_n], N); Q} sends the output value \text{N} (truncated to the maximum length of bitstrings on channel \text{c}) to the attacker over a channel \text{c[i\_1, \ldots, i\_n]} where \text{c} is declared by
\( \text{channel} ::= \langle \text{ident} \rangle[\text{seq} \langle \text{ident} \rangle] \)

\( \text{ooprocess} ::= \langle \text{ident} \rangle[\text{seq} \langle \text{term} \rangle] \)
\( \quad | (\text{ooprocess}) \)
\( \quad | \text{yield} \)
\( \quad | \text{event} \langle \text{ident} \rangle[\text{seq} \langle \text{term} \rangle]; (\text{ooprocess}) \)
\( \quad | \langle \text{ident} \rangle \leftarrow \langle \text{ident} \rangle[; \langle \text{ooprocess} \rangle] \)
\( \quad | \text{new} \langle \text{ident} \rangle[; \langle \text{ooprocess} \rangle] \)
\( \quad | \langle \text{basicpat} \rangle \leftarrow \langle \text{term} \rangle[; \langle \text{ooprocess} \rangle] \)
\( \quad | \text{let} \langle \text{pattern} \rangle = \langle \text{term} \rangle \text{ in } (\text{ooprocess}) \quad | \text{else } (\text{ooprocess}) \)
\( \quad | \text{if} \langle \text{cond} \rangle \text{ then } (\text{ooprocess}) \quad | \text{else } (\text{ooprocess}) \)
\( \quad | \text{find}[\text{unique}] \langle \text{ndbranch} \rangle \text{ orfind } \langle \text{ndbranch} \rangle^* \text{ else } (\text{ooprocess}) \)
\( \quad | \text{insert } \langle \text{ident} \rangle(\text{seq} \langle \text{term} \rangle); (\text{ooprocess}) \)
\( \quad | \text{get}[\text{unique}] \langle \text{ident} \rangle\langle \text{seq} \langle \text{pattern} \rangle \text{ suchthat } \langle \text{term} \rangle \text{ in } (\text{ooprocess}) \quad | \text{else } (\text{ooprocess}) \)
\( \quad | \text{out}(\langle \text{channel} \rangle, \langle \text{term} \rangle)[; \langle \text{ioprocess} \rangle] \)

\( \langle \text{findbranch} \rangle ::= \text{seq} \langle \text{ident bound} \rangle \text{ suchthat } \langle \text{cond} \rangle \text{ then } (\text{ooprocess}) \)

\( \langle \text{ioprocess} \rangle ::= \langle \text{ident} \rangle[\text{seq} \langle \text{term} \rangle] \quad | (\langle \text{ioprocess} \rangle) \quad | 0 \)
\( \quad | \langle \text{ioprocess} \rangle \mid \langle \text{ioprocess} \rangle \)
\( \quad | \langle \text{!ident} \rangle \leftarrow \langle \text{ident} \rangle \langle \text{ioprocess} \rangle \)
\( \quad | \text{foreach } \langle \text{ident} \rangle \leftarrow \langle \text{ident} \rangle \text{ do } \langle \text{ioprocess} \rangle \)
\( \quad | \text{in}(\langle \text{channel} \rangle, \langle \text{pattern} \rangle)[; \langle \text{ooprocess} \rangle] \)

Figure 8: Grammar for processes (channels front-end)
channel \( c \) and \( i_1, \ldots, i_n \) are the current replication indices at the considered output. The inputs declared in the input process \( Q \) that follows the output are then made available; the attacker can send messages to them.

In inputs and outputs, the channels \( c[i_1, \ldots, i_n] \) where \( i_1, \ldots, i_n \) are the current replication indices at the considered input or output, can be abbreviated as \( c \). CryptoVerif automatically adds the current replication indices.

Due to the translation to oracles, there are multiple restrictions over the input and output process definitions:

- Each input must use a different channel, except in disjoint execution branches. The channel name is used in the translation to name the corresponding oracle.
- There cannot be two directly consecutive outputs. 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.
- All inputs on a given channel must have current replication indices of the same types and patterns and all outputs below these inputs must be of the same type, so that the resulting oracle is well typed. Tuples are considered of the same type when they are of the same arity and have elements of the same type.
- An output channel and an input channel both available at the same time can never be equal. It is recommended to use disjoint channels in parallel processes to ensure this.

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.

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

### 5 Summary of the Main Differences between the Two Front-ends

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

<table>
<thead>
<tr>
<th>channels</th>
<th>oracles</th>
</tr>
</thead>
<tbody>
<tr>
<td>input process</td>
<td>oracle definition</td>
</tr>
<tr>
<td>output process</td>
<td>oracle body</td>
</tr>
<tr>
<td><code>newChannel c</code></td>
<td><code>newOracle O</code></td>
</tr>
<tr>
<td><code>in(c, (x_1:T_1, ..., x_l:T_l)); P</code></td>
<td><code>O(x_1:T_1, ..., x_l:T_l) := P</code></td>
</tr>
<tr>
<td><code>out(c, (M_1, ..., M_l)); Q</code></td>
<td><code>return(M_1, ..., M_l); Q</code></td>
</tr>
<tr>
<td><code>proc(M_1, ..., M_l)</code></td>
<td><code>run proc(M_1, ..., M_l)</code></td>
</tr>
</tbody>
</table>

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.

### 6 Predefined Cryptographic Primitives

A number of standard cryptographic primitives, with and without cryptographic assumptions, are predefined in CryptoVerif. The definitions of these primitives are given as macros in the library file `default.ocvl` (or `default.cvl` for the `channels` front-end) that is automatically loaded at startup. Users do not need to redefine these primitives, they 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.
In addition to the default library, the library file `pq.ocvl` (or `pq.cvl` for the channel's front-end) also contains the subset of assumptions that are known to have instantiations against post-quantum attackers, which is essentially all assumptions from the default library except the Random Oracle Model, the Ideal Cipher Model, and the Diffie-Hellman based assumptions. One can load them by using the `-lib pq` command line argument. A CryptoVerif model only relying on such assumptions yields a post-quantum sound proof.

Many macros provided in the library have two versions: a standard one and one with suffix `_all_args`. The standard version is sufficient for most usages. The version with suffix `_all_args` uses additional arguments, e.g. types for randomness of probabilistic functions, which are useful only in exceptional cases.

Here is a list of the predefined primitives, sorted by categories.

### 6.1 Probabilistic symmetric encryption

The macros for symmetric encryption are defined in a modular fashion, where

- some macros only define a primitive without any assumption on its security, their names only contain the name of the primitive;
- some macros only define a security property, assuming the primitive is already defined; their names contain the `_prop_` keyword;
- some macros, building on the two previous kinds, directly declare a primitive along with the corresponding assumptions.

Note that the macros corresponding to security assumptions also define additional function symbols, for instance the macro `IND_CPA_prop_sym_enc` assumes a predefined `enc_r` encryption function, but also defines a new `enc_r'`. The core idea is that whenever the IND-CPA assumption is applied to a particular `enc_r`, we replace this occurrence by the newly defined function symbol, and we can then know that the assumption was already applied here, and avoid loops. This is a common feature of many assumptions to avoid loops in their applications.

The main macros to be used are of the third class. The most common macro is `IND_CPA_INT_CTXT_sym_enc`, which directly defines an IND-CPA and INT-CTXT probabilistic symmetric encryption. The macro `sym_enc_all_args` defines a probabilistic symmetric encryption scheme, without any assumption on its security. Each macro `*_prop_sym_enc` defines security properties of symmetric encryption independently of each other. They should be used as follows:

- `expand sym_enc_all_args`
- optionally, one of
  - `expand IND_CPA_prop_sym_enc`
  - `expand INDdollar_CPA_prop_sym_enc`
  - `expand IND_CCA2_prop_sym_enc`
- optionally, one of
  - `expand INT_CTXT_prop_sym_enc`
  - `expand INT_PTXT_prop_sym_enc`

The details of each macro are listed below.

- `expand sym_enc_all_args(key, cleartext, ciphertext, enc_seed, enc, enc_r, dec, injbot)`, defines a probabilistic symmetric encryption scheme, without any assumption on its security.
  - `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_r` without mentioning the length of the key), typically `fixed` and `large`.
  - `cleartext` is the type of cleartexts.
\textbf{ciphertext} is the type of ciphertexts.

\texttt{enc\_seed} is the type of random coins for encryption (must be bounded).

\texttt{enc\{cleartext, key\}}: ciphertext is the encryption function. Internally, it generates random coins, so that it is probabilistic.

\texttt{enc \_r\{cleartext, key, enc \_seed\}}: ciphertext: encryption function that takes coins as argument.

\texttt{dec\{ciphertext, key\}}: bitstringbot is the decryption function; it returns bottom when decryption fails.

\texttt{injbot\{cleartext\}}: bitstringbot is the natural injection from cleartext to bitstringbot.

The types \texttt{key}, \texttt{cleartext}, \texttt{ciphertext} and \texttt{enc \_seed} must be declared before this macro is expanded. The functions \texttt{enc}, \texttt{enc \_r}, \texttt{dec} and \texttt{injbot} are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

- \texttt{expand sym\_enc\{key, cleartext, ciphertext, enc, dec, injbot\}} is similar to the above, but hides the random coins for encryptions and \texttt{enc \_r}.

The arguments have the same meaning as in \texttt{sym\_enc\_all\_args}. This macro defines a new type for the \texttt{enc \_seed} internally.

- \texttt{expand IND\_CPA\_prop\_sym\_enc\{key, cleartext, ciphertext, enc \_seed, enc, enc \_r, enc \_r', Z, Penc\}} defines the IND-CPA (indistinguishable under chosen plaintext attacks) property over a probabilistic symmetric encryption scheme.

The arguments already present in \texttt{sym\_enc\_all\_args} have the same meaning, but must be predefined.

\texttt{enc \_r'}\{cleartext, key, enc \_seed\}: ciphertext is the symbol that replaces \texttt{enc \_r} after game transformation.

\texttt{Z\{cleartext\}}: cleartext is the function that returns for each cleartext a cleartext of the same length consisting only of zeroes.

\texttt{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 \texttt{key}, \texttt{cleartext}, \texttt{ciphertext}, \texttt{enc \_seed}, the function \texttt{enc \_r}, and the probability \texttt{Penc} must be declared before this macro is expanded. The functions \texttt{enc \_r'} and \texttt{Z} are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

The argument \texttt{enc} is intuitively meant to be the encryption function, but in fact used only for naming the equivalence statement. This macro defines the equivalence named \texttt{ind\_cpa\{enc\}} for use in the crypto command in interactive proofs (see Section 7).

- \texttt{expand IND\$\_CPA\_prop\_sym\_enc\{key, cleartext, ciphertext, enc \_seed, cipher \_stream, enc, enc \_r, Z, enc \_len, truncate, Penc\}} defines the IND$\$-CPA (indistinguishable under chosen plaintext attacks) property over a probabilistic symmetric encryption scheme. IND$\$-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 string of the same length.

The arguments already present in \texttt{sym\_enc\_all\_args} have the same meaning, but must be predefined.

\texttt{cipher \_stream} is the type of unbounded streams (must be nonuniform).

\texttt{Z\{cleartext\}}: cleartext is the function that returns for each cleartext a cleartext of the same length consisting only of zeroes.

\texttt{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.

\texttt{truncate\{cipher \_stream, ciphertext\}}: ciphertext is the function such that \texttt{truncate\{s, x\}} is the truncation of \(s\) to the length of \(x\), where \(s\) is a stream of unbounded length.

\texttt{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`, `enc_seed`, `cipher_stream`, the function `enc_r`, and the probability `Penc` must be declared before this macro is expanded. The functions `Z`, `enc_len` and `truncate` are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

The argument `enc` is intuitively meant to be the encryption function, but it is in fact used only for naming the equivalence statement. This macro defines the equivalence `ind_dollar_cpa(enc)`.

- **expand IND_CCA2_prop_sym_enc**(`key`, `cleartext`, `ciphertext`, `enc_seed`, `enc`, `enc_r`, `dec`, `dec'`, `injbot`, `Z`, `Penc`). defines the IND-CCA2 (indistinguishable under adaptive chosen ciphertext attacks) property over a probabilistic symmetric encryption scheme.

The arguments already present in `expand sys_enc_all_args` have the same meaning, but must be predefined.

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

`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`, `enc_seed`, the functions `enc_r`, `dec`, `injbot`, and the probability `Penc` must be declared before this macro is expanded. The functions `enc_r'`, `Z`, and `dec'` are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

The argument `enc` is intuitively meant to be the encryption function, but it is in fact used only for naming the equivalence statements. 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. Users should map the occurrences of encryption that they want to transform to oracle `Oenc`, the ones they want 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 INT_CTXT_prop_sym_enc**(`key`, `cleartext`, `ciphertext`, `enc_seed`, `enc`, `enc_r`, `dec`, `injbot`, `Penctxt`). defines the INT-CTXT (ciphertext integrity) property over a probabilistic symmetric encryption scheme.

The arguments already present in `sys_enc_all_args` have the same meaning, but must be predefined.

`Penctxt(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 `enc_seed`, the functions `enc_r`, `dec`, `injbot`, and the probability `Penctxt` must be declared before this macro is expanded.

The argument `enc` is intuitively meant to be the encryption function, but it is in fact used only for naming the equivalence statements. This macro defines the equivalences named `int_ctxt(enc)`, and `int_ctxt_corrupt(enc)` for use in the `crypto` command (see Section 7). 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 INT_PTXT_prop_sym_enc**(`key`, `cleartext`, `ciphertext`, `enc_seed`, `enc`, `enc_r`, `dec`, `dec'`, `injbot`, `Penctxt`). defines the INT-PTXT (plaintext integrity) property over a probabilistic symmetric encryption scheme.
6.2 Symmetric encryption with a nonce

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

\[ \Pr_{\text{ptxt}}(t, N, N', Nu', l, l') \] 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`, `enc_seed`, the functions `enc_r`, `dec`, `injbot`, and the probability \( \Pr_{\text{ptxt}} \) must be declared before this macro is expanded. The function `dec'` is declared by this macro. It must not be declared elsewhere, and it can be used only after expanding the macro.

The argument `enc` is intuitively meant to be the encryption function, but it is in fact used only for naming the equivalence statements. This macro defines the equivalences named `ind_ptxt` and `ind_ptxt_corrupt_partial`, for use in the `crypto` command (see Section 7). While the equivalence `ind_ptxt` replaces all decryption with lookups in encryption queries, the equivalence `ind_ptxt_corrupt_partial` 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, users should map the occurrences of decryption that they want to transform to oracle `Odec`, the ones they want 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 `IND_CPA_sym_enc_all_args(key, cleartext, ciphertext, enc_seed, enc, enc_r, enc_r', dec, injbot, Z, Penc)`
- expand `IND_CPA_INT_CTXT_sym_enc_all_args(key, cleartext, ciphertext, enc_seed, enc, enc_r, enc_r', dec, injbot, Z, Penc, Pencxt)`
- expand `INDollar_CPA_sym_enc_all_args(key, cleartext, ciphertext, enc_seed, cipher_stream, enc, enc_r, dec, injbot, Z, enc_len, truncate, Penc)`
- expand `INDollar_CPA_INT_CTXT_sym_enc_all_args(key, cleartext, ciphertext, enc_seed, cipher_stream, enc, enc_r, dec, injbot, Z, enc_len, truncate, Penc, Pencxt)`

all define a probabilistic symmetric encryption scheme, expanding `sym_enc_all_args` and adding the corresponding security assumptions, by expanding the corresponding \( *\_prop_sym_enc \) macros. Each argument can be understood by looking at the underlying macros.

- expand `IND_CPA_sym_enc(key, cleartext, ciphertext, enc, dec, injbot, Z, Penc)`
- expand `IND_CPA_INT_CTXT_sym_enc(key, cleartext, ciphertext, enc, dec, injbot, Z, Penc, Pencxt)`
- expand `INDollar_CPA_sym_enc(key, cleartext, ciphertext, cipher_stream, enc, dec, injbot, Z, enc_len, truncate, Penc)`
- expand `INDollar_CPA_INT_CTXT_sym_enc(key, cleartext, ciphertext, cipher_stream, enc, dec, injbot, Z, enc_len, truncate, Penc, Pencxt)`

are all similar to the above, but hides the random coins for encryptions and `enc_r`.

These macros define a new type for the `enc_seed` internally.

6.2 Symmetric encryption with a nonce

- expand `sym_enc_nonce(key, cleartext, ciphertext, nonce, enc, dec, injbot)`, defines a symmetric encryption with a nonce, without any assumption on its security. This is similar to the `sym_enc_all_args` macro, but it uses 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_r` 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.

\texttt{enc(cleartext, key, nonce)} : ciphertext is the encryption function.

\texttt{dec(ciphertext, key, nonce)} : bitstringbot is the decryption function; it returns \texttt{bottom} when decryption fails.

\texttt{injbot(cleartext)} : bitstringbot is the natural injection from cleartext to bitstringbot.

The types \textit{key}, \textit{cleartext}, \textit{ciphertext} and \textit{nonce} must be declared before this macro is expanded. The functions \texttt{enc}, \texttt{dec} and \texttt{injbot} are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

- **expand** \texttt{IND\_CPA\_sym\_enc\_nonce\_all\_args(key, cleartext, ciphertext, nonce, enc, enc', dec, injbot, Z, Penc)}.
- **expand** \texttt{IND\_CPA\_sym\_enc\_nonce(key, cleartext, ciphertext, nonce, enc, dec, injbot, Z, Penc)}.
- **expand** \texttt{IND\_CPA\_INT\_CTX\_sym\_enc\_nonce\_all\_args(key, cleartext, ciphertext, nonce, enc, enc', dec, injbot, Z, Penc, Pencctxt)}.
- **expand** \texttt{IND\_CPA\_INT\_CTX\_sym\_enc\_nonce(key, cleartext, ciphertext, nonce, enc, dec, injbot, Z, Penc, Pencctxt)}.
- **expand** \texttt{IND\_dollar\_CPA\_sym\_enc\_nonce(key, cleartext, ciphertext, nonce, cipher\_stream, enc, dec, injbot, Z, enc\_len, truncate, Penc)}.
- **expand** \texttt{IND\_dollar\_CPA\_INT\_CTX\_sym\_enc\_nonce(key, cleartext, ciphertext, nonce, cipher\_stream, enc, dec, injbot, Z, enc\_len, truncate, Penc, Pencctxt)}.

\subsection{6.3 AEAD (authenticated encryption with additional data)}

- **expand** \texttt{AEAD\_no\_assumption\_all\_args(key, cleartext, ciphertext, add\_data, enc\_seed, enc, enc\_r, dec, injbot)}.

\textit{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 \texttt{enc} without mentioning the length of the key), typically \texttt{fixed} and \texttt{large}.

\textit{cleartext} is the type of cleartexts.

\textit{ciphertext} is the type of ciphertexts.

\textit{add\_data} is the type of additional data.

\textit{enc\_seed} is the type of random coins for encryption, must be bounded.

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

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

\texttt{dec(ciphertext, add\_data, key)} : bitstringbot is the decryption function; it returns \texttt{bottom} when decryption fails.

\texttt{injbot(cleartext)} : bitstringbot is the natural injection from cleartext to bitstringbot.

The types \textit{key}, \textit{cleartext}, \textit{ciphertext}, \textit{enc\_seed}, and \textit{add\_data} must be declared before this macro is expanded. The functions \texttt{enc}, \texttt{enc\_r}, \texttt{dec} and \texttt{injbot} are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

- **expand** \texttt{AEAD\_no\_assumption(key, cleartext, ciphertext, add\_data, enc, dec, injbot)}.

The arguments have the same meaning as in \texttt{AEAD\_no\_assumption\_all\_args}. This macro defines a new type for the \textit{enc\_seed} internally.
6.4 AEAD (authenticated encryption with additional data) with a nonce.

- **expand AEAD_no_assumption_all_args(key, cleartext, ciphertext, add_data, enc_seed, enc, enc_r, enc_r', dec, injbot, Z, Penc, Pencctxt)**. defines an authenticated encryption scheme with additional data without any assumption on its security, similarly to AEAD_no_assumption_all_args, but 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.
6.5 Permutation Ciphers

- expand permutation_cipher(key, blocksize, enc, dec), defines a permutation cipher without any assumption on its security.

  - 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.
  - enc(blocksize, key): blocksize is the encryption function.
  - dec(blocksize, key): blocksize is the decryption function.

  The types key and blocksize 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.

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

  The arguments already present in permutation_cipher have the same meaning.

  Note that 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.

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

  The arguments already present in permutation_cipher have the same meaning.

  Note that 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.
\[ P_{enc}(t, N, N') \] is the probability of breaking the SPRP property in time \( t \) for one key, \( N \) encryption queries, and \( N' \) decryption queries.

The types \( \text{key} \), \( \text{blocksize} \) and the probability \( P_{enc} \) must be declared before this macro is expanded. The functions \( \text{enc} \) and \( \text{dec} \) 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 \( \text{sprp}(\text{enc}) \) and \( \text{sprp}_{\text{partial}}(\text{enc}) \) for use in the crypto command (see Section 7). These equivalences are generated via \( \text{equiv} \ldots \text{special} \); the crypto command therefore supports special arguments \( \text{collisions}_{\text{LHS}} \) and, for \( \text{sprp}_{\text{partial}}(\text{enc}) \), collision matrix. See the explanation of the \( \text{collisions}_{\text{LHS}} \) argument, the collision matrix, and the oracles present in these equivalences in the documentation of \( \text{equiv} \ldots \text{special} \).

- **expand two_keys_permutation_cipher** (\( \text{cipherkey} \), \( \text{key} \), \( \text{blocksize} \), \( \text{enc} \), \( \text{dec} \), \( \text{enc}_{\text{dec} \text{oracle}} \), \( qE \), \( qD \) ) defines a permutation cipher with two keys, used as a version of the ideal cipher model without any assumption on its security.

- **expand ICM_cipher** (\( \text{cipherkey} \), \( \text{key} \), \( \text{blocksize} \), \( \text{enc} \), \( \text{dec} \), \( \text{enc}_{\text{dec} \text{oracle}} \), \( qE \), \( qD \) ) defines a block cipher in the ideal cipher model.

The arguments are the same as in **two_keys_permutation_cipher**.

This macro defines equivalences named \( \text{icm}(\text{enc}) \) and \( \text{icm}_{\text{partial}}(\text{enc}) \) for use in the crypto command (see Section 7). These equivalences are generated via \( \text{equiv} \ldots \text{special} \); the crypto command therefore supports special arguments \( \text{collisions}_{\text{LHS}} \) and, for \( \text{icm}_{\text{partial}}(\text{enc}) \), collision matrix. See the explanation of the \( \text{collisions}_{\text{LHS}} \) argument, the collision matrix, and the oracles present in these equivalences in the documentation of \( \text{equiv} \ldots \text{special} \).

### 6.6 Deterministic MACs

- **expand det_mac** (\( \text{mkkey} \), \( \text{macinput} \), \( \text{macres} \), \( \text{mac} \), \( \text{check} \) ) defines a deterministic MAC (message authentication code), without any assumption on its security.
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.

\[ \text{mac} (\text{macinput}, mkey) : \text{macres} \] is the MAC function.

\[ \text{check} (\text{macinput}, mkey, \text{macres}) : \text{bool} \] is the verification function. For deterministic MACs, the verification can be done by recomputing the MAC. This macro transforms tests \( \text{mac}(k,m) = m' \) into \( \text{check}(k,m,m') \), using an equation named check_to_mac(mac), so that the MAC verification can also be written \( \text{mac}(k,m) = m' \). The use command (see Section 7) allows users to disable or revert this equation.

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

- expand SUF_CMA_det_mac_all_args(mkey, macinput, macres, mac, 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 it has already seen a MAC. Such a forgery is guaranteed to be hard for a SUF-CMA MAC. When the verification is done by recomputing the MAC, an UF-CMA MAC is always SUF-CMA, so we model only SUF-CMA deterministic MACs.

The arguments already present in det_mac have the same meaning.

\( m' \) is the symbol that replaces mac after game transformation.

\( Pmac(t, N, N', Nu', l) \) 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, mac', 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 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. Users should map the occurrences they want to transform to the oracle Ocheck and the ones they do not want to transform to the oracle Ocheck_unchanged.

- expand SUF_CMA_det_mac(mkey, macinput, macres, mac, check, Pmac). is similar to the above, but hides mac'.

6.7 Probabilistic MACs

- expand proba_mac_all_args(mkey, macinput, macres, mac_seed, mac, mac_r, check). defines a probabilistic MAC (message authentication code), without any assumption on its security.

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

mac(macinput, mkey) : macres is the MAC function that generates coins internally.

mac_r(macinput, mkey, mac_seed) : macres is the MAC function that takes coins as argument
(instead of generating them internally).

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

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

• expand proba_mac(mkey, macinput, macres, mac, check), is similar to the above, but hides the
random coins for MACs and mac_r.
The arguments have the same meaning as in proba_mac_all_args. This macro defines a new type
for the mac_seed internally.

• expand UF_CMA_proba_mac_all_args(mkey, macinput, macres, mac_seed, mac, mac_r,
mac_r', check, check', Pmac), defines a UF-CMA (unforgeable under chosen message attacks)
probabilistic MAC (message authentication code).
The arguments already present in proba_mac_all_args have the same meaning.
mac_r' and check' are the symbols that replace mac_r and check respectively after game transform-
ation.

Pmac(t, N, N', Nu, l) is the probability of breaking the UF-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 mac_seed and the probability Pmac must be declared
before this macro is expanded. The functions mac, mac_r, mac_r', check, 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(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(mkey, macinput, macres, mac, check, Pmac). is similar to the above,
but hides the random coins for MACs and the functions mac_r, mac_r', and check'.
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.

• expand SUF_CMA_proba_mac_all_args(mkey, macinput, macres, mac_seed, mac, mac_r,
mac_r', check, Pmac), defines a SUF-CMA (strongly unforgeable under chosen message attacks)
probabilistic MAC (message authentication code).
The arguments already present in proba_mac_all_args have the same meaning.
mac_r' is the symbol that replaces mac_r after game transformation.

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 mac_seed and the probability Pmac must be declared
before this macro is expanded. The functions mac, mac_r, mac_r', 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 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(mkey, macinput, macres, mac, check, Pmac). is similar to the above, but hides the random coins for MACs and the functions mac_r and mac_r'.

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.

6.8 Public-key Encryption

- expand public_key_enc_all_args(keyseed, pkey, skey, cleartext, ciphertext, enc_seed, skgen, pkgen, enc, enc_r, dec, injbot, Pkeycoll). defines a probabilistic public-key encryption scheme, without any assumption on its security.

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.

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

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.

enc_r(cleartext, pkey, enc_seed) : ciphertext is the encryption function that takes coins as argument (instead of generating them internally).

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.

Pkeycoll is the maximal probability of generating a given public key.

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

- expand public_key_enc(keyseed, pkey, skey, cleartext, ciphertext, skgen, pkgen, enc, enc_r, injbot, Pkeycoll). is similar to the above, but hides the random coins for encryptions and enc_r.

The arguments have the same meaning as in public_key_enc_all_args. This macro defines a new type for the enc_seed internally.

- expand IND_CPA_public_key_enc_all_args(keyseed, pkey, skey, cleartext, ciphertext, enc_seed, skgen, skgen', pkgen, pkgen', enc, enc_r, enc_r', dec, injbot, Z, Penc, Pkeycoll). defines a IND-CPA (indistinguishable under adaptive chosen plaintext attacks) probabilistic public-key encryption scheme.

The arguments already present in public_key_enc_all_args have the same meaning.

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

\( Penc(t) \) is the probability of breaking the IND-CPA property in time \( t \) for one key.

The types keyseed, pkey, skey, cleartext, ciphertext, enc_seed, and the probabilities \( Penc, Pkeycoll \) must be declared before this macro is expanded. The functions \( skgen, skgen', pkgen, pkgen', enc, enc, enc_r, enc_r', dec, injbot \), 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 \( \text{ind}_\text{cpa}(\text{enc}) \) and \( \text{ind}_\text{cpa}_\text{partial}(\text{enc}) \) for use in the \text{crypto} command (see Section 7). The equivalence \( \text{ind}_\text{cpa}_\text{partial}(\text{enc}) \) 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** \( \text{IND}_\text{CPA}\_\text{public}\_\text{key}\_\text{enc}\_\text{all}\_\text{args}(\text{keyseed}, \text{pkey}, \text{skey}, \text{cleartext}, \text{ciphertext}, \text{skgen}, \text{pkgen}, \text{enc}, \text{dec}, \text{injbot}, \text{Z}, \text{Penc}, \text{Pkeycoll}) \) is similar to the above, but hides the random coins for encryptions and the functions \( \text{pkgen}', \text{skgen}', \text{enc}_r, \text{and enc}_r' \). The arguments have the same meaning as in \( \text{IND}_\text{CPA}\_\text{public}\_\text{key}\_\text{enc}\_\text{all}\_\text{args} \). This macro defines a new type for the \( \text{enc}_\text{seed} \) internally.

- **expand** \( \text{IND}_\text{CCA2}\_\text{public}\_\text{key}\_\text{enc}\_\text{all}\_\text{args}(\text{keyseed}, \text{pkey}, \text{skey}, \text{cleartext}, \text{ciphertext}, \text{skgen}, \text{pkgen}, \text{enc}, \text{dec}, \text{injbot}, \text{Z}, \text{Penc}, \text{Pkeycoll}) \) defines an IND-CCA (indistinguishable under adaptive chosen ciphertext attacks) probabilistic public-key encryption scheme. The arguments already present in \( \text{IND}_\text{CPA}\_\text{public}\_\text{key}\_\text{enc}\_\text{all}\_\text{args} \) have the same meaning, except that \( Penc(t,N) \) is the probability of breaking the IND-CCA2 property in time \( t \) for one key and \( N \) decryption queries.

\( \text{dec'} \) is the symbol that replaces \( \text{dec} \) after game transformation.

This macro defines the equivalences named \( \text{ind}_\text{cca2}(\text{enc}) \) and \( \text{ind}_\text{cca2}_\text{partial}(\text{enc}) \) for use in the \text{crypto} command (see Section 7). The equivalence \( \text{ind}_\text{cca2}_\text{partial}(\text{enc}) \) 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** \( \text{IND}_\text{CCA2}\_\text{public}\_\text{key}\_\text{enc}\_\text{all}\_\text{args}(\text{keyseed}, \text{pkey}, \text{skey}, \text{cleartext}, \text{ciphertext}, \text{skgen}, \text{pkgen}, \text{enc}, \text{dec}, \text{injbot}, \text{Z}, \text{Penc}, \text{Pkeycoll}) \) is similar to the above, but hides the random coins for encryptions and the functions \( \text{pkgen}', \text{skgen}', \text{enc}_r, \text{enc}_r', \text{dec}, \text{dec'}, \text{injbot}, \text{Z}, \text{Penc}, \text{Pkeycoll} \). The arguments have the same meaning as in \( \text{IND}_\text{CCA2}\_\text{public}\_\text{key}\_\text{enc}\_\text{all}\_\text{args} \). This macro defines a new type for the \( \text{enc}_\text{seed} \) internally.

### 6.9 Deterministic signatures

- **expand** \( \text{det}\_\text{signature}(\text{keyseed}, \text{pkey}, \text{skey}, \text{signinput}, \text{signature}, \text{skgen}, \text{pkgen}, \text{sign}, \text{check}, \text{Pkeycoll}) \) defines a deterministic signature scheme, without any assumption on its security.

\( \text{keyseed} \) 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 \( \text{pkgen} \) without mentioning the length of the key), typically fixed and large.

\( \text{pkey} \) is the type of public keys, must be bounded.

\( \text{skey} \) is the type of secret keys, must be bounded.

\( \text{signinput} \) is the type of signature inputs.

\( \text{signature} \) is the type of signatures.

\( \text{skgen(keyseed)} : \text{skey} \) is the secret key generation function.

\( \text{pkgen(keyseed)} : \text{pkey} \) is the public key generation function.
sign(signinput, skey) : signature is the signature function.
check(signinput, pkey, signature) : bool is the verification function.
Pkeycoll is the maximal probability of generating a given public key.
The types keyseed, pkey, skey, signinput, signature and the probability Pkeycoll 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.

- expand UF_CMA_det_signature_all_args(keyseed, pkey, skey, signinput, signature, skgen, skgen', pkgen, pkgen', sign, sign', check, check', Psign, Pkeycoll), defines a UF-CMA (unforgeable under chosen message attacks) deterministic signature scheme.

The arguments already present in det_signature have the same meaning.
pkgen', skgen', sign', and check' are the symbols that replace pkgen, skgen, sign and check respectively after game transformation.

Psigt(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.
The types keyseed, pkey, skey, signinput, signature and the probabilities Psigt, Pkeycoll must be declared before this macro is expanded. The functions skgen, pkgen, sign, check, pkgen', skgen', 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(keyseed, pkey, skey, signinput, signature, skgen, pkgen, sign, check, Psign, Pkeycoll), is similar to the above, but hiding pkgen', skgen', sign' and check'.
The arguments have the same meaning as in UF_CMA_det_signature_all_args.

- expand SUF_CMA_det_signature_all_args(keyseed, pkey, skey, signinput, signature, skgen, skgen', pkgen, pkgen', sign, sign', check, check', Psign, Psigncoll), defines a SUF-CMA (strongly unforgeable under chosen message attacks) deterministic signature scheme.

The arguments are the same as for UF_CMA_det_signature, expect that Psign is the probability of breaking the SUF-CMA property.
The difference between a UF-CMA signature and a SUF-CMA signature is that, for a UF-CMA signature, the adversary may easily forge a new signature for a message for which it has already seen a signature. Such a forgery is guaranteed to be hard for a SUF-CMA 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), similarly to UF_CMA_det_signature_all_args.

- expand SUF_CMA_det_signature(keyseed, pkey, skey, signinput, signature, skgen, pkgen, sign, check, Psign, Pkeycoll), is similar to the above, but hiding pkgen', skgen', sign' and check'.
The arguments have the same meaning as in SUF_CMA_det_signature_all_args.
6.10 Probabilistic signatures

- **expand proba_signature_all_args(keyseed, pkey, sk, signinput, signature, sign_seed, skgen, pkgen, sign, sign_r, check, Pkeycoll)** defines a probabilistic signature scheme, without any assumption on its security.

  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.

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

  signinput is the type of signature inputs.

  signature is the type of signatures.

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

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

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

  sign(signinput, sk) : signature is the signature function.

  sign_r(signinput, sk, sign_seed) : signature is the signature function that takes coins as argument (instead of generating them internally).

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

  Pkeycoll is the maximal probability of generating a given public key.

The types keyseed, pkey, sk, signinput, signature, sign_seed and the probability Pkeycoll must be declared before this macro is expanded. The functions skgen, pkgen, sign, sign_r, and check are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.

- **expand proba_signature(keyseed, pkey, sk, signinput, signature, skgen, pkgen, sign, check, Pkeycoll)** is similar to the above, but hides the random coins for signatures and sign_r.

  The arguments have the same meaning as in proba_signature_all_args. This macro defines a new type for the sign_seed internally.

- **expand UF_CMA_proba_signature_all_args(keyseed, pkey, sk, signinput, signature, sign_seed, skgen, skgen', pkgen, pkgen', sign, sign_r, sign_r', check, check', Psign, Pkeycoll)** defines a UF-CMA (strongly unforgeable under chosen message attacks) probabilistic signature scheme.

  The arguments already present in proba_signature_all_args have the same meaning.

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

  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.

  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), similarly to UF_CMA_det_signature_all_args.

- **expand UF_CMA_proba_signature(keyseed, pkey, sk, signinput, signature, skgen, pkgen, sign, check, Psign, Pkeycoll)** is similar to the above, but hiding pkgen', skgen', sign_r' and check'.

  The arguments have the same meaning as in UF_CMA_proba_signature_all_args. The arguments are the same as for UF_CMA_det_signature, but the signature function internally generates random coins, so that it is probabilistic as for proba_signature_all_args.
6.11 Key Encapsulation Mechanism

- **expand SUF_CMA_proba_signature_all_args**(*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *sign_seed*, *skgen*, *skgen'*, *pkgen*, *pkgen'*, *sign_*, *sign_r*, *check*, *check'*, *Psign*, *Pkeycoll*).

This macro defines a SUF-CMA (strongly unforgeable under chosen message attacks) probabilistic signature scheme.

- **expand SUF_CMA_proba_signature**(*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *sign_seed*, *skgen*, *skgen'*, *pkgen*, *pkgen'*, *sign_*, *sign_r*, *check*, *check'*, *Psig_*, *Pkeycoll*).

The arguments have the same meaning as in SUF_CMA_proba_signature_all_args. The arguments are the same as for SUF_CMA_det_signature, but the signature function internally generates random coins, so that it is probabilistic as for SUF_CMA_proba_signature_all_args.

- **expand KEM**(*keyseed*, *pkey*, *skey*, *encapseed*, *sec*, *ciphertext*, *encapoutput*, *pkgen*, *skgen*, *encap_*, *pair*, *get_*, *decap*, *injbot*, *Pkeycoll*, *Pctxtcoll*)

defines a Key Encapsulation Mechanism (KEM) without any assumption on its security.

- **expand SUF_CMA_det_signature_all_args**(*keyseed*, *pkey*, *skey*, *signinput*, *signature*, *sign_seed*, *skgen*, *skgen'*, *pkgen*, *pkgen'*, *sign_*, *sign_r*, *check*, *check'*, *Psig_*, *Pkeycoll*).

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), similarly to UF_CMA_det_signature_all_args.

**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.

**encapseed** is the type of random coins for encryption, must be bounded.

**sec** is the type of the encapsulated secrets, must be bounded.

**ciphertext** is the type of ciphertexts.

**encapoutput** is the type of pairs of encapsulated secrets and ciphertexts.

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

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

**encap(pkgen)**: **encapoutput** is the full encryption function, generating random coins internally.

**encap_**(pkey, encapseed): **encapoutput** is the encryption function that takes random coins as argument.

**pair(sec, ciphertext)**: **encapoutput** is the pair construction function.

**get_(sec, encapoutput)**: **sec** extracts the secret from the result of the encryption function.

**get_(ciphertext, encapoutput)**: **ciphertext** extracts the ciphertext from the result of the encryption function.

**decap(ciphertext, skey)**: **bitstringbot** is the ciphertext decapsulation function; it returns bottom when decapsulation fails.

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

**Pkeycoll** is the maximal probability of generating a given public key.

**Pctxtcoll** is the maximal probability that encapsulation generates a given ciphertext.

The types keyseed, pkey, skey, encapseed, sec, ciphertext, encapoutput and the probabilities Pkeycoll and Pctxtcoll must be declared before this macro is expanded. The functions pkgen, skgen, encap, encap_, pair, get_, decap, and injbot are declared by this macro. They must not be declared elsewhere, and they can be used only after expanding the macro.
• expand IND_CPA_KEM(keyseed, pkey, skey, encapseed, sec, ciphertext, encapoutput, pkgen, skgen, encap, encap_r, pair, get_sec, get_ct, decap, injbot, Penc, Pkeycoll, Pctxtcoll). defines a IND-CPA (indistinguishable under chosen plaintext attacks) KEM.

The arguments already present in KEM have the same meaning.

Penc(t, Nk, Ne) is the probability of breaking the IND-CPA property in time t for Nk keys and Ne encryption queries.

This macro defines the equivalence named ind_cpa(encaps) for use in the crypto command (see Section 7).

• expand IND_CCA2_KEM(keyseed, pkey, skey, encapseed, sec, ciphertext, encapoutput, pkgen, skgen, encap, encap_r, pair, get_sec, get_ct, decap, injbot, Penc, Pkeycoll, Pctxtcoll). defines a IND-CCA2 KEM.

The arguments already present in KEM have the same meaning.

Penc(t, Nk, Ne, Nd) is the probability of breaking the IND-CCA2 property in time t for Nk keys, Ne encryption queries and Nd decryption queries.

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

6.12 Hash functions

• expand HiddenKey_hash(key, hashinput, hashoutput, hash, hashoracle, qH). defines a hash function without security assumption and whose key is hidden. (The adversary can call the hash function via an oracle.)

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 run hashoracle(hk) (hashoracle(hk) in the channels front-end) 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.

• expand PublicKey_hash(key, hashinput, hashoutput, hash, hashoracle). defines a hash function without security assumption and whose key is public. (It is given to the adversary.)

The arguments have the same meaning as in HiddenKey_hash, except that hashoracle(hk) leaks hk to the adversary, and qH is absent: one cannot count the calls to the hash function, because the adversary can evaluate it without interacting with the honest process.

WARNING: The key must be generated once and for all at the beginning of the game and must be made available to the adversary, by including run hashoracle(hk) (hashoracle(hk) in the channels front-end) in the executed process, where hk is the key.

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

The arguments are the same as in HiddenKey_hash.
This macro defines equivalences named \texttt{rom}(\texttt{hash}) and \texttt{rom\_partial}(\texttt{hash}) for use in the \texttt{crypto} command (see Section 7). These equivalences are generated via \texttt{equiv} \ldots \texttt{special}; the \texttt{crypto} command therefore supports special arguments \texttt{collisions\_LHS} and, for \texttt{rom\_partial}(\texttt{hash}), collision matrix. See the explanation of the \texttt{collisions\_LHS} argument, the collision matrix, and the oracles present in these equivalences in the documentation of \texttt{equiv} \ldots \texttt{special}.

- \texttt{expand ROM\_hash\_large}([\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{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 \texttt{ROM\_hash} above.

- \texttt{expand CollisionResistant\_hash}([\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hashoracle}, \texttt{Phash}]). defines a collision-resistant hash function [13], [9, Section 8.2]. The arguments already present in \texttt{PublicKey\_hash} have the same meaning.

- \texttt{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 \( hh \).)

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

- \texttt{expand HiddenKeyCollisionResistant\_hash}([\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hashoracle}, qH, \texttt{Phash}]). defines a hidden-key collision-resistant hash function [9, 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 arguments already present in \texttt{HiddenKey\_hash} have the same meaning.

- \texttt{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 \texttt{collision\_res}(\texttt{hash}) for use in the \texttt{crypto} command (see Section 7).

- \texttt{expand FixedCollisionResistant\_hash}([\texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{Phash}]). defines a collision-resistant hash function, for a hash function without key. (WARNING: This definition makes sense only for a fixed adversary. See [12].)

- \texttt{hashinput} is the type of the input of the hash function.

- \texttt{hashoutput} is the type of the output of the hash function.

- \texttt{Phash} is the probability of breaking collision resistance, for the considered adversary.

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

- \texttt{expand SecondPreimageResistant\_hash}([\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hashoracle}, Phash]). defines a second-preimage-resistant hash function [13]. The interface is the same as for collision-resistant hash functions above. However, note that the argument type \texttt{hashinput} must be \texttt{ bounded} or \texttt{ nonuniform} so that one can generate random values in it. It is typically \texttt{ fixed} and \texttt{ large}.

- \texttt{expand HiddenKeySecondPreimageResistant\_hash}([\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hashoracle}, qH, \texttt{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 \texttt{hashinput} must be \texttt{ bounded} or \texttt{ nonuniform} so that one can generate random values in it. It is typically \texttt{ fixed} and \texttt{ large}.

- This macro defines the equivalence named \texttt{second\_pre\_res}(\texttt{hash}) for use in the \texttt{crypto} command (see Section 7).
\begin{itemize}
  \item \texttt{expand FixedSecondPreimageResistant_hash}(\texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{Phash})\texttt{.} 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 \cite{13}, which we omit in our model.) \texttt{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 \texttt{fixed} and \texttt{large}.

  \texttt{hash(hashinput)}: \texttt{hashoutput} is the type of the output of the hash function.

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

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

  \item \texttt{expand PreimageResistant_hash}(\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hashoracle}, \texttt{Phash})\texttt{.} defines a preimage-resistant hash function \cite{13}. The interface is the same as for collision-resistant hash functions above. However, note that the argument type \texttt{hashinput} must be bounded or nonuniform so that one can generate random values in it. It is typically \texttt{fixed} and \texttt{large}.

  This macro defines the equivalence named \texttt{preimage_res(hash)} for use in the \texttt{crypto} command (see Section \ref{section}).

  \texttt{expand PreimageResistant_hash_all_args(\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hash'}, \texttt{hashoracle}, \texttt{Phash})}. is similar, with an additional argument \texttt{hash'}, which is a symbol that replaces \texttt{hash} after game transformation.

  \item \texttt{expand HiddenKeyPreimageResistant_hash}(\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hashoracle}, \texttt{qH}, \texttt{Phash})\texttt{.} 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 \texttt{hashinput} must be bounded or nonuniform so that one can generate random values in it. It is typically \texttt{fixed} and \texttt{large}.

  This macro defines the equivalence named \texttt{preimage_res(hash)} for use in the \texttt{crypto} command (see Section \ref{section}).

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

  \item \texttt{expand FixedPreimageResistant_hash}(\texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{Phash})\texttt{.} 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 \cite{13}, 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 \texttt{preimage_res(hash)} for use in the \texttt{crypto} command (see Section \ref{section}).

  \texttt{expand FixedPreimageResistant_hash_all_args(\texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hash'}, \texttt{Phash})}. is similar, with an additional argument \texttt{hash'}, which is a symbol that replaces \texttt{hash} after game transformation.

  \item \texttt{expand UniversalOneWay_hash}(\texttt{key}, \texttt{hashinput}, \texttt{hashoutput}, \texttt{hash}, \texttt{hashoracle}, \texttt{Phash})\texttt{.} defines a universal one-way hash function \cite{11}. The interface is the same as for collision-resistant hash functions above.

  \item Similarly to the macros above, for \texttt{N} from 1 to 10, the macros

  \texttt{expand HiddenKey_hash_N(\texttt{key}, \texttt{hashinput1}, \ldots, \texttt{hashinputN}, \texttt{hashoutput}, \texttt{hash}, \texttt{hashoracle}, \texttt{qH})}. \texttt{expand PublicKey_hash_N(\texttt{key}, \texttt{hashinput1}, \ldots, \texttt{hashinputN}, \texttt{hashoutput}, \texttt{hashoracle})}. \texttt{expand ROM_hash_N(\texttt{key}, \texttt{hashinput1}, \ldots, \texttt{hashinputN}, \texttt{hashoutput}, \texttt{hashoracle}, \texttt{qH})}. \texttt{expand ROM_hash_large_N(\texttt{key}, \texttt{hashinput1}, \ldots, \texttt{hashinputN}, \texttt{hashoutput}, \texttt{hashoracle}, \texttt{qH})}. 

\end{itemize}
\textbf{expand CollisionResistant_hash}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hashoracle, Phash).
\textbf{expand HiddenKeyCollisionResistant_hash}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hashoracle, qH, Phash).
\textbf{expand FixedCollisionResistant_hash}_N(hashinput_1, ..., hashinput_N, hashoutput, hash, hashoracle, Phash).
\textbf{expand SecondPreimageResistant_hash}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hashoracle, Phash).
\textbf{expand HiddenKeySecondPreimageResistant_hash}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hashoracle, qH, Phash).
\textbf{expand FixedSecondPreimageResistant_hash}_N(hashinput_1, ..., hashinput_N, hashoutput, hash, Phash).
\textbf{expand PreimageResistant_hash}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hashoracle, Phash).
\textbf{expand PreimageResistant_hash_all_args}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hash', hashoracle, Phash).
\textbf{expand HiddenKeyPreimageResistant_hash}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hashoracle, qH, Phash).
\textbf{expand HiddenKeyPreimageResistant_hash_all_args}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hash', hashoracle, qH, Phash).
\textbf{expand FixedPreimageResistant_hash}_N(hashinput_1, ..., hashinput_N, hashoutput, hash, Phash).
\textbf{expand FixedPreimageResistant_hash_all_args}_N(hashinput_1, ..., hashinput_N, hashoutput, hash, hash', Phash).
\textbf{expand UniversalOneWay_hash}_N(key, hashinput_1, ..., hashinput_N, hashoutput, hash, hashoracle, Phash).

define hash functions with \textit{N} arguments, with the same properties as above.

\textit{hashinput}_1, ..., \textit{hashinput}_\textit{N} are the types of the inputs of the hash function and 
\textit{hash}(\textit{hashinput}_1, ..., \textit{hashinput}_\textit{N}) : \textit{hashoutput} is the hash function.

- \textbf{expand PRF(key, input, output, f, Pprf)}. defines a pseudo-random function.

\textit{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 \textit{f} without mentioned the length of the key), typically \textit{fixed} and \textit{large}.

\textit{input} is the type of the input of the PRF.

\textit{output} is the type of the output of the PRF, must be \textit{bounded}, typically \textit{fixed}.

\textit{f(key, input) : output} is the PRF function.

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

The types \textit{key, input, output} and the probability \textit{Pprf} must be declared before this macro is 
expanded. The function \textit{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 \textit{prf(f)} and \textit{prf\_partial(f)} for use in the \texttt{crypto} command 
(see Section 4). These equivalences are generated via \texttt{equiv...special}; the \texttt{crypto} command 
therefore supports special arguments \texttt{collisions\_LHS} and, for \textit{prf\_partial(f)}, collision matrix.
See the explanation of the \texttt{collisions\_LHS} argument, the collision matrix, and the oracles present 
in these equivalences in the documentation of \texttt{equiv...special}.

- \textbf{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 \textbf{PRF} above.
• Similarly, for $N$ from 1 to 10, the macros
  \begin{align*}
    &\text{expand PRF}_N(key, \text{input}1, \ldots, \text{input}N, \text{output}, f, Pprf), \\
    &\text{expand PRF}\_\text{large}_N(key, \text{input}1, \ldots, \text{input}N, \text{output}, f, Pprf),
  \end{align*}
define pseudo-random functions with $N$ arguments, similarly to PRF and PRF\_large above. \text{input}1, \ldots, \text{input}N are the types of the inputs of the PRF and $f(key, \text{input}1, \ldots, \text{input}N) : \text{output}$ is the PRF.

6.13 Trapdoor permutations

• expand trapdoor\_perm(seed, pkey, skey, D, pkgen, skgen, f, invf) defines a trapdoor permutation, without any security assumption.
  
  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.
  
  $\text{pkgen}(\text{seed}) : \text{pkey}$ is the public key generation function.
  
  $\text{skgen}(\text{seed}) : \text{skey}$ is the secret key generation function.
  
  $f(pkey, D) : D$ is the permutation (taking as argument the public key)
  
  $\text{invf}(\text{skey}, D) : D$ is the inverse permutation of $f$ (taking as argument the secret key, i.e. the trapdoor)
  
  The types seed, pkey, skey, and $D$ 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.

• expand OW\_trapdoor\_perm(seed, pkey, skey, D, pkgen, skgen, f, invf, POW) defines a one-way trapdoor permutation.
  
  The arguments already present in trapdoor\_perm have the same meaning.
  
  $\text{POW}(t)$ 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 $\text{POW}$ must be declared before this macro. The functions $\text{pkgen, skgen, f, invf}$ 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 $\text{remove} \_\text{invf}(f)$, which expresses that, for $y$ chosen randomly in $D$, $y$ and $\text{invf}(\text{skey}, y)$ are distributed like for $x$ chosen randomly in $D$, $f(\text{pkey}, x)$ and $x$, and $\text{or}(f)$, 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 $\text{remove} \_\text{invf}(f)$ as above and $\text{or} \_\text{rsr}(f)$.

• expand PD\_trapdoor\_perm(seed, pkey, skey, D, Dow, Dr, pkgen, skgen, f, invf, concat) defines a partial-domain trapdoor permutation, without any security assumption.
  
  The arguments already present in trapdoor\_perm have the same meaning.
  
  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 $f(pk, \text{concat}(x, y))$ where $y$ is a random element of Dr). Dow and Dr must be bounded, typically fixed.
  
  $\text{concat}(\text{Dow}, \text{Dr}) : D$ is bitstring concatenation.
The types seed, pkey, skkey, D, Dow, and Dr 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.

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

  The arguments already present in PD_trapdoor_perm have the same meaning.

  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 trials.

  The types seed, pkey, skkey, 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_inv(f), which expresses that, for y chosen randomly in D, y and invf(skkey,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, skkey, D, pkgen, pkgen', skgen, f, f', invf, POW).
- expand OW_trapdoor_perm_RSR_all_args(seed, pkey, skkey, D, pkgen, pkgen', skgen, f, f', invf, POW).
- expand set_PD_OW_trapdoor_perm_all_args(seed, pkey, skkey, 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.

6.14 Diffie-Hellman key agreements

The specification of Diffie-Hellman key agreements is typically composed of two or three macro expansions:

- 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 exp, 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 \( \text{is neutral}(X) \) is true if and only if \( X \) is the neutral element. Prime-order groups without the neutral element also satisfy this assumption, with \( \text{is neutral}(X) = \text{false} \).

- \( \text{expand DH subgroup}(G, Z, g, \text{exp}, \text{mult}, \text{subG}, g_k, \text{exp div k}, \text{exp div k}', \text{pow k}, \text{subGtoG}) \), 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. \( \text{subG} = \{X^k \mid X \in G \} \) is a subset of \( G \);
3. \( X, X' \) in \( \text{subG} \), for any \( x \) prime to \( n \), \( X^x = X'^x \Rightarrow X = X' \);
4. exponentiation yields the same results for exponents equal modulo \( kn \).

\( \text{exp}(G, Z) : G \) is defined by \( \text{exp}(X, y) = X^y \).

\( \text{mult}(Z, Z) : Z \) is the product of integers (possibly modulo \( kn \)), commutative.

\( \text{subG} = \{X^k \mid X \in G \} \) is a subset of \( G \) as mentioned above. The type \( \text{subG} \) must be bounded and large.

\( g_k = g^k \in \text{subG} \).

\( \text{exp div k}(\text{subG}, Z) : \text{subG} \) is defined by \( \text{exp div k}(X, y) = X^y/k \).

\( \text{exp div k}' \) is defined like \( \text{exp div k} \); it replaces \( \text{exp div k} \) after games transformations.

\( \text{pow k}(G) : \text{subG} \) is defined by \( \text{pow k}(X) = X^k \).

\( \text{subGtoG}(\text{subG}) : G \) is the injection from \( \text{subG} \) to \( G \).

The types \( G, Z, \) and \( \text{subG} \) must be declared before expanding this macro. The constants \( g \) and \( g_k \), and the functions \( \text{exp}, \text{mult}, \text{exp div k}, \text{exp div k}', \text{pow k}, \text{subGtoG} \) 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 \( \text{DH exclude weak keys} \) should be applied to the subgroup, that is, \( \text{expand assumption}(\text{subG}, Z, g_k, \text{exp div k}, \text{exp div k}', \text{mult}, \ldots) \).

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](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 \( k'p' \) (\( k' = 4, p \) and \( p' \) are large primes). This can be done as follows using \( \text{DH exclude weak keys} \) defined below:

\[
\text{expand DH subgroup}(G, Znw, g, \text{expnw}, \text{mult}, \text{subG}, g_k, \text{exp div k}, \text{exp div k}', \text{pow k}, \text{subGtoG}),
\]

\[
\text{letproba Pweak key} = 2^n - 445.
\]

\[
\text{expand DH exclude weak keys}(G, Z, Znw, ZnutoZ, \text{exp}, \text{expnw}, \text{Pweak key}).
\]

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

- \( \text{expand DH subgroup with is neutral}(G, Z, g, \text{exp}, \text{mult}, \text{subG}, g_k, \text{exp div k}, \text{exp div k}', \text{pow k}, \text{subGtoG}, \text{is neutral} G, \text{is neutral} \text{subG}) \), defines a Diffie-Hellman structure like \( \text{expand DH subgroup}(G, Z, g, \text{exp}, \text{mult}, \text{subG}, g_k, \text{exp div k}, \text{exp div k}', \text{pow k}, \text{subGtoG}) \), with additionally \( \text{is neutral}(g^k) \) is false and for \( X \) in \( \text{subG} \), \( \text{is neutral}(X^y) \) if and only if \( \text{is neutral}(X) \).
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.

Curve25519 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:

\[
\text{expand DH_subgroup_with_is_neutral}(G, Z nw, g, exp nw, mult, subG, g_k, \\
\quad \text{exp\_div\_k, exp\_div\_k', pow\_k, subGtoG, is\_neutral\_G, is\_neutral\_subG}).
\]

let proba Pweak\_key = 2^−445.

\[
\text{expand DH_exclude_weak_keys}(G, Z, Z nw, \text{ZnwtoZ}, \text{exp, exp nw, Pweak\_key}).
\]

Prime order groups (without the neutral element) satisfy these properties with is_neutral(X) = false.

- expand DH_good_group(G, Z, g, exp, \text{exp}', \text{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, \ldots, q-1\}, g is a generator of G, mult is the product modulo q in \{1, \ldots, 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):

* \text{group\_to\_exp\_strict} \text{(exp)} which allows to replace a random \( X \in G \) with \( \text{exp}(g, x) \) for a random \( x \in Z \), provided \( \text{exp}(X, _) \) occurs in the game.

* \text{group\_to\_exp} \text{(exp)} which allows to replace a random \( X \in G \) with \( \text{exp}(g, x) \) for a random \( x \in Z \) in any case. (This transformation is applied only manually.)

* \text{exp\_to\_group} \text{(exp)} which allows to replace \( \text{exp}(g, x) \) for a random \( x \in Z \) with a random \( X \in G \).

* \text{exp\_t'o\_group} \text{(exp)} which allows to replace \( \text{exp}'(g, x) \) for a random \( x \in Z \) with a random \( X \in G \).

- expand DH_single_coord_ladder(G, Z, g, exp, \text{mult}, subG, Znw, \text{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 \( E(\mathbb{F}_p) \) 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 \( q = X_0(g_0) \) 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 \( \text{red} \circ \text{repr} \) is the identity. The Diffie-Hellman “exponentiation” is defined by

\[
\text{exp}(X, y) = \text{repr}(y \cdot \text{red}(X))
\]

The secret keys are chosen uniformly in \( \{kn\ |\ n \in [n_{min}, n_{max}]\} \) 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 \text{exclude_weak_keys}(Z) defined by this macro, automatically or with the crypto command (see Section 7).
This model is justified in [10].

\( G \): 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 \( \{ x \cdot y \mid x \in \{1, \ldots, q-1\} \} \). 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 \( \phi(q') \) that is, exponents without weak keys. Random choices in \( Znw \) are done by choosing uniformly in \( \{ kn \mid n \in [n_{min}, n_{max}], n \text{ prime to } \phi(q') \} \). \( Z \) is the set of integers multiple of \( k \) modulo \( \phi(q') \), 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 \( \text{Pcollrand}(Z) = 1/(n_{max} - n_{min} + 1) \).

\( Znw(Znw) : Z \): injection from \( Znw \) to \( Z \).

\( g : G \): represents the base point.

\( \text{exp}(G, Z) : G \): the exponentiation function.

\( \text{mult}(Znw, Znw) : Znw \): \( Znw \): the multiplication function for exponents, defined as \( \text{mult}(x, y) = x \cdot y \mod \phi(q') \). (It remains in \( Znw \.)

\( \text{exp}(x, y) = y \cdot k \mod \phi(q') \) is an element of \( subG \).

\( \text{exp} \div k (subG, Znw) : subG \): defined by \( \text{exp} \div k(X, Y) = (y/k) \cdot X \).

\( \text{exp} \div k \): symbol that replaces \( \text{exp} \div k \) after game transformation, with the same definition as \( \text{exp} \div k \).

\( \text{pow}_k(G) : subG \): defined by \( \text{pow}_k(x) = k \cdot \text{red}(x) \).

\( \text{GtoG}(subG) : G \): is \text{rep} restricted to \( subG \).

\( \text{is_zero}_G (G) : \text{bool} \): defined by: \( \text{is_zero}_G(X) \) is true when \( X \) is the public key 0.

\( \text{is_zero}_{subG}(G) : \text{bool} \): defined by: \( \text{is_zero}_{subG}(X) \) is true when \( X \) is the public key 0.

The types \( G, subG, Z, \) and \( Znw \) must be declared before this macro. The functions \( g, \text{exp}, \text{mult}, \text{ZnwtoZ}, g \cdot k, \text{exp} \div k, \text{exp} \div k', \text{pow}_k, \text{subGtoG}, \text{is_zero}_G, \text{is_zero}_{subG} \) 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, \( \text{expand assumption}(subG, Znw, g \cdot k, \text{exp} \div k, \text{exp} \div k', \text{mult}, \ldots) \).

- \( \text{expand DH} \_X25519(G, Z, g, \text{exp}, \text{mult}, subG, g \cdot k, \text{exp} \div k, \text{exp} \div k', \text{pow}_k, \text{subGtoG}, \text{is_zero}_G, \text{is_zero}_{subG}) \). models Curve25519 as defined in RFC 7748 (https://tools.ietf.org/html/rfc7748). It is justified in detail in [10]. More generally, it supports the same curves as \( \text{DH}_\text{single_coord_ladder} \) with the additional assumption that all secret keys are prime to \( q' \). 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 \( 0 < \delta < 2^{128}, q' = 2^{253} - 9 - 2\delta, \text{red}(X) = (X \mod 2^{255}) \mod \delta, \text{repr}(X) = (X) \text{is the representation of } X \) as an element of \( \{0, \ldots, p-1\} \), \( n_{min} = 2^{251} \), \( n_{max} = 2^{252} - 1 \), \( \text{Pcollrand}(Z) = 2^{-251} \). (For simple examples that use Curve25519, using the macro \( \text{DH}_\text{proba_collision} \), possibly with \( \text{DH}_\text{subgroup} \) or \( \text{DH}_\text{subgroup_with_is_neutral, may also work.} \)

- \( \text{expand DH} \_X448(G, Z, g, \text{exp}, \text{mult}, subG, Znw, ZnwtoZ, g \cdot k, \text{exp} \div k, \text{exp} \div k', \text{pow}_k, \text{subGtoG}, \text{is_zero}_G, \text{is_zero}_{subG}) \). models Curve448 as defined in RFC 7748 (https://tools.ietf.org/html/rfc7748). More generally, it supports the same curves as \( \text{DH}_\text{single_coord_ladder} \) with the additional assumptions that there is at least 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 [10].

- Optionally, one or more of the following macros:
- expand DH\_exclude\_weak\_keys(G, Z, Znw, ZnwoZ, 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.

ZnwoZ(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 ZnwoZ 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 \{kp\}. Pweak\_key = 2^{-445}. It is also useful for groups of prime order q in case private keys are chosen in {0,...,q-1}; one should eliminate the weak private key 0, so Z = {0,...,q-1}, Znw = {1,...,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 ≥ 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 ∈ G with exp(g, x) for a random x ∈ Z, provided exp(X, _) occurs in the game.
* group\_to\_exp(exp) which allows to replace a random X ∈ G with exp(g, x) for a random x ∈ 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 ∈ Z with a random X ∈ G.
* exp\_to\_group(exp) which allows to replace exp'(g, x) for a random x ∈ Z with a random X ∈ 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, ZnutoZ, 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 (CDH). The CDH assumption means that the adversary has negligible probability of computing exp(g, mult(a, b)) when it knows exp(g, a) and exp(g, b) for
random $a, b \in \mathbb{Z}$; $p(t)$ is the probability of breaking the CDH assumption, for one pair of exponents, in time $t$. This macro defines the equivalence $\text{cdh}(\text{exp})$, which corresponds to the CDH property, for use in the $\text{crypto}$ command (see Section 7).

- **expand $\text{CDH\_RSR}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, p_d)$**, 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 $\text{CDH\_good\_group}$). $p_d = 2^{-126}$ for Curve25519, and $p_d = 2^{-221}$ for Curve448.

- **expand $\text{DDH}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p)$**, says that the group $G$ satisfies the decisional Diffie-Hellman assumption (DDH). The DDH assumption means that the adversary has negligible probability of distinguishing a random element of $G$ from $\text{exp}(g, \text{mult}(a, b))$ when it knows $\text{exp}(g, a)$ and $\text{exp}(g, b)$ for random $a, b \in \mathbb{Z}$; $p(t)$ is the probability of breaking the DDH assumption, for one pair of exponents, in time $t$. The default distribution on $G$ is typically the distribution of $\text{exp}(g, c)$ for random $c \in \mathbb{Z}$, following the default distribution on $Z$; the probability $p$ may be adjusted in case a slightly different default distribution is used. This macro defines the equivalence $\text{ddh}(\text{exp})$, which corresponds to the DDH property, for use in the $\text{crypto}$ command (see Section 7).

- **expand $\text{DDH\_RSR}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, p_d)$**, 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 $\text{DDH\_RSR}$). 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 point for Curve25519/Curve448. CDH and GDH with random self reducibility do not have such limitations.

- **expand $\text{GDH}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, p_d)$**, says that the group $G$ satisfies the gap Diffie-Hellman assumption (GDH). In CryptoVerif, the GDH assumption means that, when $\text{exp}(g, a)$ and $\text{exp}(g, b)$ are known to the adversary for random $a, b \in \mathbb{Z}$, the adversary can compute $\text{exp}(g, \text{mult}(a, b))$ only with negligible probability, even in the presence of a decisional Diffie-Hellman (DDH) oracle $\text{GDH}(\Gamma, X, Y, Z)$ that tells

  * for $\Gamma = \ast$, whether $X = \text{exp}(g, x)$ and $Z = \text{exp}(Y, x)$ for some $x$;
  * for $\Gamma \neq \ast$, whether $\Gamma = \text{exp}(g, t)$, $X = \text{exp}(g, x)$, and $\text{exp}(Z, t) = \text{exp}(Y, x)$ for some $x, t \in \mathbb{Z}$.

(This rather non-standard definition is useful for Curve25519/Curve448. When we work in a prime order group, this is equivalent to $X = \text{exp}(\Gamma, x)$, $Y = \text{exp}(\Gamma, y)$, and $Z = \text{exp}(\Gamma, xy)$ for some $x, y \in \mathbb{Z}$, that is, $X, Y, Z$ is a correct Diffie-Hellman triple with generator $\Gamma$, and the case $\Gamma = \ast$ is equivalent to $\Gamma = g$.) 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 $\text{GDH\_RSR}$). 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 $\text{gdh}(\text{exp})$, which corresponds to the GDH property, for use in the $\text{crypto}$ command (see Section 7).

- **expand $\text{GDH\_RSR}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, p_d)$**, is similar to GDH, but uses random self reducibility.
- \texttt{expand square\_CDH}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, \text{sqp}), says that the group \( G \) satisfies the computational Diffie-Hellman assumption and the square computational Diffie-Hellman assumption. The square CDH assumption means that the adversary has negligible probability of computing \( \text{exp}(g, \text{mult}(a, a)) \) when it knows \( \text{exp}(g, a) \) for random \( a \in Z; \). \( p(t) \) is the probability of breaking the CDH assumption, for one pair of exponents, in time \( t \) and \( \text{sqp}(t) \) is the probability of breaking the square CDH assumption, for one pair of exponents, in time \( t \). This macro defines the equivalence \text{cdh}(\text{exp}), which corresponds to the (square) CDH property, for use in the \texttt{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 \( \text{exp}(g, \text{mult}(x, x)) \).

- \texttt{expand square\_CDH\_RSR}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, \text{sqp}). says that the group \( G \) satisfies the square computational Diffie-Hellman assumption; \( \text{sqp}(t) \) 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 \text{CDH\_RSR}). This macro defines the equivalence \text{cdh}\text{(exp)}, which corresponds to the square CDH property, for use in the \texttt{crypto} command (see Section 7).

- \texttt{expand square\_DDH}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, \text{sqp}). says that the group \( G \) satisfies the decisional Diffie-Hellman assumption and the square decisional Diffie-Hellman assumption. The DDH assumption means that the adversary has negligible probability of distinguishing a random element of \( G \) from \( \text{exp}(g, \text{mult}(a, a)) \) when it knows \( \text{exp}(g, a) \) for random \( a \in Z; \). \( p(t) \) is the probability of breaking the DDH assumption, for one pair of exponents, in time \( t \) and \( \text{sqp}(t) \) is the probability of breaking the square DDH assumption, for one pair of exponents, in time \( t \). This macro defines the equivalence \text{ddh}(\text{exp}), which corresponds to the (square) DDH property, for use in the \texttt{crypto} command (see Section 7).

- \texttt{expand square\_GDH}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, \text{sqp}, p_d). says that the group \( G \) satisfies the GDH assumption and the square GDH assumption. In CryptoVerif, the square GDH assumption means that, when \( \text{exp}(g, a) \) is known to the adversary for random \( a \in Z \), the adversary can compute \( \text{exp}(g, \text{mult}(a, a)) \) only with negligible probability, even in the presence of the same DDH oracle as for the GDH 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 DDH oracle and \( \text{sqp}(t, n) \) is the probability of breaking the square GDH assumption, for one pair of exponents, in time \( t \) with at most \( n \) calls to the DDH oracle. \( p_d \) is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key (see \text{CDH\_RSR}). This macro defines the equivalence \text{gdh}(\text{exp}), which corresponds to the (square) GDH property, for use in the \texttt{crypto} command (see Section 7).

- \texttt{expand square\_GDH\_RSR}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, p_d). says that the group \( G \) satisfies the square GDH assumption; \( \text{sqp}(t, n) \) 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 \text{CDH\_RSR}). This macro defines the equivalence \text{gdh}(\text{exp}), which corresponds to the square GDH property, for use in the \texttt{crypto} command (see Section 7).

- \texttt{expand fixed\_gen\_GDH}(G, Z, g, \text{exp}, \text{exp}', \text{mult}, p, p_d), says that the group \( G \) satisfies the (fixed-generator) gap Diffie-Hellman assumption (GDH). This assumption means that, when \( \text{exp}(g, a) \) and \( \text{exp}(g, b) \) are known to the adversary for random \( a, b \in Z \), the adversary can compute \( \text{exp}(g, \text{mult}(a, b)) \) only with negligible probability, even in the presence of a decisional Diffie-Hellman (DDH) oracle \( \text{DDH}(X, Y, Z) \) that tells whether \( X = \text{exp}(g, x) \) and \( Z = \text{exp}(Y, x) \) for some \( x \). The probability \( p(t, n) \) is the probability of breaking this 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 \text{CDH\_RSR}). This macro defines the equivalence \text{gdh}(\text{exp}) for use in the \texttt{crypto} command (see Section 7).
- expand \texttt{fixed\_gen\_GDH\_RSR}(G, Z, g, exp, exp', mult, p, p_q), is similar to \texttt{fixed\_gen\_GDH}, but uses random self reducibility.

- expand \texttt{fixed\_gen\_square\_GDH}(G, Z, g, exp, exp', mult, p, sqp, p_q), says that the group \( G \) satisfies the fixed-generator GDH assumption and the square fixed-generator GDH assumption. The square fixed-generator GDH assumption means that, when \( \exp(g, a) \) is known to the adversary for random \( a \in Z \), the adversary can compute \( \exp(g, \mult(a, a)) \) only with negligible probability, even in the presence of the same DDH oracle as for the fixed-generator GDH assumption; \( p(t, n) \) is the probability of breaking the fixed-generator GDH assumption, for one pair of exponents, in time \( t \) with at most \( n \) calls to the DDH oracle and \( sqp(t, n) \) is the probability of breaking the square fixed-generator GDH assumption, for one pair of exponents, in time \( t \) with at most \( n \) calls to the DDH oracle. \( p_d \) is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key (see \texttt{CDH\_RSR}). This macro defines the equivalence \( \text{gdh}(\exp) \), which corresponds to the fixed-generator (square) GDH property, for use in the crypto command (see Section 7).

- expand \texttt{Fixed\_Gen\_Square\_GDH\_RSR}(G, Z, g, exp, exp', mult, sqp, p_q), says that the group \( G \) satisfies the square fixed-generator GDH assumption; \( sqp(t, n) \) is the probability of breaking the square fixed-generator GDH assumption, for one pair of exponents, in time \( t \) with at most \( n \) calls to the DDH 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 \texttt{CDH\_RSR}). This macro defines the equivalence \( \text{gdh}(\exp) \), which corresponds to the square fixed-generator GDH property, for use in the crypto command (see Section 7).

- expand \texttt{StDH}(G, Z, g, exp, exp', mult, p, p_q), says that the group \( G \) satisfies the strong Diffie-Hellman assumption (\texttt{StDH}). This assumption means that, when \( \exp(g, a) \) and \( \exp(g, b) \) are known to the adversary for random \( a, b \in Z \), the adversary can compute \( \exp(g, \mult(a, b)) \) only with negligible probability, even in the presence of a DDH oracle \( DDH(Y, Z) \) that tells whether \( Z = \exp(Y, a) \). The probability \( p(t, n) \) is the probability of breaking this assumption for one pair of exponents in time \( t \) with at most \( n \) calls to the DDH oracle. \( p_d \) is the probability of distinguishing a key that comes from rerandomization from an honestly chosen key (see \texttt{CDH\_RSR}). This macro defines the equivalence \( \text{stdh}(\exp) \), which corresponds to the square fixed-generator GDH property, for use in the crypto command (see Section 7).

- expand \texttt{bPRF\_ODH}(G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p, PCollKey1), where \texttt{bprf} \( \in \{\texttt{mm}, \texttt{ss}, \texttt{mm}, \texttt{ss}, \texttt{mm}, \texttt{mm} \} \), says that the group \( G \) satisfies the \texttt{bPRF-ODH} (pseudo-random function oracle Diffie-Hellman) assumption \cite{8}. 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 \( O_a(Y, x) = \text{prf}(\exp(Y, a), x) \) and \( O_b(X, x) = \text{prf}(\exp(X, b), x) \) cannot distinguish \( x \mapsto \text{prf}(\exp(g, \mult(a, b)), x) \) from a random function. The character \( l \) determines the number of allowed calls to \( O_x \): \( n \) means 0 (no call), \( m \) means 1 (single call), \( m \) means any number (multiple calls); similar \( r \) determines the number of allowed calls to \( O_y \). (This definition differs from the one of \cite{8} in that the oracles can be called in any order and there can be several calls to \( x \mapsto \text{prf}(\exp(g, \mult(a, b)), x) \). When multiple calls to \( O_a \) or \( O_b \) are allowed, they can be used together with a hybrid argument to simulate multiple calls to \( x \mapsto \text{prf}(\exp(g, \mult(a, b)), x) \).) The probability \( p(t, n, n') \) is the probability of breaking the \texttt{PRF-ODH} assumption in time \( t \) with \( n \) queries to \( \text{prf}(\exp(g, \mult(a, b)), x) \) and \( n' \) queries to \( O_a \) and \( O_b \) in total. The argument \( n' \) is absent when \( l = r = 0 \). The pseudo-random function \( \text{prf}(G, \text{prf\_in}) : \text{prf\_out} \) takes as argument a group element in \( G \) and an element in \( \text{prf\_in} \), and produces a result in \( \text{prf\_out} \). The type \( \text{prf\_out} \) must be bounded or nonuniform. This macro defines the equivalence \( \text{prf\_odh}(\text{prf}) \), which corresponds to the \texttt{bPRF-ODH} property, for use in the crypto command (see Section 7). When this assumption is used with \texttt{DH\_subgroup}, \texttt{DH\_subgroup\_with\_is\_neutral}, \texttt{DH\_X25519}, \texttt{DH\_X448}, or \texttt{DH\_single\_coord\_ladder}, it must be applied to the subgroup, which
can be done as follows:

\begin{align*}
\text{expand DH\_single\_coord\_ladder}(G, Z, g, exp, mult, subG, Zn\_zuO, g_k, \\
\quad \exp\_div\_k, \exp\_div\_k', \pow\_k, subGtoG, zero, sub\_zero). \\
\text{expand mmPRF\_ODH}(subG, Zn, prf\_in, prf\_out, g_k, \exp\_div\_k, \exp\_div\_k', mult, \\
\quad prf\_subG(p). \\
\text{fun prf}(G, prf\_in) : prf\_out. \\
equation \forall x : subG, y : prf\_in; prf(subGtoG(x), y) = prf\_subG(x, y).
\end{align*}

If \( G \) is Curve448 and \( lr \neq nn \), the weak private key \( kp \) must be excluded, which can be done using \text{DH\_exclude\_weak\_keys}, \text{DH\_X448}, or \text{DH\_single\_coord\_ladder}.

Additionally, when \( lr \neq nn \), this assumption requires that it is possible to test efficiently whether \( \exp(Y, a) = \exp(Z, a) \) knowing just \( Y \) and \( Z \) (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 \( Z \) are equivalent. The argument \text{PCollKey1} is present only when \( l \) or \( r \) is \( m \). It bounds the probability that two honestly generated random keys are equivalent.

- \text{expand square\_PRF\_ODH}(G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p, sqp). says that the group \( G \) satisfies the square IPRF-ODH assumption and the \( l\)PRF-ODH assumption. The square IPRF-ODH assumption means that an adversary that has a public Diffie-Hellman key \( \exp(g, a) \) for random \( a \) and has access to the oracle \( O_a(Y, x) = prf(\exp(Y, a), x) \) cannot distinguish \( x \mapsto prf(\exp(g, mult(a, a)), x) \) from a random function. The character \( l \) determines the number of allowed calls to \( O_a \), as in PRF\_ODH. The probability \( sqp(t, n[l,n'] \) is the probability of breaking the square IPRF-ODH assumption in time \( t \) with \( n \) queries to \( prf(\exp(g, mult(a, a)), x) \), and when \( l = m \), \( n' \) queries to \( O_a \) in total. The probability \( p(t, n[l,n']) \) is the probability of breaking the \( l\)PRF-ODH assumption in time \( t \) with \( n \) queries to \( prf(\exp(g, mult(a, b)), x) \), and when \( l = n \), \( n' \) queries to \( O_a \) and \( O_b \) in total. The argument \( n' \) is absent when \( l \neq m \).

The types \text{prf\_in} and \text{prf\_out} and the pseudo-random function \text{prf} are defined as for PRF\_ODH. This macro defines the equivalence \text{prf\_odh(prf)}, which corresponds to the square IPRF-ODH and \( l\)PRF-ODH properties, for use in the crypto command (see Section \[7\]).

When this assumption is used with \text{DH\_subgroup}, \text{DH\_subgroup\_with\_is\_neutral}, \text{DH\_X25519}, \text{DH\_X448}, or \text{DH\_single\_coord\_ladder}, it must be applied to the subgroup, which can be done as for \text{PRF\_ODH}.

If \( G \) is Curve448 and \( l \neq n \), the weak private key \( kp \) must be excluded, which can be done using \text{DH\_exclude\_weak\_keys}, \text{DH\_X448}, or \text{DH\_single\_coord\_ladder}.

Additionally, when \( l \neq n \), this assumption requires that it is possible to test efficiently whether \( \exp(Y, a) = \exp(Z, a) \) knowing just \( Y \) and \( Z \) (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 \text{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.

- \text{expand CDH\_single}(G, Z, g, exp, exp', mult, p).
- \text{expand CDH\_RSR\_single}(G, Z, g, exp, exp', mult, p, p\_d).
- \text{expand DDH\_single}(G, Z, g, exp, exp', mult, p).
- \text{expand GDH\_single}(G, Z, g, exp, exp', mult, p).
- \text{expand GDH\_RSR\_single}(G, Z, g, exp, exp', mult, p, p\_d).
- \text{expand fixed\_gen\_GDH\_single}(G, Z, g, exp, exp', mult, p, p\_d).
- \text{expand fixed\_gen\_GDH\_RSR\_single}(G, Z, g, exp, exp', mult, p, p\_d).
- \text{expand mmPRF\_ODH\_single}(G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p).
- \text{expand mmPRF\_ODH\_single}(G, Z, prf\_in, prf\_out, g, exp, exp', mult, prf, p).
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.

6.15 Miscellaneous

- **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_CTX_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 $Pmac(t, N, N', Nu', l, l')$ 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_CTX_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 $Pencctxt(t, N, N', l', ld, ld')$ is the probability of breaking the INT-CTX 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_CTX_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 \( Penc ctxt(t, N, N', l', l', ld, ld') \) is the probability of breaking the INT-CTX 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 a 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 \( Pmac(t, N, N', l', l', ld, ld') \) 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, Pmac)** 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 \( Pmac(t, N, N', Nu', l', ld, ld') \) is the probability of breaking the INT-CTX 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_splitN(input_t, part1_t, ..., 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
  - \( part1_t, ..., partN_t \): types of the output parts
  - \( tuple_t \): type of a tuple of the output parts
  - \( tuple(part1_t, ..., partN_t) : tuple_t \): 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 \( tuple(x_1, ..., x_N) \) of \( N \) independent random values in \( part1_t, ..., partN_t \).

  To split a value \( y \) of type \( input_t \) into \( N \) parts of types \( part1_t, ..., partN_t \), write:

  \[
  \text{let } tuple(x_1, ..., x_N) = split(y) \text{ in } ...
  \]

  Note that a priori, CryptoVerif thinks that the pattern-matching with \( tuple(x_1, ..., x_N) \) may fail, and thus requires an else branch when the \text{let} occurs in a term. CryptoVerif realizes that the pattern-matching never fails when it expands the definition of \text{split}.

  This macro defines the equivalence named \text{splitter(split)} which replaces the splitting of a random number generation in \( input_t \) with \( N \) independent random number generations in \( part1_t, ..., partN_t \).

  \( input_t, part1_t, ..., partN_t, tuple_t \) 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. (Recall that to go inside the interactive mode, one can put inside the manually specified proof sequence `proof {c_1; . . . , c_n}` at any point the `interactiveMode` command, and otherwise set the `interactiveMode` to true.)

Some of the commands take a program point (or occurrence) in the current game as argument. One should use the command `show_game_occ` or `out_game_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]+}

As an example, the partial command `at_nth 1 2 "return{[0-9]+}\{([0-9]+)\}` looks for the first occurrence of a return inside the process, and gives back the occurrence point corresponding to the second occurrence point inside the `regexp`, which is the one after the parenthesis corresponding to the returned term.

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 a command, 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 commands:

- `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 find**: removes useless assignments, as above, as well as assignments let \( x = M \) in inside conditions of `find`.

• **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](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. Backslash (\) 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 \( x_i \gets M_i \), 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, \( M_i[M/i] \) is replaced with \( x_i[M] \), when \( x_i[M] \) is guaranteed to be defined, where \( i \) 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 random number generations and assignments. It supports two versions.
  - **move up \( x_1 \ldots x_n \) to occ** moves the random number generations or assignments of \( x_1, \ldots, x_n \) up in the syntax tree, to the program point `occ`. This program point must correspond to an oracle body. After the game transformation, a variable \( x \) (which may be among \( x_1, \ldots, x_n \)) is defined at program point `occ`, and all other variables \( x_i \) are defined by \( x_i \gets x \). All variables \( x_1, \ldots, x_n \) must have the same type. They must not be defined syntactically above the program point `occ`. Either all variables \( x_1, \ldots, x_n \) must be defined by random number generations or all of them must be defined by assignments.
    - If \( x_1, \ldots, x_n \) are defined by random number generations, this transformation performs eager sampling of \( x_i \). The random number generation of \( x_1, \ldots, x_n \) must be executed at most once for each execution of program point `occ`.
    - If \( x_1, \ldots, x_n \) are defined by assignments of terms \( M_i \), then all \( M_i \) must consist of variables, function applications, and tests; there must be one \( M_i \) defined at program point `occ` (which will be used as the definition of \( x \)); and all terms \( M_i \) must be equal.

For all other values of \( m \), **move \( m \)** moves random number generations and assignments down in the syntax tree:
    - It moves random number generations down in the syntax tree as much as possible, in order to delay the choice of random numbers (lazy sampling). 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 `Rename` instruction.
    - It moves 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.)

Only the random number generations and assignment at the process level can be moved. (Those that are terms will be left unchanged.) 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 $x$ "$exp_1$, ..., $exp_n$": 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 return 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.) This transformation provides a stronger form of lazy sampling than other variants of move.

The expressions "$exp$" allow the user to specify expressions that do not 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

$\forall \text{seq}\{\text{vartype}\}; \{\text{ident}\} <\Rightarrow \{\text{ident}\}; \{\text{simpleterm}\}$

The expression can be $\forall y_1 : T_1,\ldots,y_n : T_n; x' <\Rightarrow T$; $M$, where $T$ is the type of $x$ and $y_1,\ldots,y_n,x'$ are the variables of $M$. ($x' <\Rightarrow T$ can be replaced with new $x';T$.) 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 not 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 ($P\text{coll}l\text{rand}(T) \leq 2^{-n_0}$, that is, $T$ has option pcoll $n$ with $n \geq n'$ where $n'$ is set by set minAutoCollElim = pest $n'$; the default is $n' = 80$).

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 colI_elims(variables: $x_1,\ldots,x_n$; types: $t_1,\ldots,t_n$; terms: $occ_1,\ldots,occ_n$): simplify the game, additionally allowing elimination of collisions on data at all occurrences of variables $x_1,\ldots,x_n$, at all data of types $t_1,\ldots,t_n$, and at the program points $occ_1,\ldots,occ_n$. See above for how to specify the program points $occ$. 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 returned 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 $x$ colI_elims(variables: $x_1,\ldots,x_n$; types: $t_1,\ldots,t_n$; terms: $occ_1,\ldots,occ_n$) performs global dependency analysis on $x$, additionally allowing elimination of collisions on data at all occurrences of variables $x_1,\ldots,x_n$, at all data of types $t_1,\ldots,t_n$, and at the program points $occ_1,\ldots,occ_n$. See above for how to specify the program points $occ$. 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 \( x_1, \ldots, x_n \) or its type must be mentioned in \( t_1, \ldots, t_n \); mentioning the occurrences of \( x \) in \( occ_1, \ldots, occ_n \) 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.

- **\$rename \( x \):** 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 \$rename is executed for each of these variables in turn.

- **\$rename random:** Renames all variables defined by random number generations \(-R \) that have several definitions and no array accesses to distinct names.

- **all_simplify:** perform several simplifications on the game, as if
  - simplify,
  - move all if autoMove = true,
  - remove_assign useless if autoRemoveAssignFindCond = false, remove_assign find cond if autoRemoveAssignFindCond = true,
  - \$rename random if autoSARename = true,
  - and merge_branches if autoMergeBranches = true

  had been called.

- **expand:** expand if, let, \(-, \) find, event, event_abort, \(-R \) 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 \( \langle \text{crypto_args} \rangle \):** 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 \( \langle \text{info} \rangle \) as described below.
  - crypto \( \langle \text{name} \rangle \) \[special(seq{specialarg})] \( \langle \text{info} \rangle \): apply a cryptographic transformation determined by the name \( \langle \text{name} \rangle \), where:
    - \( \langle \text{name} \rangle \) 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 \( \langle \text{name} \rangle \) 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 \( \text{rom}, \text{prf}, \text{prp}, \text{sppr}, \) and \( \text{icm} \) there must be at most one special argument, and when one argument is present, it overrides the collisions_LHS argument of the special equivalence.
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. \( x_1 \ldots x_n \): apply the cryptographic transformation, where \( x_1, \ldots, x_n \) 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 \( x_1, \ldots, x_n \) if needed, except when a dot is added at the end of the list \( x_1, \ldots, x_n \). The transformation is applied only once. If several disjoint lists \( x_1, \ldots, x_n \) 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: \( x_1 \rightarrow y_1, \ldots, x_n \rightarrow y_n \); terms: \( \text{occ}_1 \rightarrow O_1, \ldots, \text{occ}_m \rightarrow O_m \)]`: apply the cryptographic transformation, where
   
   (a) \( x_1, \ldots, x_n \) are variable names of the game which correspond to random number generations \( y_1, \ldots, y_n \) respectively in the left-hand side of the equivalence. (CryptoVerif may automatically add variables to the list \( x_1 \rightarrow y_1, \ldots, x_n \rightarrow y_n \) 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) \( \text{occ}_1, \ldots, \text{occ}_m \) are program points at which terms will be transformed using oracles \( O_1, \ldots, O_m \) respectively of the equivalence. See above for how to specify the program points \( \text{occ}_i \). (CryptoVerif may automatically add elements to the list \( \text{occ}_1 \rightarrow O_1, \ldots, \text{occ}_m \rightarrow O_m \) 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 \textit{occ}. The program point \textit{occ} must correspond to an oracle body or to a term not in a condition of \textit{find}.

When the setting \texttt{autoExpand} is true and the occurrence \textit{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.

- \textbf{insert \textit{occ} "[ins]\text{" inserts instruction (ins) at program point \textit{occ}}. The instruction \texttt{(ins)} can be

\[
\text{(ident)} <- \text{R}\text{ (ident)}; \text{(ins)}
\]
\[
\text{new}\text{ (ident)};\text{(ident)}; \text{(ins)}
\]
\[
\text{event\_abort}\text{ (ident)}
\]
\[
\text{if}\text{ (cond) then (ins) [else (ins)]}
\]
\[
\text{let }\text{ (pattern) }=\text{(term) in (ins) [else (ins)]}
\]
\[
\text{let}\text{ (pattern) }<\text{ (term);}\text{(ins)}
\]
\[
\text{find}[\text{unique}]\text{ (findbranch) (orf\text{ (findbranch)})}\times\text{ [else (ins)]}
\]
\[
\text{(ins)}
\]

\text{where (findbranch) ::= seq\text{ (identbound) suchthat (cond) then (ins)}}

The empty instruction is replaced by the code that follows the insertion point. In all cases except \texttt{event\_abort}, 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 \texttt{event\_abort} in the inserted instruction. CryptoVerif then tries to bound this probability. This is similar to \texttt{insert\_event \textit{occ}}. However, the \texttt{insert\_event} command does not subsume \texttt{insert\_event}, because \texttt{insert} allows inserting only at process points while \texttt{insert\_event} allows inserting events at term points as well.

The instruction \texttt{(ins)} is equivalent to \texttt{ins}. The parentheses just help resolving ambiguities of the grammar, for instance to specify to which process \texttt{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 \texttt{set \text{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 \texttt{event}, \texttt{event\_abort}, \texttt{get}, \texttt{insert}, \texttt{new}, \texttt{<-R}, \texttt{if}, \texttt{find}, \texttt{let}, or \texttt{<-} are not expanded, so terms if, find, let and its synonym <- can occur only in conditions of find, event, event\_abort, new and its synonym <- must not occur as a term, get and insert must not occur. 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 \texttt{x} must be either \texttt{x} without array index syntactically under its definition, inside a defined condition of a find, or \texttt{x}[\texttt{M}_1,\ldots,\texttt{M}_n] under a defined condition that contains \texttt{x}[\texttt{M}_1,\ldots,\texttt{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 \textit{occ}. The program point \textit{occ} must correspond to an oracle body.

- \textbf{replace \textit{occ} "\textit{term}" replaces the term at program point \textit{occ} with the term \textit{term}}. Obviously, CryptoVerif must be able to prove that these two terms are equal. These terms must not contain \texttt{if}, \texttt{let}, \texttt{<-}, \texttt{find}, \texttt{<-R}, \texttt{new}, \texttt{event}, \texttt{event\_abort}, \texttt{insert}, \texttt{get}. See above for how to specify the program point \textit{occ}. The program point \textit{occ} must correspond to a term not containing \texttt{if}, \texttt{let}, \texttt{<-}, \texttt{find}, \texttt{<-R}, \texttt{new}, \texttt{event}, \texttt{event\_abort}, \texttt{insert}, \texttt{get}.
• **guess** 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. It performs several merge 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** $x_{i1} \ldots x_{in} \ldots , \ldots , x_{k1} \ldots x_{kn}$ takes as argument $k$ lists of $n$ variables separated by commas. It merges the variables $x_{i1}, \ldots , x_{in}$ into $x_{i1}$. This is useful when these variables play the same role in different branches of if, find, let: merging them into a single variable may allow to merge the branches of if, find, let by **merge_branches**.

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

In general, the variables $x_{ij}$ 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 $x_{ij}$ 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 $x_{ij}$ which are used only for testing that branch $j$ has been executed. This allows the merge to succeed more often.

• **guess** $(\text{guessspec})$ guesses the value of $(\text{guessspec})$, 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. $\text{occ}$, where $\text{occ}$ is the occurrence of a replication; then one guesses the value of the replication index of that replication in the tested session.

3. $i \& \& \text{above}$, 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. $\text{occ} \& \& \text{above}$, where $\text{occ}$ 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. "$x[i_1, \ldots , i_n]!n$", where $x[i_1, \ldots , i_n]$ is the variable to be guessed, defined under $n$ replications, and $i_1, \ldots , i_n$ 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 $i_1, \ldots , i_n$ or not. Otherwise, the transformation fails. When $x$ is defined under no replication, one writes $x$ instead of "$x[i_1, \ldots , i_n]!n$".

CryptoVerif distinguishes whether the element specified by $(\text{guessspec})$ is equal to a constant value $v_{\text{tested}}$ by introducing a test under the definition of that element, and tries to prove the security properties for $(\text{guessspec}) = v_{\text{tested}}$. The queries are adjusted accordingly. The probability of breaking the initial query is typically the size of the guessed element $(\text{guessspec})$ (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 $(\text{guessspec}) = v_{\text{tested}}$. The size of the guessed element must be estimated less than $\text{maxGuess}$ (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 \"x[i_1, \ldots, i_n]\" no_test** guesses the value of \(x[i_1, \ldots, i_n]\), where \(x[i_1, \ldots, i_n]\) is defined under \(n\) replications, and \(i_1, \ldots, i_n\) 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 \(i_1, \ldots, i_n\) or not. Otherwise, the transformation fails. When \(x\) is defined under no replication, one writes \(x\) instead of "\(x[i_1, \ldots, i_n]\". The variable \(x\) must be defined by \(\text{let } x = \ldots\)

CryptoVerif replaces the definition of \(x\) with a constant value \(v_{\text{tested}}\). When the definition of \(x\) is not a simple term (a term built from replication indices, variables, and function applications), the old definition is still evaluated but ignored.

Assuming \(x\) is a variable of type \(T\), the probability of breaking the initial query is \(|T|\) times the probability of breaking the query for \(x[i_1, \ldots, i_n] = v_{\text{tested}}\). \(|T|\) must be estimated less than \(\text{maxGuess}\) (see the command set \(\text{maxGuess}\)).

This transformation does not apply to secrecy queries, 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 \(\text{bad_guess}\).

(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 probabilities 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 \(\text{guess_branch}\), use focus to limit oneself to the queries that can be proved and redo the proof from \(\text{guess_branch}\).

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

- **guess_branch occ no_test guesses which branch is taken at program point occ.** The instruction at program point \(occ\) must be a test if. 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). CryptoVerif generates 2 games \(G_0\) and \(G_1\), keeping respectively only the else branch and the then branch. The test is removed. If the condition of the test is not a simple term (a term built from replication indices, variables, and function applications), it is still evaluated but ignored. The desired queries must then be proved in both games \(G_i\) and the probability of breaking the queries in the initial game is bounded by the sum of the probabilities of breaking them in both 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 game $G_1$.

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.

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

- **move_if_fun** *arg*: moves the predefined function if_fun or transforms it into a term if ... then ... else ... It supports the following variants:
  - **move_if_fun** *loc_1 ... loc_n*, where each *loc_j* is either a program point or a function symbol. When *loc_j* is a program point, moves occurrences of if_fun from inside the term at that program point to the root of that term. When *loc_j* is a function symbol, moves occurrences of if_fun from under that function symbol to just above it.
  - **move_if_fun** *level n*, where *n* is a positive integer. Moves occurrences of if_fun *n* function symbols up in the syntax tree (provided those if_fun occur under at least *n* function symbols).
  - **move_if_fun** *to_term occ_1 ... occ_n* transforms the function symbols if_fun that occur at program points *occ_1*, ..., *occ_n* into terms if ... then ... else ... When no program point is given, performs that transformation everywhere in the game. When autoExpand = true (the default), a call to expand is automatically performed after move_if_fun, which transforms the terms if ... then ... else ... into processes.

- **use** $[n_1, ..., n_k]$: $n_1', ..., n_k'$[command] activates the equations, collisions, and equiv statements named $n_1, ..., n_k$, and deactivates those named $n_1', ..., n_k'$. Active equiv statements are used by the auto command. Active equations and collisions are used by many other commands (simplify, replace, success, ...). When the proof command command is present, the change applies only to that command, and the previously activated equations, collisions, and equiv statements are restored after running the command. When command is absent, the change applies to all subsequent commands.

The names $n_1, ..., n_k$ and $n_1', ..., n_k'$ can be id or id(f) where id and f are identifiers. They can also be revert:id or revert:id(f). In this case, they designate the reverse equation of the equation named id, resp. id(f), that is, the one obtained by swapping both sides of the equality given in the equation. (revert: never applies to collisions and equiv statements.) When an equation is activated, its reverse equation is automatically deactivated to avoid trivial loops.

- **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 initially 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_abort 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 -o commands 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 
\[(\text{psize})_1^{*n_1} \times \cdots \times (\text{psize})_k^{*n_k} / (\text{pest})\],
where the exponents \(n_i\) can be omitted when equal to 1, writing \((\text{psize})_i\), instead of \((\text{psize})_i^{*1}\), and the whole factor \((\text{psize})_1^{*n_1} \times \cdots \times (\text{psize})_k^{*n_k}\) is replaced with 1 when there is no \((\text{psize})_i\), factor at all; \((\text{psize})_i\) is an identifier that determines the size of a parameter: \(\text{size}\) 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; \((\text{pest})\)
(probability estimate) is an identifier such that \(1/(\text{pest})\) estimates a probability. It can take the following values: \(\text{psize}\) means that the probability \(1/(\text{pest})\) is at most \(2^{-n}\); \(\text{password}\) is equivalent to \(\text{pest}\) \(20\), i.e. the probability \(1/(\text{pest})\) is at most \(2^{-20}\); \(\text{large}\) is equivalent to \(\text{pest}\) \(160\), i.e. the probability \(1/(\text{pest})\) is at most \(2^{-160}\). (See also the declarations \(\text{param}\), \(\text{proba}\), and \(\text{type}\) for explanations of these estimates.)
Collisions are eliminated when, for some formula \((\text{psize})_1^{*n_1} \times \cdots \times (\text{psize})_k^{*n_k} / (\text{pest})\) in the list (formulas), their probability is at most of the form
\[\text{constant} \times p_{i_1}^{n_{i_1}} \times \cdots \times p_{i_k}^{n_{i_k}} \times \text{proba}_0\],
where \(p_i\) is a parameter of size at most \((\text{psize})_i\), and the estimate of the probability \(\text{proba}_0\) is at most \(1/(\text{pest})\). This condition is applied both for collisions between elements of a type \(T\), in which case \(\text{proba}_0 = \text{Pcoll}1\text{rand}(T)\), and for collision statements, in which case \(\text{proba}_0\) 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 \(\text{anything} \times \text{proba}_0\) when \(\text{proba}_0 \leq 2^{-155}\) (where \(\text{large}\) would mean \(2^{-160}\)), to allow collisions that have probability a small factor times collisions between elements of a large type, and for \(p \times \text{proba}_0\) when \(p \leq 2^2\) (small) and \(\text{proba}_0 \leq 2^{-20}\) (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 \(\text{constant} \times p_{i_1}^{n_{i_1}} \times \cdots \times p_{i_k}^{n_{i_k}} \times \text{Pcoll}1\text{rand}(T)\) 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 \(\text{constant} \times p_{i_1}^{n_{i_1}} \times \cdots \times p_{i_k}^{n_{i_k}}\) times a collision statement remains negligible.
- `allowed_collisions (pest)`: where \((\text{pest})\) estimates a probability: \(\text{psize}\) means that the probability is at most \(2^{-n}\); \(\text{password}\) is equivalent to \(\text{pest}\) \(20\), i.e. probability at most \(2^{-20}\); \(\text{large}\) is equivalent to \(\text{pest}\) \(160\), 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 \((\text{pest})\). This behavior fits the exact security framework nicely: we eliminate collisions when we have a small enough probability.

- `focus "(querydecl)" \ldots \"(querydecl)\"`: where \((\text{querydecl}) := \text{query} \ \text{seq} (\text{vartyp eb}) ; \ (\text{query}) \); \((\text{query})\) 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 \text{seq} (\text{vartyp eb}). 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 CryptoVerif

CryptoVerif 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: `#O` designates the number of oracles calls.

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 `\text{T}E\text{X}` file containing the sequence of games and the proved properties.

**Correspondence between ASCII and `\text{T}E\text{X}` outputs** To use nicer and more conventional notations, the `\text{T}E\text{X}` output sometimes differs from the ASCII output. Here is a table of correspondence:
9 Generation of OCaml Implementations

CryptoVerif can generate an OCaml implementation of the protocol from the CryptoVerif specification, using the command-line option `-impl OCaml`. When the CryptoVerif file uses `diff`, the implementation is generated using the first argument of `diff`. (The second argument is supposed to define a specification, and CryptoVerif shows indistinguishability between the protocol and the specification.) When the CryptoVerif file uses `equivalence`, the implementation is generated using the first process. (Again, the second process is supposed to define a specification.)

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 `cv2OCaml` 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` [https://github.com/xavierleroy/cryptokit](https://github.com/xavierleroy/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 cv2OCaml/nspk and cv2OCaml/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 (ident)(seq(pattern)) := .../in and a return/out 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 (ident)(seq(pattern)) := /in statement, and $T'_1 \times \ldots \times T'_n$ are the types of the result given in the return/out statements, and
- the list containing for each of the following oracles, its name and whether it is under a replication or not.

An oracle can have multiple return/out statements. To be able to implement it, we must be able to define the signature above for each oracle, that is, all return/out must return the same type of elements, and the oracles present after each return/out 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:
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:
- `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 $ty \to \text{string}$, the deserialization function $d$ must be of type $\text{string} \to ty$. When deserialization fails, it must raise an exception.

- `pred=\"p\"`: Sets the predicate function, this function must be an OCaml function of type $ty \to \text{bool}$. It returns whether an element is of type $T$ or not. The default predicate function is a function that accepts every element.

- `random=\"r\"`: Sets the random generation function. This function must be an OCaml function of type $\text{unit} \to ty$, 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 $sl_1 \to sl_2 \to \ldots \to sl_n \to st$, where for all $i$ between 1 and $n$, $sl_i$ 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 $\text{unit} \to st$, 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 $st \to sl_1 \times sl_2 \times \ldots \times sl_n$. $s_inv$ 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 Generation of F* Implementations

CryptoVerif can also generate an $F^*$ implementation of the protocol from the CryptoVerif specification, using the command-line option `-impl FStar`. It works along similar principles to the OCaml generation of implementations, but additionally generates $F^*$ lemmas for equations used as assumptions in CryptoVerif, to be proved in $F^*$. In the future, we plan to generate $F^*$ axioms for security properties proved by CryptoVerif.

The subdirectory `cv2fstar` contains files for CryptoVerif to $F^*$ as well as an example, the Needham-Schroeder-Lowe public-key protocol.

More documentation is going to be added here in the future.
11 Additional Programs

11.1 test

Usage:

```
test [-timeout \(n\)] \(\langle\text{mode}\rangle\) \(\langle\text{test\_set}\rangle\)
```

where `-timeout \(n\)` sets the timeout for each execution of the tested program to \(n\) seconds (by default, there is no timeout), \(\langle\text{mode}\rangle\) 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 \(\langle\text{test\_set}\rangle\) 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
- `cv2EasyCrypt`: runs tests of the translation of CryptoVerif assumptions to Easy-Crypt (always tests the scripts; there are currently no expected results in the files)
- `cv2OCaml`: runs tests of the generation of OCaml implementations
- `cv2Fstar`: runs tests of the generation of F* implementations
- `all`: runs all tests included in `basic`, `proverif`, `converted`, `big`, `cv2OCaml`, and `cv2Fstar`

\(\langle\text{test\_set}\rangle\) can be omitted when it is `basic`, and \(\langle\text{mode}\rangle\) \(\langle\text{test\_set}\rangle\) 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 run 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.

11.2 analyze

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

Usage:

```
analyze [\(\langle\text{options}\rangle\)] \(\langle\text{prog}\rangle\) \(\langle\text{mode}\rangle\) \(\langle\text{tmp\_directory}\rangle\) \(\langle\text{prefix\_for\_output\_files}\rangle\) \(\langle\text{dirs\_directories}\rangle\)
analyze [\(\langle\text{options}\rangle\)] \(\langle\text{prog}\rangle\) \(\langle\text{mode}\rangle\) \(\langle\text{tmp\_directory}\rangle\) \(\langle\text{prefix\_for\_output\_files}\rangle\) \(\langle\text{file\_directory}\rangle\) \(\langle\text{filename}\rangle\)
```

where \(\langle\text{options}\rangle\) 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, .pl, .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 warnings and 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.

11.3 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 EXPECT 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 Bruno Blanchet and David Cadé 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 SSA Formacrypt, by the ANR projects ProSe (decision ANR-2010-VERS-004-01) and TECAP (decision ANR-17-CE39-0004-03), and benefited from funding managed by the French National Research Agency under the France 2030 programme with the references ANR-22-PECY-0006 (PEPR Cybersecurity SVP) and ANR-22-PETQ-0008 (PEPR Quantic PQ-TLS).

References


89

