CryptoVerif:
A Computationally-Sound Security Protocol Verifier

Bruno Blanchet
Inria Paris, France
`Bruno.Blanchet@inria.fr`

November 20, 2017

Abstract

This document presents the security protocol verifier CryptoVerif. In contrast to most previous provers, CryptoVerif does not rely on the Dolev-Yao model, but on the computational model. It can verify secrecy and correspondence properties (which include authentication). It produces proofs presented as sequences of games, like those manually written by cryptographers; these games are formalized in a probabilistic polynomial-time process calculus. CryptoVerif provides a generic method for specifying security properties of the cryptographic primitives. It produces proofs valid for a any number of sessions of the protocol, and provides an upper bound on the probability of success of an attack against the protocol as a function of the probability of breaking each primitive and of the number of sessions. It can work automatically, or the user can guide it with manual proof indications.

1 Introduction

There exist two main approaches for analyzing security protocols. In the computational model, messages are bitstrings, and the adversary is a probabilistic polynomial-time Turing machine. This model is close to the real execution of protocols, but the proofs are usually manual and informal. In contrast, in the symbolic, Dolev-Yao model, cryptographic primitives are considered as perfect blackboxes, modeled by function symbols in an algebra of terms, possibly with equations. The adversary can compute using these blackboxes. This abstract model makes it easier to build automatic verification tools, but the security proofs are in general not sound with respect to the computational model.

In contrast to most protocol verifiers, CryptoVerif works directly in the computational model, without considering the Dolev-Yao model. It produces proofs valid for any number of sessions of the protocol, in the presence of an active adversary. These proofs are presented as sequences of games, as used by cryptographers [3, 9, 10]: the initial game represents the protocol to prove; the goal is to bound the probability of breaking a certain security property in this game; intermediate games are obtained each from the previous one by transformations such that the difference of probability between consecutive games can easily be bounded; the final game is such that the desired probability is obviously bounded from the form of the game. (In general, it is simply 0 in that game.) The desired probability can then be easily bounded in the initial game.

We represent games in a process calculus. This calculus is inspired by the pi-calculus and by the calculi of [8] and of [7]. In this calculus, messages are bitstrings, and cryptographic primitives are functions from bitstrings to bitstrings. The calculus has a probabilistic semantics. The main tool for specifying security properties is indistinguishability: Q is indistinguishable from Q' up to probability p, $Q \approx_p Q'$, when the adversary has probability at most p of distinguishing Q from Q'. With respect to previous calculi mentioned above, our calculus introduces an important novelty which is key for the automatic proof of security protocols: the values of all variables during the execution of a process are stored in arrays. For instance, $x[i]$ is the value of x in the i-th copy of the process that defines x. Arrays replace lists often used by cryptographers in their manual proofs of protocols. For example, consider the standard security assumption on a message authentication code (MAC). Informally, this definition says that the adversary has a negligible probability of forging a MAC, that is, that all correct MACs have been computed by calling the MAC oracle (*i.e.*, function). So, in cryptographic proofs, one defines a list containing the arguments of calls to the MAC oracle, and when checking a MAC of a message m, one can additionally check that m is in this list, with a negligible change in probability. In our calculus, the arguments of the MAC oracle are stored in arrays, and we perform a lookup in these arrays in order to find the message m. Arrays make it easier to automate proofs since they are always present in the calculus: one does not need to add explicit instructions to insert values in them, in contrast to the lists used in manual proofs. Therefore, many trivially sound but difficult to automate syntactic transformations disappear. Furthermore, relations between elements of arrays can easily be expressed by equalities, possibly involving computations on array indices.

CryptoVerif relies on a collection of game transformations, in order to transform the initial protocol into a game on which the desired security property is obvious. The most important kind of transformations exploits the security assumptions on cryptographic primitives in order to obtain a simpler game. These transformations can be specified in a generic way: we represent the security assumption of each cryptographic primitive by an observational equivalence $L \approx_p R$, where the processes L and R encode oracles: they input the arguments of the oracle and send its result back. Then, the prover can automatically transform a process Q that calls the oracles of L (more precisely, contains as subterms terms that perform the same computations as oracles of L) into a process Q' that calls the oracles of R instead. We have used this technique to specify several variants of shared-key and public-key encryption, signature, message authentication codes, hash functions, Diffie-Hellman key agreement, simply by giving the appropriate equivalence $L \approx_p R$ to the prover. Other game transformations are syntactic transformations, used in order to be able to apply an assumption on a cryptographic primitive, or to simplify the game obtained after applying such an assumption.

In order to prove protocols, these game transformations are organized using a proof strategy based on advice: when a transformation fails, it suggests other transformations that should be applied before, in order to enable the desired transformation. Thanks to this strategy, protocols can often be proved in a fully automatic way. For delicate cases, CryptoVerif has an interactive mode, in which the user can manually specify the transformations to apply. It is usually sufficient to specify a few transformations coming from the security assumptions of primitives, by indicating the concerned cryptographic primitive and the concerned secret key if any; the prover infers the intermediate syntactic transformations by the advice strategy. This mode is helpful for proving some public-key protocols, in which several security assumptions on primitives can be applied, but only one leads to a proof of the protocol. Importantly, CryptoVerif is always sound: whatever indications the user gives, when the prover shows a security property of the protocol, the property indeed holds assuming the given assumptions on the cryptographic primitives.

CryptoVerif has been implemented in Ocaml (29800 lines of code for version 1.12 of CryptoVerif) and is available at `http://cryptoverif.inria.fr/`.

Outline Currently, this document only presents the process calculus that CryptoVerif uses for representing games, with its syntax, type system and formal semantics, in the next section. We plan to add information on the game transformations, the proof strategy, and the algorithms for proving security properties in the future.

Notations We recall the following standard notations. We denote by $\{M_1/x_1, \dots, M_m/x_m\}$ the substitution that replaces x_j with M_j for each $j \leq m$. The cardinal of a set or multiset S is denoted by $|S|$. We use $\uplus$ for multiset union. When S is a multiset, $S(x)$ is the number of elements of S equal to x. If S is a finite set, $x \overset{R}{\leftarrow} S$ chooses a random element uniformly in S and assigns it to x. If $\mathcal{A}$ is a probabilistic algorithm, $x \leftarrow \mathcal{A}(x_1, \dots, x_m)$ denotes the experiment of choosing random coins r and assigning to x the result of running $\mathcal{A}(x_1, \dots, x_m)$ with coins r. Otherwise, $x \leftarrow M$ is a simple assignment statement. If D is a discrete probability distribution, we denote by $D(a)$ the probability that $X = a$, $\Pr[X = a]$, where X is a random variable with probability distribution D.

2 A Calculus for Cryptographic Games

2.1 Syntax and Informal Semantics

CryptoVerif represents games in the syntax of Figure 1. This calculus assumes a countable set of channel names, denoted by c. It uses parameters, denoted by n, which are integers that bound the number of executions of processes.

It also uses types, denoted by T, which are sets of values. A type is *fixed* when it is the set of all bitstrings of a certain length; a type is *bounded* when it is a finite set. Particular types are predefined: *bool* $= \{\text{true}, \text{false}\}$, where false is 0 and true is 1; *bitstring* is a the set of all bitstrings; *bitstring*$_\bot$ $=$ *bitstring* $\cup \{\bot\}$ where $\bot$ is a special symbol; $[1, n]$ where n is a parameter. (We consider integers as bitstrings without leading zeroes.)

The calculus also uses function symbols f. Each function symbol comes with a type declaration $f : T_1 \times \ldots \times T_m \to T$, and represents an efficiently computable, deterministic function that maps each tuple in $T_1 \times \ldots \times T_m$ to an element of T. Particular functions are predefined, and some of them use the infix notation: $M = N$ for the equality test, $M \neq N$ for the inequality test (both taking two values of the same type T and returning a value of type *bool*), $M \vee N$ for the boolean or, $M \wedge N$ for the boolean and, $\neg M$ for the boolean negation (taking and returning values of type *bool*), tuples $(M_1, \dots, M_m)$ (taking values of any types and returning values of type *bitstring*; tuples are assumed to provide unambiguous concatenation, with tags for the types of $M_1, \dots, M_m$ so that tuples of different types are always different).

In this calculus, terms represent computations on bitstrings. The replication index i is an integer which serves in distinguishing different copies of a replicated process $!^{i \leq n}$. (Replication indices are typically used as array indices.) The variable access $x[M_1, \dots, M_m]$ returns the content of the cell of indices $M_1, \dots, M_m$ of the m-dimensional array variable x. We use x, y, z, u as variable names. The function application $f(M_1, \dots, M_m)$ returns the result of applying function f to $M_1, \dots, M_m$ Terms contain additional constructs which are very similar to those also included in output processes and explained below. These constructs conclude by evaluating a term, instead of executing a process. The construct event_abort e executes event e (without argument) and aborts the game; it is in fact intended for use in the definition of cryptographic primitives.

The calculus distinguishes two kinds of processes: input processes Q are ready to receive a message on a channel; output processes P output a message on a channel after executing some internal computations. The input process 0 does nothing; $Q \mid Q'$ is the parallel composition of Q and Q'; $!^{i \leq n} Q$ represents n copies of Q in parallel, each with a different value of $i \in [1, n]$; newChannel $c; Q$ creates a new private channel c and executes Q; this construct is useful in proofs, but does not occur in games manipulated by CryptoVerif. The semantics of the input $c[M_1, \dots, M_l](p); P$ will be explained below together with the semantics of the output.

The output process new $x[\widetilde{i}] : T; P$ chooses a new random value in T, stores it in $x[\widetilde{i}]$, and executes P. The abbreviation $\widetilde{i}$ stands for a sequence of replication indices $i_1, \dots, i_m$. The random value is chosen according to the default distribution D_T for type T, which is determined

$M, N ::=$	terms
i	replication index
$x[M_1, \dots, M_m]$	variable access
$f(M_1, \dots, M_m)$	function application
$\mathsf{new}\ x[\widetilde{i}] : T; N$	random number
$\mathsf{let}\ p = M\ \mathsf{in}\ N\ \mathsf{else}\ N'$	assignment (pattern-matching)
$\mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ N$	assignment
$\mathsf{if}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ N\ \mathsf{else}\ N'$	conditional
$\mathsf{find}[\mathit{unique}?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}$ $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M'_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N'$	array lookup
$\mathsf{insert}\ \mathit{Tbl}(M_1, \dots, M_l); N$	insert in table
$\mathsf{get}\ \mathit{Tbl}(p_1, \dots, p_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ N\ \mathsf{else}\ N'$	get from table
$\mathsf{event}\ e(M_1, \dots, M_l); N$	event
$\mathsf{event\_abort}\ e$	event e and abort
$p ::=$	pattern
$x[\widetilde{i}] : T$	variable
$f(p_1, \dots, p_m)$	function application
$=M$	comparison with a term
$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq n} Q$	replication n times
$\mathsf{newChannel}\ c; Q$	channel restriction
$c[M_1, \dots, M_l](p); P$	input
$P ::=$	output process
$\overline{c[M_1, \dots, M_l]}\langle N \rangle; Q$	output
$\mathsf{new}\ x[\widetilde{i}] : T; P$	random number
$\mathsf{let}\ p = M\ \mathsf{in}\ P\ \mathsf{else}\ P'$	assignment
$\mathsf{if}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$	conditional
$\mathsf{find}[\mathit{unique}?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}$ $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$	array lookup
$\mathsf{insert}\ \mathit{Tbl}(M_1, \dots, M_l); P$	insert in table
$\mathsf{get}\ \mathit{Tbl}(p_1, \dots, p_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$	get from table
$\mathsf{event}\ e(M_1, \dots, M_l); P$	event
$\mathsf{event\_abort}\ e$	event e and abort
yield	end

Figure 1: Syntax of the process calculus

as follows:

- When the type T is declared with option *nonuniform*, the default probability distribution D_T for type T may be non-uniform. It is left unspecified.
- Otherwise, if T is *fixed*, T consists of all bitstrings of a certain length, and the default distribution is the uniform distribution. The probability of each element of T is $1/|T|$.
- If T is *bounded* but not *fixed*, T is finite, and the default distribution is an approximately uniform distribution, such that its distance to the uniform distribution is at most ϵ_T. The distance between two probability distributions D_1 and D_2 for type T is

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

  Indeed, probabilistic Turing machines that run in bounded time cannot choose random elements exactly uniformly in sets whose cardinal is not a power of 2.

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

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

  so

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

  Therefore

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

  so we can take $\epsilon_T = \frac{m}{2^k}$. A given precision of $\epsilon_T = \frac{1}{2^{k'}}$ can be obtained by choosing $k = (k' + \text{number of bits of } m)$ random bits.

  By default, CryptoVerif does not display ϵ_T in probability formulas, to make them more readable.

When T is not declared with any of the options *nonuniform*, *fixed*, or *bounded*, CryptoVerif rejects the construct $\mathsf{new}\ x[\widetilde{i}] : T; P$. Function symbols represent deterministic functions, so all random numbers must be chosen by $\mathsf{new}\ x[\widetilde{i}] : T$. Deterministic functions make automatic syntactic manipulations easier: we can duplicate a term without changing its value.

The process $\mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ P$ stores the value of M (which must be in T) in $x[\widetilde{i}]$ and executes P. Furthermore, we say that a function $f : T_1 \times \ldots \times T_m \to T$ is *efficiently injective* when it is injective and its inverses are efficiently computable, that is, there exist functions $f_j^{-1} : T \to T_j$ $(1 \leq j \leq m)$ such that $f_j^{-1}(f(x_1, \ldots, x_m)) = x_j$ and f_j^{-1} is efficiently computable. When f is efficiently injective, we define a pattern matching construct $\mathsf{let}\ f(x_1, \ldots, x_m) = M\ \mathsf{in}\ P\ \mathsf{else}\ Q$ as an abbreviation for $\mathsf{let}\ y : T = M\ \mathsf{in}\ \mathsf{let}\ x'_1 : T_1 = f_1^{-1}(y)\ \mathsf{in}\ \ldots \mathsf{let}\ x'_m : T_m = f_m^{-1}(y)\ \mathsf{in}\ \mathsf{if}\ f(x'_1, \ldots, x'_m) = y\ \mathsf{then}\ (\mathsf{let}\ x_1 : T_1 = x'_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_m : T_m = x'_m\ \mathsf{in}\ P)\ \mathsf{else}\ Q$ where $y, x'_1, \ldots, x'_m$ are fresh variables. (The variables $x'_1, \ldots, x'_m$ are introduced to make sure that none of the variables $x_1, \ldots x_m$ is defined when the pattern-matching fails.) We naturally generalize this construct to $\mathsf{let}\ p = M\ \mathsf{in}\ P\ \mathsf{else}\ Q$ where p is built from variables, efficiently injective functions, and equality tests. When p is simply a variable, the pattern-matching always succeeds, so the else branch of the assignment is never executed and can be omitted.

The process $\mathsf{event}\ e(M_1, \dots, M_l); P$ executes the event $e(M_1, \dots, M_l)$, then runs P. This event records that a certain program point has been reached with certain values of $M_1, \dots, M_l$, but otherwise does not affect the execution of the process.

The process $\mathsf{event\_abort}\ e$ executes event e (without argument) and aborts the game.

Next, we explain the process $\mathsf{find}[\mathit{unique}?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$. The order and array indices on tuples are taken component-wise, so for instance, $u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}$ can be further abbreviated $\widetilde{u_j}[\widetilde{i}] = \widetilde{i_j} \leq \widetilde{n_j}$. A simple example is the following: $\mathsf{find}\ u = i \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[i]) \wedge x[i] = a\ \mathsf{then}\ P'\ \mathsf{else}\ P$ 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 it in u and executes P' with that value of u; otherwise, it executes P. In other words, this find construct looks for the value a in the array x, and when a is found, it stores in u an index such that $x[u] = a$. Therefore, the find construct allows us to access arrays, which is key for our purpose. More generally, $\mathsf{find}\ u_1[\widetilde{i}] = i_1 \leq n_1, \dots, u_m[\widetilde{i}] = i_m \leq n_m\ \mathsf{suchthat}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ P'\ \mathsf{else}\ P$ tries to find values of $i_1, \dots, i_m$ for which $M_1, \dots, M_l$ are defined and M is true. In case of success, it stores the obtained values in $u_1[\widetilde{i}], \dots, u_m[\widetilde{i}]$ and executes P'. In case of failure, it executes P. This is further generalized to m branches: $\mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$ tries to find a branch j in $[1, m]$ such that there are values of $i_{j1}, \dots, i_{jm_j}$ for which $M_{j1}, \dots, M_{jl_j}$ are defined and M_j is true. In case of success, it stores them in $u_{j1}[\widetilde{i}], \dots, u_{jm}[\widetilde{i}]$ and executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ for each j and each value of $i_{j1}, \dots, i_{jm_j}$ in $[1, n_{j1}] \times \dots \times [1, n_{jm_j}]$. If none of these conditions is true, it executes P. Otherwise, it chooses randomly one j and one value of $i_{j1}, \dots, i_{jm_j}$ such that the corresponding condition is true, according to the distribution $D_{\mathsf{find}}(S)$ where S is the set of possible solutions $j, i_{j1}, \dots, i_{jm_j}$, stores it in $u_{j1}[\widetilde{i}], \dots, u_{jm_j}[\widetilde{i}]$, and executes P_j. The distribution $D_{\mathsf{find}}(S)$ is almost uniform: formally, the distance between $D_{\mathsf{find}}(S)$ and the uniform distribution is at most ϵ_{find}, that is, $d(D_{\mathsf{find}}(S), \mathit{uniform}) \leq \epsilon_{\mathsf{find}}$. By default, CryptoVerif does not display ϵ_{find} in probability formulas, to make them more readable. We cannot take the first element found because the game transformations made by CryptoVerif may reorder the elements. For these transformations to preserve the behavior of the game, the distribution of the chosen element must be invariant by reordering, up to a small probability ϵ_{find}. In this definition, the variables $i_{j1}, \dots, i_{jm_j}$ are considered as replication indices, while $u_{j1}[\widetilde{i}], \dots, u_{jm_j}[\widetilde{i}]$ are considered as array variables. The indication $[\mathit{unique}?]$ stands for either $[\mathsf{unique}_e]$ or empty. The empty case has just been explained. When the find is marked $[\mathsf{unique}_e]$ and there are several solutions (that is, $k > 1$), we execute the event e and abort the game. When there is zero or one solution (that is, $k = 0$ or $k = 1$), the find is executed as when $[\mathit{unique}?]$ is empty. This semantics allows us to perform game transformations that require the find to have a single solution.

The conditional $\mathsf{if}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$ executes P if $M_1, \dots, M_l$ are defined and M evaluates to true. Otherwise, it executes P'. This conditional is equivalent to $\mathsf{find}\ \mathsf{suchthat}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$. The conjunct $\mathsf{defined}(M_1, \dots, M_l)$ can be omitted when $l = 0$ and M can be omitted when it is true.

The constructs insert and get handle tables, used for instance to store the keys of the protocol participants. A table can be represented as a list of tuples; $\mathsf{insert}\ \mathit{Tbl}(M_1, \dots, M_l); P$ inserts the element $M_1, \dots, M_l$ in the table Tbl; $\mathsf{get}\ \mathit{Tbl}(x_1 : T_1, \dots, x_l : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$ tries to retrieve an element $(x_1, \dots, x_l)$ in the table Tbl such that M is true. When such an element is found, it executes P with $x_1, \dots, x_l$ bound to that element. (When several such elements are found, one of them is chosen randomly according to distribution $D_{\mathsf{get}}(\{1, \dots, |L|\})$ where L is the list of suitable elements, with $d(D_{\mathsf{get}}(\{1, \dots, |L|\}), \mathit{uniform}) \leq \epsilon_{\mathsf{find}}$.) When no such element is found, P' is executed. We can generalize this construct to patterns instead of variables similarly to the let case. CryptoVerif internally translates the insert and get constructs into find.

Let us explain the output $\overline{c[M_1, \dots, M_l]}\langle N\rangle; Q$. A channel $c[M_1, \dots, M_l]$ consists of both a channel name c and a tuple of terms $M_1, \dots, M_l$. Channel names c can be declared private by newChannel c; the adversary can never have access to channel $c[M_1, \dots, M_l]$ when c is private. (This is useful in the proofs, although all channels of protocols are often public.) Terms $M_1, \dots, M_l$ are intuitively analogous to IP addresses and ports, which are numbers that the adversary may guess. A semantic configuration always consists of a single output process (the process currently being executed) and several input processes. When the output process executes $\overline{c[M_1, \dots, M_l]}\langle N\rangle; Q$, one looks for an input on channel $c[M'_l \dots, M'_l]$, where $M'_1, \dots, M'_l$ evaluate to the same bitstrings as $M_1, \dots, M_l$, in the available input processes. If no such input process is found, the process blocks. Otherwise, one such input process $c[M'_1, \dots, M'_l](x[\widetilde{i}] : T); P$ is chosen randomly according to the probability distribution $D_{\mathsf{in}}(S)$ where S is the multiset of suitable input processes. The communication is then executed: the output message N is evaluated and stored in $x[\widetilde{i}]$ if it is in T (otherwise the process blocks). Finally, the output process P that follows the input is executed. The input process Q that follows the output is stored in the available input processes for future execution. The input construct can be generalized to patterns instead of variables similarly to the let case; when pattern-matching fails, the input process executes yield. The syntax requires an output to be followed by an input process, as in [7]. 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.

Using different channels for each input and output allows the adversary to control the network. For instance, we may write $!^{i \le n} c[i](x[i] : T) \dots \overline{c'[i]}\langle M\rangle \dots$ The adversary can then decide which copy of the replicated process receives its message, simply by sending it on $c[i]$ for the appropriate value of i.

The yield construct is an abbreviation for $\overline{yield}\langle()\rangle$. By performing an output, this construct returns control to the adversary, which is going to receive the message. An else branch of find, if, get, or let may be omitted when it is else yield. (Note that "else 0" would not be syntactically correct.) Similarly, ; yield may be omitted after event, new, or insert and in yield may be omitted after let. A trailing 0 after an output may be omitted.

The *current replication indices* at a certain program point in a process are the replication indices $i_1, \dots, i_m$ bound by replications and find above that program point. The replication $!^{i \le n} Q$ binds the replication index i in Q. The find construct find[*unique*?] $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \le n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \le n_{jm_j}$ suchthat defined$(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ then $\dots)$ else $\dots$ binds the replication indices $i_{j1}, \dots, i_{jm_j}$ in defined$(M_{j1}, \dots, M_{jl_j}) \wedge M_j$. We often abbreviate $x[i_1, \dots, i_m]$ by x when $i_1, \dots, i_m$ are the current replication indices, but it should be kept in mind that this is only an abbreviation. Variables defined under a replication must be arrays: for example $!^{i_1 \le n_1} \dots !^{i_m \le n_m}$let $x[i_1, \dots, i_m] : T = M$ in $\dots$ More formally, we require the following invariant:

Invariant 1 (Single definition) The process Q_0 satisfies Invariant 1 if and only if

1. in every definition of $x[i_1, \dots, i_m]$ in Q_0, the indices $i_1, \dots, i_m$ of x are the current replication indices at that definition, and

2. two different definitions of the same variable x in Q_0 are in different branches of a find (or if or let) or get.

Invariant 1 guarantees that each variable is assigned at most once for each value of its indices. (Indeed, item 2 shows that only one definition of each variable can be executed for given indices in each trace.) A definition of $x[\widetilde{i}]$ can be new $x[\widetilde{i}] : T$, a let, get, or input that contains the pattern $x[\widetilde{i}] : T$, or find $\dots x[\widetilde{i}] = i \le n \dots$.

Invariant 2 (Defined variables) The process Q_0 satisfies Invariant 2 if and only if every occurrence of a variable access $x[M_1, \dots, M_m]$ in Q_0 is either

- syntactically under the definition of $x[M_1, \dots, M_m]$ (in which case $M_1, \dots, M_m$ are in fact the current replication indices at the definition of x);
- or in a defined condition in a find process or term;
- or in M'_j in a process or term of the form find $(\bigoplus_{j=1}^{m''} \widetilde{u_j}[\widetilde{i}] = \widetilde{i_j} \leq \widetilde{n_j}$ suchthat defined$(M'_{j1}, \dots, M'_{jl_j}) \wedge M'_j$ then $P_j)$ else P where for some $k \leq l_j$, $x[M_1, \dots, M_m]$ is a subterm of M'_{jk}.
- or in P_j in a process or term of the form find $(\bigoplus_{j=1}^{m''} \widetilde{u_j}[\widetilde{i}] = \widetilde{i_j} \leq \widetilde{n_j}$ suchthat defined$(M'_{j1}, \dots, M'_{jl_j}) \wedge M'_j$ then $P_j)$ else P where for some $k \leq l_j$, there is a subterm N of M'_{jk} such that $N\{\widetilde{u_j}[\widetilde{i}]/\widetilde{i_j}\} = x[M_1, \dots, M_m]$.

Invariant 2 guarantees that variables can be accessed only when they have been initialized. It checks that the definition of the variable access is either in scope (first item) or checked by a find (last two items). We recall that if is a particular case of find. The scope of variable definitions is defined as follows: $x[\widetilde{i}]$ is syntactically under its definition when it is

- inside P in new $x[\widetilde{i}] : T; P$;
- inside N in new $x[\widetilde{i}] : T; N$;
- inside P in let $p = M$ in P else P' when $x[\widetilde{i}] : T$ is bound in the pattern p;
- inside N in let $p = M$ in N else N' when $x[\widetilde{i}] : T$ is bound in the pattern p;
- inside N in let $x[\widetilde{i}] : T = M$ in N;
- inside P_j in find$[unique?]$ $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}$ suchthat defined$(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P when x is u_{jk} for some $k \leq m_j$;
- inside N_j in find$[unique?]$ $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}$ suchthat defined$(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ then $N_j)$ else N when x is u_{jk} for some $k \leq m_j$;
- inside M or P in get $Tbl(p_1, \dots, p_l)$ suchthat M in P else P' when $x[\widetilde{i}] : T$ is bound in one of the patterns $p_1, \dots, p_l$;
- inside M or N in get $Tbl(p_1, \dots, p_l)$ suchthat M in N else N' when $x[\widetilde{i}] : T$ is bound in one of the patterns $p_1, \dots, p_l$;
- inside P in $c[M_1, \dots, M_l](p); P$ when $x[\widetilde{i}] : T$ is bound in the pattern p.

A variable access that does not correspond to the first item of Invariant 2 is called an *array access.* We furthermore require the following invariant.

Invariant 3 (Variables defined in find and get conditions) The process Q_0 satisfies Invariant 3 with public variables V if and only if the variables defined in conditions of find and the variables defined in patterns and in conditions of get have no array accesses and are not in the set of variables V.

These conditions are needed for variables of get, because they will be transformed into variables defined in conditions of find by the transformation of get into find.

Invariant 4 (Terms in find and get conditions) The process Q_0 satisfies Invariant 4 if and only if new, event, and insert do not occur in conditions of find and get.

Invariant 4 guarantees that evaluating the condition of a find or get does not change the state of the system.

Invariant 5 (Terms in input channels and defined conditions) The process Q_0 satisfies Invariant 5 if and only if only replication indices, variables, and function applications occur in input channels $c[M_1, \dots, M_l]$ and in conditions defined$(M_1, \dots, M_l)$ in find or if.

Other terms (new, let, if, find, event) are handled by expanding them into their corresponding processes. Invariant 5 is needed because terms in input channels cannot be expanded, as we need an output process to put the computations coming from expanded terms, and similarly terms in defined conditions cannot be expanded. By combining this invariant with Invariant 2, we see that the terms of all variable accesses $x[M_1, \dots, M_m]$ contain only replication indices, variables, and function applications.

The last 3 invariants did not appear in previous versions of the calculus because all terms were only replication indices, variables, and function applications.

Invariant 6 (Events) We distinguish three disjoint sets of events e, Shoup events, non-unique events, and other events. The process Q_0 satisfies Invariant 6 if and only if

- Shoup events occur only in processes of the form event_abort e in Q_0, and
- non-unique events occur only in find[unique$_e$] in Q_0.

The name "Shoup events" is used because these events are introduced when applying Shoup's lemma [11]. The non-unique events are those triggered when a find[unique$_e$] actually has several solutions.

All these invariants are checked by the prover for the initial game and preserved by all game transformations.

We denote by var(P) the set of variables that occur in P, vardef(P) the set of variables defined in P (var(P) may contain more variables than vardef(P) in case some variables are read using find but never defined), and by fc(P) the set of free channels of P. (We use similar notations for input processes.)

2.2 Example

Let us introduce two cryptographic primitives that we use below.

Definition 1 Let T_{mr}, T_{mk}, and T_{ms} be types that correspond intuitively to random seeds, keys, and message authentication codes, respectively; T_{mr} is a fixed-length type. A message authentication code scheme MAC [2] consists of three function symbols:

- mkgen : $T_{mr} \to T_{mk}$ is the key generation algorithm taking as argument a random bitstring and returning a key. (Usually, mkgen is a randomized algorithm; here, since we separate the choice of random numbers from computation, mkgen takes an additional argument representing the random coins.)
- mac : $\mathit{bitstring} \times T_{mk} \to T_{ms}$ is the MAC algorithm taking as arguments a message and a key, and returning the corresponding tag. (We assume here that mac is deterministic; we could easily encode a randomized mac by adding an additional argument as for mkgen.)
- verify : $\mathit{bitstring} \times T_{mk} \times T_{ms} \to \mathit{bool}$ is a verification algorithm such that verify$(m, k, t) =$ true if and only if t is a valid MAC of message m under key k. (Since mac is deterministic, verify(m, k, t) is typically mac$(m, k) = t$.)

We have $\forall m \in \mathit{bitstring}, \forall r \in T_{mr}, \mathrm{verify}(m, \mathrm{mkgen}(r), \mathrm{mac}(m, \mathrm{mkgen}(r))) = \mathrm{true}$.

The advantage of an adversary against unforgeability under chosen message attacks (UF-CMA) is

$$\mathsf{Succ}_{\mathsf{MAC}}^{\mathsf{uf-cma}}(t, q_m, q_v, l) = \max_{\mathcal{A}} \Pr \begin{bmatrix} r \overset{R}{\leftarrow} T_{mr}; k \leftarrow \mathrm{mkgen}(r); \\ (m, s) \leftarrow \mathcal{A}^{\mathrm{mac}(.,k),\mathrm{verify}(.,k,.)} : \mathrm{verify}(m, k, s) \\ \wedge\, m \text{ was never queried to the oracle } \mathrm{mac}(., k) \end{bmatrix}$$

where the adversary $\mathcal{A}$ is any probabilistic Turing machine that runs in time at most t, calls $\mathrm{mac}(., k)$ at most q_m times with messages of length at most l, and calls $\mathrm{verify}(., k, .)$ at most q_v times with messages of length at most l.

$\mathsf{Succ}_{\mathsf{MAC}}^{\mathsf{uf-cma}}(t, q_m, q_v, l)$ is the probability that an adversary forges a MAC, that is, returns a pair (m, s) where s is a correct MAC for m, without having queried the MAC oracle $\mathrm{mac}(., k)$ on m. Intuitively, when the MAC is secure, this probability is small: the adversary has little chance of forging a MAC. Hence, the MAC guarantees the integrity of the MACed message because one cannot compute the MAC without the secret key.

Two frameworks exist for expressing security properties. In the asymptotic framework, used in [4, 5], the length of keys is determined by a security parameter η, and a MAC is UF-CMA when $\mathsf{Succ}_{\mathsf{MAC}}^{\mathsf{uf-cma}}(t, q_m, q_v, l)$ is a negligible function of η when t is polynomial in η. ($f(\eta)$ is *negligible* when for all polynomials q, there exists $\eta_o \in \mathbb{N}$ such that for all $\eta > \eta_0$, $f(\eta) \leq \frac{1}{q(\eta)}$.) The assumption that functions are efficiently computable means that they are computable in time polynomial in η and in the length of their arguments. The goal is to show that the probability of success of an attack against the protocol is negligible, assuming the parameters n are polynomial in η and the network messages are of length polynomial in η. In contrast, in the exact security framework, on which we focus in this report, one computes the probability of success of an attack against the protocol as a function of the probability of breaking the primitives such as $\mathsf{Succ}_{\mathsf{MAC}}^{\mathsf{uf-cma}}(t, q_m, q_v, l)$, of the runtime of functions, of the parameters n, and of the length of messages, thus providing a more precise security result. Intuitively, the probability $\mathsf{Succ}_{\mathsf{MAC}}^{\mathsf{uf-cma}}(t, q_m, q_v, l)$ is assumed to be small (otherwise, the computed probability of attack will be large), but no formal assumption on this probability is needed to establish the security theorem.

Definition 2 Let T_r and T'_r be fixed-length types representing random coins; let T_k and T_e be types for keys and ciphertexts respectively. A symmetric encryption scheme SE [2] consists of three function symbols:

- $\mathrm{kgen} : T_r \to T_k$ is the key generation algorithm taking as argument random coins and returning a key,
- $\mathrm{enc} : \mathit{bitstring} \times T_k \times T'_r \to T_e$ is the encryption algorithm taking as arguments the cleartext, the key, and random coins, and returning the ciphertext,
- $\mathrm{dec} : T_e \times T_k \to \mathit{bitstring}_\perp$ is the decryption algorithm taking as arguments the ciphertext and the key, and returning either the cleartext when decryption succeeds or $\perp$ when decryption fails,

such that $\forall m \in \mathit{bitstring}, \forall r \in T_r, \forall r' \in T'_r$, $\mathrm{dec}(\mathrm{enc}(m, \mathrm{kgen}(r), r'), \mathrm{kgen}(r)) = m$.

Let $LR(x, y, b) = x$ if $b = 0$ and $LR(x, y, b) = y$ if $b = 1$, defined only when x and y are bitstrings of the same length. The advantage of an adversary against indistinguishability under chosen plaintext attacks (IND-CPA) is

$$\mathsf{Succ}_{\mathsf{SE}}^{\mathsf{ind-cpa}}(t, q_e, l) = \max_{\mathcal{A}} 2 \Pr \begin{bmatrix} b \overset{R}{\leftarrow} \{0, 1\}; r \overset{R}{\leftarrow} T_r; k \leftarrow \mathrm{kgen}(r); \\ b' \leftarrow \mathcal{A}^{r' \overset{R}{\leftarrow} T'_r; \mathrm{enc}(LR(.,.,b),k,r')} : b' = b \end{bmatrix} - 1$$

where $\mathcal{A}$ is any probabilistic Turing machine that runs in time at most t and calls $r' \overset{R}{\leftarrow} T'_r; enc(LR(.,.,b),k,r')$ at most q_e times on messages of length at most l.

Given two bitstrings a_0 and a_1 of the same length, the left-right encryption oracle $r' \overset{R}{\leftarrow} T'_r; enc(LR(.,.,b),k,r')$ returns $r' \overset{R}{\leftarrow} T'_r; \mathrm{enc}(LR(a_0,a_1,b),k,r')$, that is, encrypts a_0 when $b = 0$ and a_1 when $b = 1$. $\mathsf{Succ}^{\mathsf{ind-cpa}}_{\mathsf{SE}}(t, q_e, l)$ is the probability that the adversary distinguishes the encryption of the messages a_0 given as first arguments to the left-right encryption oracle from the encryption of the messages a_1 given as second arguments. Intuitively, when the encryption scheme is IND-CPA secure, this probability is small: the ciphertext gives almost no information what the cleartext is (one cannot determine whether it is a_0 or a_1 without having the secret key).

Example 1 Let us consider the following trivial protocol:

$$\begin{aligned} A \to B : e, \mathrm{mac}(e, x_{mk}) \quad & \text{where } e = \mathrm{enc}(x'_k, x_k, x'_r) \\ & \text{and } x'_r, x'_k \text{ are fresh random numbers} \end{aligned}$$

A and B are assumed to share a key x_k for a symmetric encryption scheme and a key x_{mk} for a message authentication code. A creates a fresh key x'_k and sends it encrypted under x_k to B. A MAC is appended to the message, in order to guarantee integrity. In other words, the protocol sends the key x'_k encrypted using an encrypt-then-MAC scheme [2]. The goal of the protocol is that x'_k should be a secret key shared between A and B. This protocol can be modeled in our calculus by the following process Q_0:

$$\begin{aligned} Q_0 = {} & \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in} \\ & \mathsf{new}\ x_{mr} : T_{mr}; \mathsf{let}\ x_{mk} : T_{mk} = \mathrm{mkgen}(x_{mr})\ \mathsf{in}\ \bar{c}\langle\rangle; (Q_A \mid Q_B) \\ Q_A = {} & !^{i \le n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x'_r : T'_r; \\ & \mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x'_r)\ \mathsf{in}\ \overline{c_A[i]}\langle x_m, \mathrm{mac}(x_m, x_{mk})\rangle \\ Q_B = {} & !^{i' \le n} c_B[i'](x'_m, x_{ma}); \mathsf{if}\ \mathrm{verify}(x'_m, x_{mk}, x_{ma})\ \mathsf{then} \\ & \mathsf{let}\ \mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle \end{aligned}$$

When Q_0 receives a message on channel start, it begins execution: it generates the keys x_k and x_{mk} by choosing random coins x_r and $x_{r'}$ and applying the appropriate key generation algorithms. Then it yields control to the adversary, by outputting on channel c. After this output, n copies of processes for A and B are ready to be executed, when the adversary outputs on channels $c_A[i]$ or $c_B[i]$ respectively. In a session that runs as expected, the adversary first sends a message on $c_A[i]$. Then Q_A creates a fresh key x'_k (T_k is assumed to be a fixed-length type), encrypts it under x_k with random coins x'_r, computes the MAC under x_{mk} of the ciphertext, and sends the ciphertext and the MAC on $c_A[i]$. The function $\mathrm{k2b} : T_k \to \mathit{bitstring}$ is the natural injection $\mathrm{k2b}(x) = x$; it is needed only for type conversion. The adversary is then expected to forward this message on $c_B[i]$. When Q_B receives this message, it verifies the MAC, decrypts, and stores the obtained key in x''_k. (The function $\mathrm{i}_\perp : \mathit{bitstring} \to \mathit{bitstring}_\perp$ is the natural injection; it is useful to check that decryption succeeded.) This key x''_k should be secret.

The adversary is responsible for forwarding messages from A to B. It can send messages in unexpected ways in order to mount an attack.

This very small example is sufficient to illustrate the main features of CryptoVerif.

2.3 Type System

We use a type system to check that bitstrings of the proper type are passed to each function and that array indices are used correctly.

To be able to type variable accesses used not under their definition (such accesses are guarded by a find construct), the type-checking algorithm proceeds in two passes. In the first pass, it builds a type environment $\mathcal{E}$, which maps variable names x to types $[1, n_1] \times \ldots \times [1, n_m] \to T$, where the definition of $x[i_1, \ldots, i_m]$ of type T occurs under replications or find that bind $i_1, \ldots, i_m$ with declaration $i_j \leq n_j$. (For instance, the definition of $x[i_1, \ldots, i_m]$ occurs under $!^{i_1 \leq n_1}$, ..., $!^{i_m \leq n_m}$ or it occurs in the condition of $\mathsf{find} u_1 = i_1 \leq n_1, \ldots, u_m = i_m \leq n_m$ under no replication. The type T is the one given in the definition of x in $\mathsf{new}\ x[\widetilde{i}] : T$ or in a pattern $x[\widetilde{i}] : T$ in an assignment, an input, or a get. In the find construct, $\mathsf{find} \ldots x[\widetilde{i}] = i \leq n$, the type T of x is $T = [1, n]$.) The tool checks that all definitions of the same variable x yield the same value of $\mathcal{E}(x)$, so that $\mathcal{E}$ is properly defined.

In the second pass, the process is typechecked in the type environment $\mathcal{E}$ using the rules of Figures 2 and 3. These figures defines four judgments:

- $\mathcal{E} \vdash M : T$ means that the term M has type T in environment $\mathcal{E}$.
- $\mathcal{E} \vdash p : T$ means that the pattern p has type T in environment $\mathcal{E}$.
- $\mathcal{E} \vdash P$ and $\mathcal{E} \vdash Q$ mean that the output process P and the input process Q are well-typed in environment $\mathcal{E}$, respectively.

In $x[M_1, \ldots, M_m]$, $M_1, \ldots, M_m$ must be of the suitable interval type. When $f(M_1, \ldots, M_m)$ is called and $f : T_1 \times \ldots \times T_m \to T$, M_j must be of type T_j, and $f(M_1, \ldots, M_m)$ is then of type T.

The term $\mathsf{new}\ x[\widetilde{i}] : T; N$ is accepted only when T is declared *fixed*, *bounded*, or *nonuniform*. We check that $x[\widetilde{i}]$ is of type T (which is in fact always true when the construction of $\mathcal{E}$ succeeds). N must well-typed, and its type is also the type of $\mathsf{new}\ x[\widetilde{i}] : T; N$.

In $\mathsf{let}\ p = M\ \mathsf{in}\ N\ \mathsf{else}\ N'$, p must have the same type as M, and N and N' must have the same type, which is also the type of $\mathsf{let}\ p = M\ \mathsf{in}\ N\ \mathsf{else}\ N'$. The typing rules for patterns p are found at the bottom of Figure 2. The pattern $x[\widetilde{i}] : T$ has type T, provided $x[\widetilde{i}]$ has type T (which is in fact always true when the construction of $\mathcal{E}$ succeeds). The other typing rules for patterns are straightforward. The particular case $\mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ N$ is typed similarly, except that the else branch is omitted.

In

$$\begin{aligned}&\mathsf{find}[\mathit{unique}?]\ (\textstyle\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}\\ &\quad \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N\end{aligned}$$

the replication indices $i_{j1}, \ldots, i_{jm_j}$ are bound in $M_{j1}, \ldots, M_{jl_j}, M_j$, of types $[1, n_{j1}], \ldots, [1, n_{jm_j}]$ respectively; M_j is of type *bool* for all $j \leq m$; N_j for all $j \leq m$ and N all have the same type, which is also the type of the find term.

In $\mathsf{insert}\ \mathit{Tbl}(M_1, \ldots, M_l); N$, $M_1, \ldots, M_l$ must be of the type declared for the elements of the table Tbl, and the type of N is the type of the insert term.

In $\mathsf{get}\ \mathit{Tbl}(p_1, \ldots, p_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ N\ \mathsf{else}\ N'$, $p_1, \ldots, p_l$ must be of the type declared for the elements of the table Tbl and M must be of type *bool*. The terms N and N' must have the same type, which is also the type of the get term.

In $\mathsf{event}\ e(M_1, \ldots, M_l); N$, $M_1, \ldots, M_l$ must be of the type declared for the arguments of event e, and the type of N is the type of the event term.

The term $\mathsf{event\_abort}\ e$ can have any type (because it aborts the game); the event e must be declared without argument, which we denote by $e : ()$.

The type system for processes requires each subterm to be well-typed. In $!^{i \leq n} Q$, i is of type $[1, n]$ in Q. The processes new, let, find, insert, get, event, and event_abort are typed similarly to the corresponding terms.

We say that an occurrence of a term M in a process Q is of type T when $\mathcal{E} \vdash M : T$ where $\mathcal{E}$ is the type environment of Q extended with $i \mapsto [1, n]$ for each replication $!^{i \leq n}$ above M

Typing rules for terms:

$$\frac{\mathcal{E}(i) = T}{\mathcal{E} \vdash i : T} \tag{TIndex}$$

$$\frac{\mathcal{E}(x) = T_1 \times \ldots \times T_m \to T \qquad \forall j \leq m, \mathcal{E} \vdash M_j : T_j}{\mathcal{E} \vdash x[M_1, \ldots, M_m] : T} \tag{TVar}$$

$$\frac{f : T_1 \times \ldots \times T_m \to T \qquad \forall j \leq m, \mathcal{E} \vdash M_j : T_j}{\mathcal{E} \vdash f(M_1, \ldots, M_m) : T} \tag{TFun}$$

$$\frac{T \textit{ fixed, bounded}, \text{ or } \textit{nonuniform} \qquad \mathcal{E} \vdash x[\widetilde{i}] : T \qquad \mathcal{E} \vdash N : T'}{\mathcal{E} \vdash \mathsf{new}\ x[\widetilde{i}] : T; N : T'} \tag{TNewT}$$

$$\frac{\mathcal{E} \vdash M : T \qquad \mathcal{E} \vdash p : T \qquad \mathcal{E} \vdash N : T' \qquad \mathcal{E} \vdash N' : T'}{\mathcal{E} \vdash \mathsf{let}\ p = M\ \mathsf{in}\ N\ \mathsf{else}\ N' : T'} \tag{TLetT}$$

$$\frac{\mathcal{E} \vdash M : T \qquad \mathcal{E} \vdash x[\widetilde{i}] : T \qquad \mathcal{E} \vdash N : T'}{\mathcal{E} \vdash \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ N : T'} \tag{TLetT2}$$

$$\frac{\begin{array}{c}\forall j \leq m, \forall k \leq m_j, \mathcal{E} \vdash u_{jk}[\widetilde{i}] : [1, n_{jk}] \\ \forall j \leq m, \forall k \leq l_j, \mathcal{E}[i_{j1} \mapsto [1, n_{j1}], \ldots, i_{jm_j} \mapsto [1, n_{jm_j}]] \vdash M_{jk} : T_{jk} \\ \forall j \leq m, \mathcal{E}[i_{j1} \mapsto [1, n_{j1}], \ldots, i_{jm_j} \mapsto [1, n_{jm_j}]] \vdash M_j : \textit{bool} \\ \forall j \leq m, \mathcal{E} \vdash N_j : T \qquad \mathcal{E} \vdash N : T\end{array}}{\begin{array}{c}\mathcal{E} \vdash \mathsf{find}[\textit{unique?}]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N : T\end{array}} \tag{TFindT}$$

$$\frac{\textit{Tbl} : T_1 \times \ldots \times T_l \qquad \forall j \leq l, \mathcal{E} \vdash M_j : T_j \qquad \mathcal{E} \vdash N : T}{\mathcal{E} \vdash \mathsf{insert}\ \textit{Tbl}(M_1, \ldots, M_l); N : T} \tag{TInsertT}$$

$$\frac{\textit{Tbl} : T_1 \times \ldots \times T_l \qquad \forall j \leq l, \mathcal{E} \vdash p_j : T_j \qquad \mathcal{E} \vdash M : \textit{bool} \qquad \mathcal{E} \vdash N : T \qquad \mathcal{E} \vdash N' : T}{\mathsf{get}\ \textit{Tbl}(p_1, \ldots, p_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ N\ \mathsf{else}\ N' : T} \tag{TGetT}$$

$$\frac{e : T_1 \times \ldots \times T_l \qquad \forall j \leq l, \mathcal{E} \vdash M_j : T_j \qquad \mathcal{E} \vdash N : T}{\mathcal{E} \vdash \mathsf{event}\ e(M_1, \ldots, M_l); N : T} \tag{TEvent}$$

$$\frac{e : ()}{\mathcal{E} \vdash \mathsf{event\_abort}\ e : T'} \tag{TEventAbortT}$$

Typing rules for patterns:

$$\frac{\mathcal{E} \vdash x[\widetilde{i}] : T}{\mathcal{E} \vdash (x[\widetilde{i}] : T) : T} \tag{TVarP}$$

$$\frac{f : T_1 \times \ldots \times T_m \to T \qquad \forall j \leq m, \mathcal{E} \vdash p_j : T_j}{\mathcal{E} \vdash f(p_1, \ldots, p_m) : T} \tag{TFunP}$$

$$\frac{\mathcal{E} \vdash M : T}{\mathcal{E} \vdash =M : T} \tag{TEqP}$$

Figure 2: Typing rules (1)

Typing rules for input processes:

$$\mathcal{E} \vdash 0 \tag{TNil}$$

$$\frac{\mathcal{E} \vdash Q \qquad \mathcal{E} \vdash Q'}{\mathcal{E} \vdash Q \mid Q'} \tag{TPar}$$

$$\frac{\mathcal{E}[i \mapsto [1,n]] \vdash Q}{\mathcal{E} \vdash !^{i \leq n} Q} \tag{TRepl}$$

$$\frac{\mathcal{E} \vdash Q}{\mathcal{E} \vdash \mathsf{newChannel}\ c; Q} \tag{TNewChannel}$$

$$\frac{\forall j \leq l, \mathcal{E} \vdash M_j : T'_j \qquad \mathcal{E} \vdash p : T \qquad \mathcal{E} \vdash P}{\mathcal{E} \vdash c[M_1, \dots, M_l](p); P} \tag{TIn}$$

Typing rules for output processes:

$$\frac{\forall j \leq l, \mathcal{E} \vdash M_j : T'_j \qquad \mathcal{E} \vdash N : T \qquad \mathcal{E} \vdash Q}{\mathcal{E} \vdash \overline{c[M_1, \dots, M_l]}\langle N \rangle; Q} \tag{TOut}$$

$$\frac{T \text{ \textit{fixed}, \textit{bounded}, or \textit{nonuniform}} \qquad \mathcal{E} \vdash x[\widetilde{i}] : T \qquad \mathcal{E} \vdash P}{\mathcal{E} \vdash \mathsf{new}\ x[\widetilde{i}] : T; P} \tag{TNew}$$

$$\frac{\mathcal{E} \vdash M : T \qquad \mathcal{E} \vdash p : T \qquad \mathcal{E} \vdash P \qquad \mathcal{E} \vdash P'}{\mathcal{E} \vdash \mathsf{let}\ p = M\ \mathsf{in}\ P\ \mathsf{else}\ P'} \tag{TLet}$$

$$\frac{\begin{array}{c}\forall j \leq m, \forall k \leq m_j, \mathcal{E} \vdash u_{jk}[\widetilde{i}] : [1, n_{jk}] \\ \forall j \leq m, \forall k \leq l_j, \mathcal{E}[i_{j1} \mapsto [1, n_{j1}], \dots, i_{jm_j} \mapsto [1, n_{jm_j}]] \vdash M_{jk} : T_{jk} \\ \forall j \leq m, \mathcal{E}[i_{j1} \mapsto [1, n_{j1}], \dots, i_{jm_j} \mapsto [1, n_{jm_j}]] \vdash M_j : \mathit{bool} \\ \forall j \leq m, \mathcal{E} \vdash P_j \qquad \mathcal{E} \vdash P\end{array}}{\begin{array}{c}\mathcal{E} \vdash \mathsf{find}[\mathit{unique}?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P\end{array}} \tag{TFind}$$

$$\frac{\mathit{Tbl} : T_1 \times \dots \times T_l \qquad \forall j \leq l, \mathcal{E} \vdash M_j : T_j \qquad \mathcal{E} \vdash P}{\mathcal{E} \vdash \mathsf{insert}\ \mathit{Tbl}(M_1, \dots, M_l); P} \tag{TInsert}$$

$$\frac{\mathit{Tbl} : T_1 \times \dots \times T_l \qquad \forall j \leq l, \mathcal{E} \vdash p_j : T_j \qquad \mathcal{E} \vdash M : \mathit{bool} \qquad \mathcal{E} \vdash P \qquad \mathcal{E} \vdash P'}{\mathsf{get}\ \mathit{Tbl}(p_1, \dots, p_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'} \tag{TGet}$$

$$\frac{e : T_1 \times \dots \times T_l \qquad \forall j \leq l, \mathcal{E} \vdash M_j : T_j \qquad \mathcal{E} \vdash P}{\mathcal{E} \vdash \mathsf{event}\ e(M_1, \dots, M_l); P} \tag{TEvent}$$

$$\frac{e : ()}{\mathcal{E} \vdash \mathsf{event\_abort}\ e} \tag{TEventAbort}$$

$$\mathcal{E} \vdash \mathsf{yield} \tag{TYield}$$

Figure 3: Typing rules (2)

in Q and with $i_{j1} \mapsto [1, n_{j1}], \dots, i_{jm_j} \mapsto [1, n_{jm_j}]$ for each $\mathsf{find}[\mathit{unique}?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}$ $\mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$ such that the considered occurrence of M is in the condition $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$.

Invariant 7 (Typing) The process Q_0 satisfies Invariant 7 if and only if the type environment $\mathcal{E}$ for Q_0 is well-defined, and $\mathcal{E} \vdash Q_0$.

We require the adversary to be well-typed. This requirement does not restrict its computing power, because it can always define type-cast functions $f : T \to T'$ to bypass the type system. Similarly, the type system does not restrict the class of protocols that we consider, since the protocol may contain type-cast functions. The type system just makes explicit which set of values may appear at each point of the protocol.

2.4 Formal Semantics

2.4.1 Definition of the Semantics

The formal semantics of our calculus is presented in Figures 4, 5, 6, and 7. A semantic configuration is a sextuple $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$, where

- E is an environment mapping array cells to values.
- (σ, P) is the output process P currently scheduled, with the associated function σ which gives values of replication indices.
- $\mathcal{Q}$ is the multiset of input processes running in parallel with P, with their associated substitutions giving values of replication indices.
- $\mathcal{C}$ is the set of channels already created.
- $\mathcal{T}$ defines the contents of tables. It is a list of $\mathit{Tbl}(a_1, \dots, a_m)$ indicating that table Tbl contains the element $(a_1, \dots, a_m)$.
- $\mathcal{E}v$ is the sequence of events $e(a_1, \dots, a_m)$ executed so far.

There is one exceptional configuration, of the form $\mathsf{abort}, \mathcal{E}v$, corresponding to the situation in which the game has been aborted after executing the events in $\mathcal{E}v$.

The semantics is defined by reduction rules of the form $E, P, \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', P', \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$ meaning that $E, P, \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$ reduces to $E', P', \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$ with probability p. The index t just serves in distinguishing reductions that yield the same configuration with the same probability in different ways, so that the probability of a certain reduction can be computed correctly:

$$\Pr[E, P, \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \to E', P', \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'] = \sum_{E, P, \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', P', \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'} p$$

The probability of a trace $\mathit{Tr} = E_1, P_1, \mathcal{Q}_1, \mathcal{C}_1, \mathcal{T}_1, \mathcal{E}v_1 \xrightarrow{p_1}_{t_1} \dots \xrightarrow{p_{m-1}}_{t_{m-1}} E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m, \mathcal{T}_m, \mathcal{E}v_m$ is $p_1 \times \dots \times p_{m-1}$. We define the semantics only for patterns $x[\widetilde{i}] : T$, the other patterns can be encoded as outlined in Section 2.1.

In Figures 4 and 5, we define an auxiliary relation for evaluating terms: $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma, M', \mathcal{T}', \mathcal{E}v'$ means that the term M reduces to M' in environment E with the replication indices defined by σ, the table contents $\mathcal{T}$, and the sequence of events $\mathcal{E}v$, with probability p. Here, the terms M, M' can be values a in addition to the grammar to terms given in Figure 1. Rule (ReplIndex) evaluates replication indices using the function σ. Rule (Var) looks for the value of the variable in the environment E. Rule (Fun) evaluates the function call. Rule (NewT)

$$E, \sigma, i, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, \sigma, \sigma(i), \mathcal{T}, \mathcal{E}v \tag{ReplIndex}$$

$$\frac{x[a_1, \dots, a_m] \in \mathrm{Dom}(E)}{E, \sigma, x[a_1, \dots, a_m], \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, \sigma, E(x[a_1, \dots, a_m]), \mathcal{T}, \mathcal{E}v} \tag{Var}$$

$$\frac{f : T_1 \times \ldots \times T_m \to T \qquad \forall j \leq m, a_j \in T_j \qquad f(a_1, \dots, a_m) = a}{E, \sigma, f(a_1, \dots, a_m), \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, \sigma, a, \mathcal{T}, \mathcal{E}v} \tag{Fun}$$

$$\frac{a \in T \qquad E' = E[x[\sigma(\widetilde{i})] \mapsto a]}{E, \sigma, \mathsf{new}\ x[\widetilde{i}] : T; N, \mathcal{T}, \mathcal{E}v \xrightarrow{D_T(a)}_{N(a)} E', \sigma, N, \mathcal{T}, \mathcal{E}v} \tag{NewT}$$

$$\frac{a \in T \qquad E' = E[x[\sigma(\widetilde{i})] \mapsto a]}{E, \sigma, \mathsf{let}\ x[\widetilde{i}] : T = a\ \mathsf{in}\ N, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E', \sigma, N, \mathcal{T}, \mathcal{E}v} \tag{LetT}$$

$$\frac{\begin{array}{c}(v_k)_{1 \leq k \leq l} \text{ is the sequence of } (j, a_1, \dots, a_{m_j}) \text{ for } a_1 \in [1, n_{j1}], \dots, a_{m_j} \in [1, n_{jm_j}] \\ \text{ordered in increasing lexicographic order} \\ \exists l_0 \leq l, \forall k \in [1, l_0], E, \sigma[i_{j1} \mapsto a_1, \dots, i_{jm_j} \mapsto a_{m_j}], D_j \wedge M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{p_k}{}^{*}_{t_k} E'', \sigma', r_k, \mathcal{T}, \mathcal{E}v \\ \text{where } v_k = (j, a_1, \dots, a_{m_j}) \text{ and for } k < l_0, r_k \text{ is a value}, r_{l_0} = \mathsf{event\_abort}\ e\end{array}}{\begin{array}{c}E, \sigma, \mathsf{find}[\mathit{unique?}]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ D_j \wedge M_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N, \mathcal{T}, \mathcal{E}v \xrightarrow{p_1 \dots p_{l_0}}_{t_1 \dots t_{l_0}} E, \sigma, \mathsf{event\_abort}\ e, \mathcal{T}, \mathcal{E}v\end{array}} \tag{FindTE}$$

$$\frac{\begin{array}{c}(v_k)_{1 \leq k \leq l} \text{ is the sequence of } (j, a_1, \dots, a_{m_j}) \text{ for } a_1 \in [1, n_{j1}], \dots, a_{m_j} \in [1, n_{jm_j}] \\ \text{ordered in increasing lexicographic order} \\ \forall k \in [1, l], E, \sigma[i_{j1} \mapsto a_1, \dots, i_{jm_j} \mapsto a_{m_j}], D_j \wedge M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{p_k}{}^{*}_{t_k} E'', \sigma', r_k, \mathcal{T}, \mathcal{E}v \\ \text{where } v_k = (j, a_1, \dots, a_{m_j}) \text{ and } r_k \text{ is a value} \\ S = \{v_k \mid r_k = \text{true}\} \qquad |S| = 1 \text{ or } [\mathit{unique?}] \text{ is empty} \\ v_0 = (j', a'_1, \dots, a'_{m_{j'}}) \in S \qquad E' = E[u_{j'1}[\sigma(\widetilde{i})] \mapsto a'_1, \dots, u_{j'm_{j'}}[\sigma(\widetilde{i})] \mapsto a'_{m_{j'}}]\end{array}}{\begin{array}{c}E, \sigma, \mathsf{find}[\mathit{unique?}]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ D_j \wedge M_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N, \mathcal{T}, \mathcal{E}v \xrightarrow{p_1 \dots p_l D_{\mathsf{find}}(S)(v_0)}_{t_1 \dots t_l F1(v_0)} E', \sigma, N_{j'}, \mathcal{T}, \mathcal{E}v\end{array}} \tag{FindT1}$$

$$\frac{\text{First four lines as in (FindT1)} \qquad S = \{v_k \mid r_k = \text{true}\} = \emptyset}{\begin{array}{c}E, \sigma, \mathsf{find}[\mathit{unique?}]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ D_j \wedge M_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N, \mathcal{T}, \mathcal{E}v \xrightarrow{p_1 \dots p_l}_{t_1 \dots t_l F2} E, \sigma, N, \mathcal{T}, \mathcal{E}v\end{array}} \tag{FindT2}$$

$$\frac{\text{First four lines as in (FindT1)} \qquad S = \{v_k \mid r_k = \text{true}\} \qquad |S| > 1}{\begin{array}{c}E, \sigma, \mathsf{find}[\mathsf{unique}_e]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ D_j \wedge M_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N, \mathcal{T}, \mathcal{E}v \xrightarrow{p_1 \dots p_l}_{t_1 \dots t_l F3} E, \sigma, \mathsf{event\_abort}\ e, \mathcal{T}, \mathcal{E}v\end{array}} \tag{FindT3}$$

Figure 4: Semantics (1): terms, first part

$$\frac{E, \sigma, \mathsf{insert}\ \mathit{Tbl}(a_1, \dots, a_l); N, \mathcal{T}, \mathcal{E}v}{E, \sigma, N, (\mathcal{T}, \mathit{Tbl}(a_1, \dots, a_l)), \mathcal{E}v} \tag{InsertT}$$

$$\frac{\begin{array}{c} [v_1, \dots, v_m] = [x \in \mathcal{T} \mid \exists a_1, \dots, \exists a_l, x = \mathit{Tbl}(a_1, \dots, a_l)] \\ \exists m_0 \leq m, \forall k \in [1, l_0], E[x_1[\sigma(\widetilde{i})] \mapsto a_1, \dots, x_l[\sigma(\widetilde{i})] \mapsto a_l], \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p_k}{}^{*}_{t_k} E'', \sigma, r_k, \mathcal{T}, \mathcal{E}v \\ \text{where } v_k = \mathit{Tbl}(a_1, \dots, a_l) \text{ and for } k < m_0, r_k \text{ is a value}, r_{m_0} = \mathsf{event\_abort}\ e \end{array}}{\begin{array}{c} E, \sigma, \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}] : T_1, \dots, x_l[\widetilde{i}] : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ N\ \mathsf{else}\ N', \mathcal{T}, \mathcal{E}v \\ \xrightarrow{p_1 \dots p_{m_0}}_{t_1 \dots t_{m_0}} E, \sigma, \mathsf{event\_abort}\ e, \mathcal{T}, \mathcal{E}v \end{array}} \tag{GetTE}$$

$$\frac{\begin{array}{c} [v_1, \dots, v_m] = [x \in \mathcal{T} \mid \exists a_1, \dots, \exists a_l, x = \mathit{Tbl}(a_1, \dots, a_l)] \\ \forall k \in [1, m], E[x_1[\sigma(\widetilde{i})] \mapsto a_1, \dots, x_l[\sigma(\widetilde{i})] \mapsto a_l], \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p_k}{}^{*}_{t_k} E'', \sigma, r_k, \mathcal{T}, \mathcal{E}v \\ \text{where } v_k = \mathit{Tbl}(a_1, \dots, a_l) \text{ and } r_k \text{ is a value} \\ L = [v_k \mid k \in [1, m], r_k = \text{true}] \\ \mathit{Tbl}(a_1, \dots, a_l) = \mathsf{nth}(L, j) \qquad E' = E[x_1[\sigma(\widetilde{i})] \mapsto a_1, \dots, x_l[\sigma(\widetilde{i})] \mapsto a_l] \end{array}}{\begin{array}{c} E, \sigma, \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}] : T_1, \dots, x_l[\widetilde{i}] : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ N\ \mathsf{else}\ N', \mathcal{T}, \mathcal{E}v \\ \xrightarrow{p_1 \dots p_m D_{\mathsf{get}}(\{1, \dots, |L|\})(j)}_{t_1 \dots t_m G1(j)} E', \sigma, N, \mathcal{T}, \mathcal{E}v \end{array}} \tag{GetT1}$$

$$\frac{\text{First four lines as in (GetT1)} \qquad L = \emptyset}{\begin{array}{c} E, \sigma, \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}] : T_1, \dots, x_l[\widetilde{i}] : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ N\ \mathsf{else}\ N', \mathcal{T}, \mathcal{E}v \\ \xrightarrow{p_1 \dots p_m}_{t_1 \dots t_m G2} E, \sigma, N', \mathcal{T}, \mathcal{E}v \end{array}} \tag{GetT2}$$

$$E, \sigma, \mathsf{event}\ e(a_1, \dots, a_l); N, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, \sigma, N, \mathcal{T}, (\mathcal{E}v, e(a_1, \dots, a_l)) \tag{EventT}$$

$$\frac{E, \sigma, N, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', N', \mathcal{T}', \mathcal{E}v'}{E, \sigma, C[N], \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E, \sigma', C[N'], \mathcal{T}', \mathcal{E}v'} \tag{CtxT}$$

$$E, \sigma, C[\mathsf{event\_abort}\ e], \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, \sigma, \mathsf{event\_abort}\ e, \mathcal{T}, \mathcal{E}v \tag{CtxEventT}$$

$$\frac{\neg \forall j \leq l, \exists a_j, E, \sigma, M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^{*} E, \sigma, a_j, \mathcal{T}, \mathcal{E}v}{E, \sigma, \mathsf{defined}(M_1, \dots, M_l) \wedge M, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, \sigma, \text{false}, \mathcal{T}, \mathcal{E}v} \tag{DefinedNo}$$

$$\frac{\forall j \leq l, \exists a_j, E, \sigma, M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^{*} E, \sigma, a_j, \mathcal{T}, \mathcal{E}v}{E, \sigma, \mathsf{defined}(M_1, \dots, M_l) \wedge M, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, \sigma, M, \mathcal{T}, \mathcal{E}v} \tag{DefinedYes}$$

Figure 5: Semantics (2): terms, second part, and defined conditions

chooses a random $a \in T$ according to distribution D_T, and stores it in $x[\sigma(\widetilde{i})]$ by extending the environment E accordingly. Similarly, Rule (LetT) extends the environment E with the value of $x[\sigma(\widetilde{i})]$.

Rules (FindTE) to (FindT3) define the semantics of find. First, they all evaluate the conditions for all branches j and all values of the indices $i_{j1}, \dots, i_{jm_j}$. If one of these evaluations executes an event (which can happen in case the condition contains a $\mathsf{find}[\mathsf{unique}_e]$), the whole find executes the same event (Rule (FindTE)). Otherwise, the branch and indices for which the condition is true are collected in a set S. If S is empty, the else branch of the find is executed (Rule (FindT2)). When S is not empty, two cases can happen. Either the find is not marked $[\mathsf{unique}_e]$, and we choose an element $v_0 = (j', a'_1, \dots, a'_{m_{j'}}$ of S randomly according to the distribution $D_{\mathsf{find}}(S)$, store the corresponding indices $a'_1, \dots, a'_{m_{j'}}$ in $u_{j'1}[\sigma(\widetilde{i})], \dots, u_{j'm_{j'}}[\sigma(\widetilde{i})]$ by extending the environment accordingly, and we continue with the selected branch N'_j. If the find is marked $[\mathsf{unique}_e]$ and S has a single element, we do the same. If the find is marked $[\mathsf{unique}_e]$ and S has several elements, we execute the event e (Rule (FindT3)). We recall that $D_{\mathsf{find}}(S)(v_0)$ denotes the probability of choosing v_0 in the distribution $D_{\mathsf{find}}(S)$. The terms in conditions of find may define variables, included in the envronment E''; we ignore these additional variables and compute the final environment from the initial environment E, because these variables have no array accesses by Invariant 3, so the values of these variables are not used after the evaluation of the condition. The conditions of find, $D \wedge M$, are evaluated using Rules (DefinedNo) and (DefinedYes). An element of the $\mathsf{defined}$ condition D is not defined, the condition is false (Rule (DefinedNo)); when all elements of the $\mathsf{defined}$ condition are defined, we evaluate M (Rule (DefinedYes)). Since terms in conditions of find do not contain insert nor event (Invariant 4), the table contents and the sequence of events are left unchanged by the evaluation of the condition of find.

Rule (InsertT) inserts the new table element in $\mathcal{T}$. Rules (GetTE), (GetT1) and (GetT2) define the semantics of get. We denote by $[x \in L \mid f(x)]$ the list of all elements x of the list L that satisfy $f(x)$, in the same order as in L. We denote by $|L|$ the length of list L. We denote by $\mathsf{nth}(L, j)$ the j-th element of the list L. Rule (GetTE) executes $\mathsf{event\_abort}\ e$ when the evaluation of the condition M executes $\mathsf{event\_abort}\ e$ for some element of the table Tbl. Rules (Get1) and (Get2) compute the list L of elements of $\mathcal{T}$ that are in table Tbl and satisfy condition M. When L is not empty, one of its elements is chosen randomly according to distribution $D_{\mathsf{get}}(\{1, \dots, |L|\})$, we store this element in $x_1[\sigma(\widetilde{i})], \dots, x_l[\sigma(\widetilde{i})]$ by extending the environment E, and continue by executing N (Rule (Get1)). If L is empty, we execute N' (Rule (Get2)). Since terms in conditions of get do not contain insert nor event (Invariant 4), the table contents and the sequence of events are left unchanged by the evaluation of M. The modified environment E'' obtained after evaluating a condition M can be ignored because there are no array accesses to the variables defined in conditions of get, by Invariant 3, so the values of these variables are not used after the evaluation of the condition.

Remark 1 Another way of defining the semantics of tables would be to consider two distinct calculi, one with tables but without find and $\mathsf{defined}$ conditions (used for the initial game), and one with find and $\mathsf{defined}$ conditions but without tables (used for the other games). The semantics of the calculus with find and $\mathsf{defined}$ conditions but without tables can be defined without the component $\mathcal{T}$. The semantics of the calculus with tables but without find and $\mathsf{defined}$ conditions can be defined with one environment for each process, instead of a global environment, and no arrays (only the value at the current replication indices is stored). We then need to relate the two semantics.

Rule EventT adds the executed event to $\mathcal{E}v$.

Rules (CtxT) and (CtxEventT) allow evaluating terms under a context. In these rules, C

$$E, \{(\sigma, 0)\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \mathcal{Q}, \mathcal{C} \tag{Nil}$$

$$E, \{(\sigma, Q_1 \mid Q_2)\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \{(\sigma, Q_1), (\sigma, Q_2)\} \uplus \mathcal{Q}, \mathcal{C} \tag{Par}$$

$$E, \{(\sigma, !^{i \leq n} Q)\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \{(\sigma[i \mapsto a], Q) \mid a \in [1, n]\} \uplus \mathcal{Q}, \mathcal{C} \tag{Repl}$$

$$\frac{c' \notin \mathcal{C}}{E, \{(\sigma, \mathsf{newChannel}\ c; Q)\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \{(\sigma, Q\{c'/c\})\} \uplus \mathcal{Q}, \mathcal{C} \cup \{c'\}} \tag{NewChannel}$$

$$\frac{\forall j \leq l, E, \sigma, M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^{*} E, \sigma, a_j, \mathcal{T}, \mathcal{E}v}{E, \{(\sigma, c[M_1, \dots, M_l](x[\widetilde{i}] : T); P)\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \{(\sigma, c[a_1, \dots, a_l](x[\widetilde{i}] : T); P)\} \uplus \mathcal{Q}, \mathcal{C}} \tag{Input}$$

$$\text{reduce}(E, \mathcal{Q}, \mathcal{C}) \text{ is the normal form of } E, \mathcal{Q}, \mathcal{C} \text{ by } \rightsquigarrow$$

Figure 6: Semantics (3): input processes

is an elementary context, of the form:

$$\begin{aligned}
C ::= {} & x[a_1, \dots, a_k, [\,], M_{k+2}, \dots, M_m] \\
& f(a_1, \dots, a_k, [\,], M_{k+2}, \dots, M_m) \\
& \mathsf{let}\ x[\widetilde{i}] : T = [\,]\ \mathsf{in}\ N \\
& \mathsf{event}\ e(a_1, \dots, a_k, [\,], M_{k+2}, \dots, M_l); N \\
& \mathsf{insert}\ \mathit{Tbl}(a_1, \dots, a_k, [\,], M_{k+2}, \dots, M_l); N
\end{aligned}$$

When the term N reduces to some other term N', Rule (CtxT) allows one to reduce it in the same way under a context C. When the term N is an event, $C[N]$ also executes the same event by Rule (CtxEventT).

These rules define a small-step semantics for terms. We consider the reflexive and transitive closure $\xrightarrow{p}{}_t^{*}$ of the relation $\xrightarrow{p}_t$ to reach directly the normal form of the term, which can be either a value a or an event $\mathsf{event\_abort}\ e$. We have $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^{*} E, \sigma, M, \mathcal{T}, \mathcal{E}v$ and, if $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$ and $E', \sigma', M', \mathcal{T}', \mathcal{E}v' \xrightarrow{p'}{}_{t'}^{*} E'', \sigma'', M'', \mathcal{T}'', \mathcal{E}v''$, then $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p \times p'}{}_{t,t'}^{*} E'', \sigma'', M'', \mathcal{T}'', \mathcal{E}v''$: we take the product of the probabilities to have the probability of a sequence of reductions, and we specify which sequence was taken by a list of indices t, t'.

Figure 6 defines the semantics of input processes. We use an auxiliary reduction relation $\rightsquigarrow$, for reducing input processes. This relation transforms configurations of the form $E, \mathcal{Q}, \mathcal{C}$. Rule (Nil) removes nil processes. Rules (Par) and (Repl) expand parallel compositions and replications, respectively. Rule (NewChannel) creates a new channel and adds it to $\mathcal{C}$. Semantic configurations are considered equivalent modulo renaming of channels in $\mathcal{C}$, so that a single semantic configuration is obtained after applying (NewChannel). Rule (Input) evaluates the terms in the input channel. The input itself is not executed: the communication is done by the (Output) rule. In the (Input) rule, the terms $M_1, \dots, M_l$ contain only replication indices, variables, and function applications by Invariant 5, so their evaluation is deterministic (the unique result is obtained with probability 1), and the environment E, the contents of tables $\mathcal{T}$, and the sequence of events $\mathcal{E}v$ are unchanged, that is why we can write $E, \sigma, M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^{*} E, \sigma, a_j, \mathcal{T}, \mathcal{E}v$. The relation $\rightsquigarrow$ is convergent (confluent and terminating), so it has normal forms. Processes in $\mathcal{Q}$ in configurations $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$ are always in normal form by $\rightsquigarrow$, so they always start with an input.

Finally, Figure 7 defines the semantics of output processes. The rules (New) to (Event) are very similar to those for terms: they just use processes instead of terms as continuations, and include a whole semantic configuration. Rule (Output) performs communications: it selects an

$$\frac{a \in T \qquad E' = E[x[\sigma(\widetilde{i})] \mapsto a]}{E, (\sigma, \mathsf{new}\ x[\widetilde{i}] : T; P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{D_T(a)}_{N(a)} E', (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v} \qquad \text{(New)}$$

$$\frac{a \in T \qquad E' = E[x[\sigma(\widetilde{i})] \mapsto a]}{E, (\sigma, \mathsf{let}\ x[\widetilde{i}] : T = a\ \mathsf{in}\ P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E', (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v} \qquad \text{(Let)}$$

$$\frac{\text{Same assumption as in (FindTE)}}{\begin{array}{c} E, (\sigma, \mathsf{find}[\mathit{unique?}]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ D_j \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p_1 \dots p_{l_0}}_{t_1 \dots t_{l_0}} \mathsf{abort}, (\mathcal{E}v, e) \end{array}} \qquad \text{(FindE)}$$

$$\frac{\begin{array}{c} \text{First four lines as in (FindT1)} \qquad S = \{v_k \mid r_k = \text{true}\} \qquad |S| = 1 \text{ or } [\mathit{unique?}] \text{ is empty} \\ v_0 = (j', a'_1, \dots, a'_{m_{j'}}) \in S \qquad E' = E[u_{j'1}[\sigma(\widetilde{i})] \mapsto a'_1, \dots, u_{j'm_{j'}}[\sigma(\widetilde{i})] \mapsto a'_{m_{j'}}] \end{array}}{\begin{array}{c} E, (\sigma, \mathsf{find}[\mathit{unique?}]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ D_j \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p_1 \dots p_l D_{\mathsf{find}}(S)(v_0)}_{t_1 \dots t_l F1(v_0)} E', (\sigma, P_{j'}), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \end{array}}$$

$$\text{(Find1)}$$

$$\frac{\text{First four lines as in (FindT1)} \qquad S = \{v_k \mid r_k = \text{true}\} = \emptyset}{\begin{array}{c} E, (\sigma, \mathsf{find}[\mathit{unique?}]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ D_j \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p_1 \dots p_l}_{t_1 \dots t_l F2} E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \end{array}} \qquad \text{(Find2)}$$

$$\frac{\text{First four lines as in (FindT1)} \qquad S = \{v_k \mid r_k = \text{true}\} \qquad |S| > 1}{\begin{array}{c} E, (\sigma, \mathsf{find}[\mathsf{unique}_e]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat} \\ D_j \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p_1 \dots p_l}_{t_1 \dots t_l F3} \mathsf{abort}, (\mathcal{E}v, e) \end{array}} \qquad \text{(Find3)}$$

$$\frac{\begin{array}{c} E, \mathcal{Q}', \mathcal{C}' = \text{reduce}(E, \{(\sigma, Q'')\}, \mathcal{C}) \\ S = \{(\sigma', Q) \in \mathcal{Q} \mid Q = c[a_1, \dots, a_l](x'[\widetilde{i}] : T').P' \text{ and } b \in T' \text{ for some } x', \widetilde{a''}, T', P'\} \\ (\sigma', Q_0) \in S \qquad Q_0 = c[a_1, \dots, a_l](x[\widetilde{i}] : T).P \end{array}}{\begin{array}{c} E, (\sigma, \overline{c[a_1, \dots, a_l]}\langle b \rangle.Q''), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{S(Q_0) \times D_{\mathsf{in}}(S)(Q_0)}_{O(Q_0)} \\ E[x[\sigma'(\widetilde{i})] \mapsto b], P, \mathcal{Q} \uplus \mathcal{Q}' \setminus \{(\sigma', Q_0)\}, \mathcal{C}', \mathcal{T}, \mathcal{E}v \end{array}} \qquad \text{(Output)}$$

$$E, (\sigma, \mathsf{insert}\ \mathit{Tbl}(a_1, \dots, a_l); P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, (\sigma, P), \mathcal{Q}, \mathcal{C}, (\mathcal{T}, \mathit{Tbl}(a_1, \dots, a_l)), \mathcal{E}v \qquad \text{(Insert)}$$

$$\frac{\text{Same assumption as in (GetTE)}}{\begin{array}{c} E, (\sigma, \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}] : T_1, \dots, x_l[\widetilde{i}] : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \\ \xrightarrow{p_1 \dots p_{m_0}}_{t_1 \dots t_{m_0}} \mathsf{abort}, (\mathcal{E}v, e) \end{array}} \qquad \text{(GetE)}$$

$$\frac{\begin{array}{c} \text{First four lines as in (GetT1)} \\ \mathit{Tbl}(a_1, \dots, a_l) = \mathsf{nth}(L, j) \qquad E' = E[x_1[\sigma(\widetilde{i})] \mapsto a_1, \dots, x_l[\sigma(\widetilde{i})] \mapsto a_l] \end{array}}{\begin{array}{c} E, (\sigma, \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}] : T_1, \dots, x_l[\widetilde{i}] : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \\ \xrightarrow{p_1 \dots p_m D_{\mathsf{get}}(\{1, \dots, |L|\})(j)}_{t_1 \dots t_m G1(j)} E', (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \end{array}} \qquad \text{(Get1)}$$

$$\frac{\text{First four lines as in (GetT1)} \qquad L = \emptyset}{\begin{array}{c} E, (\sigma, \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}] : T_1, \dots, x_l[\widetilde{i}] : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \\ \xrightarrow{p_1 \dots p_m}_{t_1 \dots t_m G2} E, (\sigma, P'), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \end{array}} \qquad \text{(Get2)}$$

$$E, (\sigma, \mathsf{event}\ e(a_1, \dots, a_l); P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, (\mathcal{E}v, e(a_1, \dots, a_l)) \qquad \text{(Event)}$$

$$E, (\sigma, \mathsf{event\_abort}\ e), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{1} \mathsf{abort}, (\mathcal{E}v, e) \qquad \text{(EventAbort)}$$

$$\frac{E, \sigma, N, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', N', \mathcal{T}', \mathcal{E}v'}{E, (\sigma, C[N]), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E, (\sigma', C[N']), \mathcal{Q}, \mathcal{C}, \mathcal{T}', \mathcal{E}v'} \qquad \text{(Ctx)}$$

$$E, (\sigma, C[\mathsf{event\_abort}\ e]), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{1} \mathsf{abort}, (\mathcal{E}v, e) \qquad \text{(CtxEvent)}$$

Figure 7: Semantics (4): output processes

input on the desired channel randomly, and immediately executes the communication. (The process blocks if no suitable input is available.) The scheduled process after this rule is the receiving process. The input processes that follow the output are stored in the available input processes, after reducing them by rules of Figure 6. In this rule, S is a multiset. When we take probabilities over multisets, we consider that $D_{\mathsf{in}}(S)(Q_0)$ is the probability of choosing *one* of the elements equal to Q_0 in S according to the distribution $D_{\mathsf{in}}(S)$, so that the probability of choosing any element equal to Q_0 is in fact $S(Q_0) \times D_{\mathsf{in}}(S)(Q_0)$.

Rule EventAbort executes event e and aborts the game, by reducing to the special configuration $\mathsf{abort}, (\mathcal{E}v, e)$.

Similarly to the case of terms, Rules (Ctx) and (CtxEvent) allow evaluating terms under a context inside a process. In these rules, C is an elementary context, of the form:

$$\begin{aligned} C ::= &\ \mathsf{let}\ x[\widetilde{i}] : T = [\,]\ \mathsf{in}\ P \\ &\ \overline{c[a_1, \dots, a_k, [\,], M_{k+2}, \dots, M_l]}\langle N \rangle.Q \\ &\ \overline{c[a_1, \dots, a_l]}\langle [\,] \rangle.Q \\ &\ \mathsf{insert}\ \mathit{Tbl}(a_1, \dots, a_k, [\,], M_{k+2}, \dots, M_l); P \\ &\ \mathsf{event}\ e(a_1, \dots, a_k, [\,], M_{k+2}, \dots, M_l); P \end{aligned}$$

After finishing execution of a process, the system produces two results: the sequence of executed events $\mathcal{E}v$, and the information whether we aborted the game ($a = \mathsf{abort}$) or it terminated normally ($a = 0$). These events and result can be used to distinguish games, so we introduce an additional algorithm, a *distinguisher* D that takes as input the sequence of events $\mathcal{E}v$ and the result a, and returns true or false.

An example of distinguisher is D_e defined by $D_e(\mathcal{E}v, a) = \text{true}$ if and only if $e \in \mathcal{E}v$: this distinguisher detects the execution of event e. We will denote the distinguisher D_e simply by e. More generally, distinguishers can detect various properties of the sequence of events $\mathcal{E}v$ executed by the game and of its result a. We denote by $D \vee D'$, $D \wedge D'$, and $\neg D$ the distinguishers such that $(D \vee D')(\mathcal{E}v, a) = D(\mathcal{E}v, a) \vee D'(\mathcal{E}v, a)$, $(D \wedge D')(\mathcal{E}v, a) = D(\mathcal{E}v, a) \wedge D'(\mathcal{E}v, a)$, and $(\neg D)(\mathcal{E}v, a) = \neg D(\mathcal{E}v, a)$. We denote by $\Pr[Q : D]$ the probability that Q executes a sequence of events $\mathcal{E}v$ and returns a result a, such that $D(\mathcal{E}v, a) = \text{true}$. This is formally defined as follows.

Definition 3 The initial configuration for running process Q is $\text{initConfig}(Q) = \emptyset, \overline{start}\langle\rangle, \mathcal{Q}, \mathcal{C}, \emptyset, \emptyset$ where $\emptyset, \mathcal{Q}, \mathcal{C} = \text{reduce}(\emptyset, \{(\sigma_0, Q)\}, \text{fc}(Q))$ and σ_0 is the empty function.

Let $\mathcal{T}r$ be the set of traces $\mathit{Tr} = \text{initConfig}(Q) \xrightarrow{p_1}_{t_1} \dots \xrightarrow{p_{m-1}}_{t_{m-1}} \mathit{Conf}_m$ such that Conf_m cannot be reduced.

For such traces, we define $\Pr[\mathit{Tr}] = p_1 \times \dots \times p_m$, and $D(\mathit{Tr}) = D(\mathcal{E}v_m, \mathsf{abort})$ if $\mathit{Conf}_m = \mathsf{abort}, \mathcal{E}v_m$ and $D(\mathit{Tr}) = D(\mathcal{E}v_m, 0)$ if $\mathit{Conf}_m = E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m, \mathcal{T}_m, \mathcal{E}v_m$.

We have $\Pr[Q : D] = \sum_{\mathit{Tr} \in \mathcal{T}r, D(\mathit{Tr}) = \text{true}} \Pr[\mathit{Tr}]$.

2.4.2 Each Variable is Defined at Most Once

In this section, we show that Invariant 1 implies that each array cell is assigned at most once during the execution of a process.

When S and S' are multisets, $\max(S, S')$ is the multiset such that $\max(S, S')(x) = \max(S(x), S'(x))$. We define the multiset of variable accesses that may be defined by a term or a process (given the replication indices fixed by a function σ) as follows:

$$\mathit{Defined}(\sigma, i) = \mathit{Defined}(\sigma, a) = \emptyset$$

$$\mathit{Defined}(\sigma, x[M_1, \dots, M_m]) = \biguplus_{j=1}^{m} \mathit{Defined}(\sigma, M_j)$$

$$\mathit{Defined}(\sigma, f(M_1,\dots,M_m)) = \biguplus_{j=1}^{m} \mathit{Defined}(\sigma, M_j)$$

$\mathit{Defined}(\sigma, \mathsf{new}\ x[\widetilde{i}] : T; N) = \{x[\sigma(\widetilde{i})]\} \uplus \mathit{Defined}(\sigma, N)$

$\mathit{Defined}(\sigma, \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ N) = \{x[\sigma(\widetilde{i})]\} \uplus \mathit{Defined}(\sigma, M) \uplus \mathit{Defined}(\sigma, N)$

$\mathit{Defined}(\sigma, \mathsf{find}\ (\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{i}] = \widetilde{i_j} \le \widetilde{n_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1},\dots,M_{jl_j}) \wedge M_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N) =$
$\quad \max(\max_{j=1}^{m}\{\widetilde{u_j}[\sigma(\widetilde{i})]\} \uplus \mathit{Defined}(\sigma, N_j), \mathit{Defined}(\sigma, N))$

$$\mathit{Defined}(\sigma, \mathsf{insert}\ \mathit{Tbl}(M_1,\dots,M_l); N) = \biguplus_{j=1}^{l} \mathit{Defined}(\sigma, M_j) \uplus \mathit{Defined}(\sigma, N)$$

$\mathit{Defined}(\sigma, \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}] : T_1,\dots,x_l[\widetilde{i}] : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ N\ \mathsf{else}\ N') =$
$\quad \max(\{x_j[\sigma(\widetilde{i})] \mid j \le l\} \uplus \mathit{Defined}(\sigma, N), \mathit{Defined}(\sigma, N'))$

$$\mathit{Defined}(\sigma, \mathsf{event}\ e(M_1,\dots,M_l); N) = \biguplus_{j=1}^{l} \mathit{Defined}(\sigma, M_j) \uplus \mathit{Defined}(\sigma, N)$$

$\mathit{Defined}(\sigma, \mathsf{event\_abort}\ e) = \emptyset$

$\mathit{Defined}(\sigma, 0) = \emptyset$

$\mathit{Defined}(\sigma, Q_1 \mid Q_2) = \mathit{Defined}(\sigma, Q_1) \uplus \mathit{Defined}(\sigma, Q_2)$

$$\mathit{Defined}(\sigma, !^{i \le n} Q) = \biguplus_{a \in [1,n]} \mathit{Defined}(\sigma[i \mapsto a], Q)$$

$\mathit{Defined}(\sigma, \mathsf{newChannel}\ c; Q) = \mathit{Defined}(\sigma, Q)$

$\mathit{Defined}(\sigma, c[M_1,\dots,M_l](x[\widetilde{i}] : T); P) = \{x[\sigma(\widetilde{i})]\} \uplus \mathit{Defined}(\sigma, P)$

$$\mathit{Defined}(\sigma, \overline{c[M_1,\dots,M_l]}\langle N\rangle; Q) = \biguplus_{j=1}^{l} \mathit{Defined}(\sigma, M_j) \uplus \mathit{Defined}(\sigma, N) \uplus \mathit{Defined}(\sigma, Q)$$

$\mathit{Defined}(\sigma, \mathsf{new}\ x[\widetilde{i}] : T; P) = \{x[\sigma(\widetilde{i})]\} \uplus \mathit{Defined}(\sigma, P)$

$\mathit{Defined}(\sigma, \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ P) = \{x[\sigma(\widetilde{i})]\} \uplus \mathit{Defined}(\sigma, M) \uplus \mathit{Defined}(\sigma, P)$

$\mathit{Defined}(\sigma, \mathsf{find}\ (\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{i}] = \widetilde{i_j} \le \widetilde{n_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1},\dots,M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P) =$
$\quad \max(\max_{j=1}^{m}\{\widetilde{u_j}[\sigma(\widetilde{i})]\} \uplus \mathit{Defined}(\sigma, P_j), \mathit{Defined}(\sigma, P))$

$$\mathit{Defined}(\sigma, \mathsf{insert}\ \mathit{Tbl}(M_1,\dots,M_l); P) = \biguplus_{j=1}^{l} \mathit{Defined}(\sigma, M_j) \uplus \mathit{Defined}(\sigma, P)$$

$\mathit{Defined}(\sigma, \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}] : T_1,\dots,x_l[\widetilde{i}] : T_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P') =$
$\quad \max(\{x_j[\sigma(\widetilde{i})] \mid j \le l\} \uplus \mathit{Defined}(\sigma, P), \mathit{Defined}(\sigma, P'))$

$$\mathit{Defined}(\sigma, \mathsf{event}\ e(M_1,\dots,M_l); P) = \biguplus_{j=1}^{l} \mathit{Defined}(\sigma, M_j) \uplus \mathit{Defined}(\sigma, P)$$

$\mathit{Defined}(\sigma, \mathsf{event\_abort}\ e) = \emptyset$

Notice that, by Invariant 5, the terms M_j in channels of inputs and the terms M_{jk} in defined conditions of find do not define any variable. By Invariant 3, the variables defined in conditions of find and get can be ignored. We define $\mathit{Defined}(E) = \mathrm{Dom}(E)$, $\mathit{Defined}(E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v) = \mathit{Defined}(E) \uplus \mathit{Defined}(\sigma, P) \uplus \biguplus_{(\sigma,Q)\in\mathcal{Q}} \mathit{Defined}(\sigma, Q)$.

Invariant 8 (Single definition, for executing games) The semantic configuration $E, (\sigma,$

$P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$ satisfies Invariant 8 if and only if $\mathit{Defined}(E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v)$ does not contain duplicate elements.

Lemma 1 *If Q_0 satisfies Invariant 1, then* $\mathrm{initConfig}(Q_0)$ *satisfies Invariant 8.*

Lemma 2 *If $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', P', \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$ and $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$ satisfies Invariant 8, then so does $E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$.*

Proof sketch We show by induction following the definition of $\xrightarrow{p}_t$ that

- if $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$ then $\mathit{Defined}(E) \uplus \mathit{Defined}(\sigma, M) \supseteq \mathit{Defined}(E') \uplus \mathit{Defined}(\sigma', M')$.
- if $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$ then $\mathit{Defined}(E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v) \supseteq \mathit{Defined}(E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v')$.

The result follows. □

Therefore, if Q_0 satisfies Invariant 1, then each variable is defined at most once for each value of its array indices in a trace of Q_0. Indeed, by Invariant 8, just before executing a definition of $x[\widetilde{a}]$, $\mathit{Defined}(E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v)$ does not contain duplicate elements, so $x[\widetilde{a}] \notin \mathrm{Dom}(E)$ since $x[\widetilde{a}] \in \mathit{Defined}(\sigma, P)$.

2.4.3 Variables are Defined Before Being Used

In this section, we show that Invariant 2 implies that all variables are defined before being used. In order to show this property, we use the following invariant:

Invariant 9 (Defined variables, for executing games) The semantic configuration $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$ satisfies Invariant 9 if and only if every occurrence of a variable access $x[M_1, \dots, M_m]$ in (σ, P) or $\mathcal{Q}$ is either

1. present in $\mathrm{Dom}(E)$: if $x[M_1, \dots, M_m]$ occurs in a process P' for $(\sigma', P') \in \{(\sigma, P)\} \cup \mathcal{Q}$, then for all $j \leq m$, $E, \sigma', M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma', a_j, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$;
2. or syntactically under the definition of $x[M_1, \dots, M_m]$ (in which case for all $j \leq m$, M_j is a constant or variable replication index);
3. or in a $\mathsf{defined}$ condition in a find process or term;
4. or in M'_j in a process or term of the form $\mathsf{find}\ (\bigoplus_{j=1}^{m''} \widetilde{u_j}[\widetilde{i}] = \widetilde{i_j} \leq \widetilde{n_j}\ \mathsf{suchthat}\ \mathsf{defined}(M'_{j1}, \dots, M'_{jl_j}) \wedge M'_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$ where for some $k \leq l_j$, $x[M_1, \dots, M_m]$ is a subterm of M'_{jk}.
5. or in P_j in a process or term of the form $\mathsf{find}\ (\bigoplus_{j=1}^{m''} \widetilde{u_j}[\widetilde{i}] = \widetilde{i_j} \leq \widetilde{n_j}\ \mathsf{suchthat}\ \mathsf{defined}(M'_{j1}, \dots, M'_{jl_j}) \wedge M'_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$ where for some $k \leq l_j$, there is a subterm N of M'_{jk} such that $N\{\widetilde{u_j}[\widetilde{i}]/\widetilde{i_j}\} = x[M_1, \dots, M_m]$.

Similarly, $E, \sigma, M, \mathcal{T}, \mathcal{E}v$ satisfies Invariant 9 if and only if every occurrence of a variable access $x[M_1, \dots, M_m]$ in M either is present in $\mathrm{Dom}(E)$ (for all $j \leq m$, $E, \sigma, M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma, a_j, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$) or satisfies one of the last four conditions above.

$E, \sigma, \mathsf{defined}(M'_1, \dots, M'_l) \wedge M, \mathcal{T}, \mathcal{E}v$ satisfies Invariant 9 if and only if every occurrence of a variable access $x[M_1, \dots, M_m]$ in M either is a subterm of $M'_1, \dots, M'_l$, or is present in $\mathrm{Dom}(E)$ (for all $j \leq m$, $E, \sigma, M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma, a_j, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$) or satisfies one of the last four conditions above.

Recall that, by Invariants 2 and 5, the terms of all variable accesses $x[M_1, \dots, M_m]$ contain only replication indices, variables, and function applications. That is why we can evaluate them by $E, \sigma', M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma', a_j, \mathcal{T}, \mathcal{E}v$.

Lemma 3 *If Q_0 satisfies Invariant 2, then* $\mathrm{initConfig}(Q_0)$ *satisfies Invariant 9.*

Lemma 4 *Let M be a term that contains only replication indices, variables, and functions. If $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma, a, \mathcal{T}, \mathcal{E}v$, then for all subterms $x[M_1, \dots, M_m]$ of M, for all $j' \leq m$, $E, \sigma, M_{j'}, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma, a_{j'}, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m]$ is in* $\mathrm{Dom}(E)$.

Proof sketch By induction on M. □

Lemma 5 *Let N, M be terms that contain only replication indices, variables, and functions. If $E, \sigma[i \mapsto a'], N, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma[i \mapsto a'], a, \mathcal{T}, \mathcal{E}v$ and $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma, a', \mathcal{T}, \mathcal{E}v$, then $E, \sigma, N\{M/i\}, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma, a, \mathcal{T}, \mathcal{E}v$.*

Proof sketch By induction on N. □

Lemma 6 *If $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$ and $E, \sigma, M, \mathcal{T}, \mathcal{E}v$ satisfies Invariant 9, then so does $E', \sigma', M', \mathcal{T}', \mathcal{E}v'$.*

If $E, \sigma, \mathsf{defined}(M_1, \dots, M_m) \wedge M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$ and $E, \sigma, \mathsf{defined}(M_1, \dots,$ $M_m) \wedge M, \mathcal{T}, \mathcal{E}v$ satisfies Invariant 9, then so does $E', \sigma', M', \mathcal{T}', \mathcal{E}v'$.

If $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$ and $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$ satisfies Invariant 9, then so does $E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$.

Moreover, if the rules that define $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$ (resp. $E, \sigma, \mathsf{defined}(M_1,$ $\dots, M_m) \wedge M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$ or $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$) require as assumption $E'', \sigma'', M'', \mathcal{T}'', \mathcal{E}v'' \xrightarrow{p}_t \dots$ or $E'', \sigma'', \mathsf{defined}(M''_1, \dots, M''_m) \wedge M'', \mathcal{T}'',$ $\mathcal{E}v'' \xrightarrow{p}_t \dots$, and the initial configuration $E, \sigma, M, \mathcal{T}, \mathcal{E}v$ (resp. $E, \sigma, \mathsf{defined}(M_1, \dots, M_m) \wedge$ $M, \mathcal{T}, \mathcal{E}v$ or $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$) satisfies Invariant 9, then so does the initial configuration of the assumption, $E'', \sigma'', M'', \mathcal{T}'', \mathcal{E}v''$ or $E'', \sigma'', \mathsf{defined}(M''_1, \dots, M''_m) \wedge M'', \mathcal{T}'', \mathcal{E}v''$.

Proof sketch The proof proceeds by induction following the definition of $\xrightarrow{p}_t$. We just sketch the main arguments.

If $x[M_1, \dots, M_m]$ is in the second case of Invariant 9, and we execute the definition of $x[M_1, \dots, M_m]$, then for all $j \leq m$, M_j is a variable replication index and $x[\sigma(M_1), \dots, \sigma(M_m)]$ is added to $\mathrm{Dom}(E)$ by rules (NewT), (LetT), (FindT1), (GetT1), (New), (Let), (Find1), (Output), or (Get1) so it moves to the first case of Invariant 9.

If $x[M_1, \dots, M_m]$ is in the third case of Invariant 9, and we execute the corresponding find, this access to x simply disappears.

If $x[M_1, \dots, M_m]$ is in the fourth case of Invariant 9, and we execute the find, then $x[M_1, \dots,$ $M_m]$ is a subterm of M'_{jk} for some $j \leq m''$ and $k \leq l_j$. Therefore, the initial configuration of the assumption $E, \sigma[\widetilde{i_j} \mapsto \widetilde{a}], D_j \wedge M'_j, \mathcal{T}, \mathcal{E}v \xrightarrow{p_{k'}}{}^*_{t_{k'}} E'', \sigma', r_{k'}, \mathcal{T}, \mathcal{E}v$ with $D_j \wedge M'_j = \mathsf{defined}(M'_{j1},$ $\dots, M'_{jl_j}) \wedge M'_j$ and $\sigma' = \sigma[\widetilde{i_j} \mapsto \widetilde{a}]$ also satisfies Invariant 9. In case this assumption is reduced by (DefinedYes), we have $E, \sigma', M'_{jk}, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma', a_{jk}, \mathcal{T}, \mathcal{E}v$. Therefore, by Lemma 4, for all $j' \leq m$, $E, \sigma', M_{j'}, \mathcal{T}, \mathcal{E}v \xrightarrow{1}^* E, \sigma', a_{j'}, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m]$ is in $\mathrm{Dom}(E)$. So $x[M_1, \dots, M_m]$ moves to the first case of Invariant 9 in $E, \sigma', M'_j, \mathcal{T}, \mathcal{E}v$ after reduction by (DefinedYes).

If $x[M_1, \dots, M_m]$ is in the last case of Invariant 9, and we execute the find selecting branch j by (FindT1) or (Find1), then there is a subterm N of M'_{jk} for some $k \leq l_j$ such that $N\{\widetilde{u_j}[\widetilde{i}]/\widetilde{i_j}\} = x[M_1, \dots, M_m]$. By hypothesis of (FindT1) or (Find1), we have $E, \sigma[\widetilde{i_j} \mapsto$

$\widetilde{a'}], D_j \wedge M'_j, \mathcal{T}, \mathcal{E}v \xrightarrow{p_{k'}}{}^*_{t_{k'}} E, \sigma', r_{k'}, \mathcal{T}, \mathcal{E}v$ where $r_{k'} = \text{true}$, $v_0 = (j, \widetilde{a'}) \in S$, $D_j \wedge M'_j = \mathsf{defined}(M'_{j1}, \dots, M'_{jl_j}) \wedge M'_j$, and $\sigma' = \sigma[\widetilde{i_j} \mapsto \widetilde{a'}]$. This assumption cannot reduce by (DefinedNo) because the result is true, so it reduces by (DefinedYes). Therefore, we have $E, \sigma', M'_{jk}, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^* E, \sigma', a, \mathcal{T}, \mathcal{E}v$ for some a. The term $N = x[N_1, \dots, N_m]$ is a subterm of M'_{jk}. Therefore, by Lemma 4, for all $j' \leq m$, $E, \sigma', N_{j'}, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^* E, \sigma', a_{j'}, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m]$ is in $\mathrm{Dom}(E)$. Moreover, the resulting environment E' is an extension of E, so a fortiori for all $j' \leq m$, $E', \sigma', N_{j'}, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^* E', \sigma', a_{j'}, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m]$ is in $\mathrm{Dom}(E')$. We have for all $j' \leq m$, $M_{j'} = N_{j'}\{\widetilde{u_j}[\widetilde{i}]/\widetilde{i_j}\}$, $E'(\widetilde{u_j}[\widetilde{i}]) = \widetilde{a'}$, and $\sigma'(\widetilde{i_j}) = \widetilde{a'}$, so by Lemma 5, for all $j' \leq m$, $E', \sigma, M_{j'}, \mathcal{T}, \mathcal{E}v \xrightarrow{1}{}^* E', \sigma, a_{j'}, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m]$ is in $\mathrm{Dom}(E')$. So $x[M_1, \dots, M_m]$ also moves to the first case of Invariant 9.

In all other cases, the situation remains unchanged. For context rules, this is because, in the allowed contexts, the hole is never under a defined condition. □

Therefore, if Q_0 satisfies Invariant 2, then in traces of Q_0, the test $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$ in rule (Var) always succeeds, except when the considered term occurs in a defined condition of a find.

Indeed, consider an application of rule (Var), where the array access $x[M_1, \dots, M_m]$ is not in a defined condition of a find. Then, this array access is not under any variable definition or find, so it is present in $\mathrm{Dom}(E)$: for all $j \leq m$, $E, \sigma, M_j, \mathcal{T}, \mathcal{E}v \xrightarrow{1} E, \sigma, a_j, \mathcal{T}, \mathcal{E}v$ and $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$. Hence, the test $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$ succeeds.

2.4.4 Typing

In this section, we show that our type system is compatible with the semantics of the calculus, that is, we define a notion of typing for semantic configurations and show that typing is preserved by reduction (subject reduction). Finally, the property that semantic configurations are well-typed shows that certain conditions in the semantics always hold.

We use the following definitions:

- $\mathcal{E} \vdash E$ if and only if $E(x[a_1, \dots, a_m]) = a$ implies $\mathcal{E}(x) = T_1 \times \dots \times T_m \to T$ with for all $j \leq m$, $a_j \in T_j$ and $a \in T$.
- We define $\mathcal{E} \vdash P$, $\mathcal{E} \vdash Q$, and $\mathcal{E} \vdash M : T$ as in Section 2.3, with the additional rule $\mathcal{E} \vdash a : T$ if and only if $a \in T$. (This rule is useful to type evaluated terms.)
- $\mathcal{E} \vdash (\sigma, P)$ if and only if $\mathcal{E}[i_1 \mapsto [1, n_1], \dots, i_m \mapsto [1, n_m]] \vdash P$ and for all $j \leq m$, $\sigma(i_j) \in [1, n_j]$ for some $n_1, \dots, n_m$, where $\mathrm{Dom}(\sigma) = \{i_1, \dots, i_m\}$. The jugdments $\mathcal{E} \vdash (\sigma, Q)$ and $\mathcal{E} \vdash (\sigma, M) : T$ are defined in the same way.
- $\mathcal{E} \vdash \mathcal{T}$ if and only if $\mathit{Tbl}(a_1, \dots, a_m) \in \mathcal{T}$ implies $\mathit{Tbl} : T_1 \times \dots \times T_m$ with for all $j \leq m$, $a_j \in T_j$.
- $\mathcal{E} \vdash \mathcal{E}v$ if and only if $e(a_1, \dots, a_m) \in \mathcal{E}v$ implies $e : T_1 \times \dots \times T_m$ with for all $j \leq m$, $a_j \in T_j$.
- $\mathcal{E} \vdash E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$ if and only if $\mathcal{E} \vdash E$, $\mathcal{E} \vdash (\sigma, P)$, $\mathcal{E} \vdash \mathcal{T}$, $\mathcal{E} \vdash \mathcal{E}v$, and for all $(\sigma', Q) \in \mathcal{Q}$, $\mathcal{E} \vdash (\sigma', Q)$.
- $\mathcal{E} \vdash E, \mathcal{Q}, \mathcal{C}$ if and only if $\mathcal{E} \vdash E$ and for all $(\sigma', Q) \in \mathcal{Q}$, $\mathcal{E} \vdash (\sigma', Q)$.
- $\mathcal{E} \vdash E, \sigma, M : T, \mathcal{T}, \mathcal{E}v$ if and only if $\mathcal{E} \vdash E$, $\mathcal{E} \vdash (\sigma, M) : T$, $\mathcal{E} \vdash \mathcal{T}$, and $\mathcal{E} \vdash \mathcal{E}v$.

Lemma 7 *If $\mathcal{E} \vdash E, \sigma, M : T, \mathcal{T}, \mathcal{E}v$ and $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$, then $\mathcal{E} \vdash E', \sigma', M' : T, \mathcal{T}', \mathcal{E}v'$.*

So, $\mathcal{E} \vdash E, \sigma, M : T, \mathcal{T}, \mathcal{E}v$ and $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}{}^{*}_t E', \sigma', a, \mathcal{T}', \mathcal{E}v'$, then $\mathcal{E} \vdash E', \sigma', a : T, \mathcal{T}', \mathcal{E}v'$.

Proof sketch By induction on the derivation of $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$. $\square$

Lemma 8 *If $\mathcal{E} \vdash E, \mathcal{Q}, \mathcal{C}$ and $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E', \mathcal{Q}', \mathcal{C}'$, then $\mathcal{E} \vdash E', \mathcal{Q}', \mathcal{C}'$.*

So, if $\mathcal{E} \vdash E, \mathcal{Q}, \mathcal{C}$, then $\mathcal{E} \vdash \mathrm{reduce}(E, \mathcal{Q}, \mathcal{C})$.

Proof sketch By cases on the derivation of $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E', \mathcal{Q}', \mathcal{C}'$. In the case of the replication, we have $\mathcal{E} \vdash (\sigma, !^{i \leq n} Q)$, so $\mathcal{E}[i_1 \mapsto [1, n_1], \dots, i_m \mapsto [1, n_m]] \vdash !^{i \leq n} Q$ and for all $j \leq m$, $\sigma(i_j) \in [1, n_j]$ for some $n_1, \dots, n_m$, where $\mathrm{Dom}(\sigma) = \{i_1, \dots, i_m\}$. By (TRepl), $\mathcal{E}[i_1 \mapsto [1, n_1], \dots, i_m \mapsto [1, n_m], i \mapsto [1, n]] \vdash Q$, so $\mathcal{E} \vdash (\sigma[i \mapsto a], Q)$ for $a \in [1, n]$. In the case of the input, we use Lemma 7. $\square$

Lemma 9 *If $\mathcal{E} \vdash Q_0$, then $\mathcal{E} \vdash \mathrm{initConfig}(Q_0)$.*

Proof sketch By Lemma 8 and the previous definitions. $\square$

Lemma 10 (Subject reduction) *If $\mathcal{E} \vdash E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$ and $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$, then $\mathcal{E} \vdash E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$.*

Proof sketch By cases on the derivation of $E, P, \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$, using Lemmas 7 and 8. $\square$

Moreover, if the rules that define $E, \sigma, M, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', \sigma', M', \mathcal{T}', \mathcal{E}v'$ (resp. $E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v \xrightarrow{p}_t E', (\sigma', P'), \mathcal{Q}', \mathcal{C}', \mathcal{T}', \mathcal{E}v'$) require as assumption $E'', \sigma'', M'', \mathcal{T}'', \mathcal{E}v'' \xrightarrow{p}_t \dots$ and the initial configuration is well-typed $\mathcal{E} \vdash E, \sigma, M : T, \mathcal{T}, \mathcal{E}v$ (resp. $\mathcal{E} \vdash E, (\sigma, P), \mathcal{Q}, \mathcal{C}, \mathcal{T}, \mathcal{E}v$) then so is the initial configuration of the assumption, that is, there exists T'' such that $\mathcal{E} \vdash E'', \sigma'', M'' : T'', \mathcal{T}'', \mathcal{E}v''$.

As an immediate consequence of Lemmas 9, 10, and 7 and the observation above, we obtain: if Q_0 satisfies Invariant 7, then in traces of Q_0, the tests $a \in T$ in rules (LetT) and (Let) and $\forall j \leq m, a_j \in T_j$ in rule (Fun) always succeed. Moreover, in rules (NewT) and (New), we always have that T is *fixed*, *bounded*, or *nonuniform*. In the rules for find, we have $r_k \in \{\mathrm{false}, \mathrm{true}\}$ when r_k is a value (not an event). In the rules (InsertT), (GetTE), (GetT1), (GetT2), (Insert), (GetE), (Get1), and (Get2), we have $a_j \in T_j$ for $j \leq l$, where $\mathit{Tbl} : T_1 \times \ldots \times T_l$. In the rules (EventT) and (Event), we have $a_j \in T_j$ for $j \leq l$, where $e : T_1 \times \ldots \times T_l$.

2.5 Subset for the Initial Game

What is described in this section is not implemented yet, it is a project. Currently, process macros with arguments are not implemented. CryptoVerif additionally requires that variables in V and in defined *conditions have a single definition.*

The variables are always defined with the current replication indices $\widetilde{i}$, so we omit them, writing x for $x[\widetilde{i}]$; they are implicitly added by CryptoVerif. When a variable is used with the current replication indices, we can also omit the indices.

Along similar lines, the channels c are used without indices, and the current replication indices are implicitly added by CryptoVerif. This allows the adversary the select to which copy of processes it sends messages. The construct newChannel cannot occur in games manipulated by CryptoVerif. It is used only inside proofs. The grammar of the resulting calculus is summarized in Figure 8.

$M, N ::=$	terms
i	replication index
$x[M_1, \dots, M_m]$	variable access
$f(M_1, \dots, M_m)$	function application
$\mathsf{new}\ x : T; N$	random number
$\mathsf{let}\ p = M\ \mathsf{in}\ N\ \mathsf{else}\ N'$	assignment (pattern-matching)
$\mathsf{let}\ x : T = M\ \mathsf{in}\ N$	assignment
$\mathsf{if}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ N\ \mathsf{else}\ N'$	conditional
$\mathsf{find}[\mathit{unique}?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}$ $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M'_j\ \mathsf{then}\ N_j)\ \mathsf{else}\ N'$	array lookup
$\mathsf{insert}\ \mathit{Tbl}(M_1, \dots, M_l); N$	insert in table
$\mathsf{get}\ \mathit{Tbl}(p_1, \dots, p_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ N\ \mathsf{else}\ N'$	get from table
$\mathsf{event}\ e(M_1, \dots, M_l); N$	event
$\mathsf{event\_abort}\ e$	event e and abort
$p ::=$	pattern
$x : T$	variable
$f(p_1, \dots, p_m)$	function application
$=M$	comparison with a term
$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq n} Q$	replication n times
$c(p); P$	input
$P ::=$	output process
$\overline{c}\langle N \rangle; Q$	output
$\mathsf{new}\ x : T; P$	random number
$\mathsf{let}\ p = M\ \mathsf{in}\ P\ \mathsf{else}\ P'$	assignment
$\mathsf{if}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$	conditional
$\mathsf{find}[\mathit{unique}?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}$ $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$	array lookup
$\mathsf{insert}\ \mathit{Tbl}(M_1, \dots, M_l); P$	insert in table
$\mathsf{get}\ \mathit{Tbl}(p_1, \dots, p_l)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$	get from table
$\mathsf{event}\ e(M_1, \dots, M_l); P$	event
$\mathsf{event\_abort}\ e$	event e and abort
yield	end

Figure 8: Subset of the calculus for the initial game

We recommend using the constructs get and insert to manage key tables, instead of find or if with defined conditions. When no find nor if with defined conditions occurs in the game, by Invariant 2, all accesses to variable x are of the form $x[\widetilde{i}]$ where $\widetilde{i}$ are the current replication indices at the definition of x. Such accesses are simply abbreviated as x. Variables can then be considered as ordinary variables instead of arrays, since we only access the array cell at the current replication indices. This choice has several other advantages:

- Tables with get/insert are closer to lists usually used by cryptographers than find, they should be easier to understand for the user.
- Tables are supported by the symbolic protocol verifier ProVerif while find is not. Similarly, ProVerif does not support channels with indices. So avoiding find and channels with indices allows us to have a language compatible with ProVerif.
- Our compiler that translates CryptoVerif specifications into OCaml implementations [6] does not support find, because tables with get/insert are also much easier to implement than find.

We can define processes by macros: let $pid(x_1 : T_1, \dots, x_m : T_m) = P$ or let $qid(x_1 : T_1, \dots, x_m : T_m) = Q$. If a process $pid(M_1, \dots, M_m)$ occurs in the initial game, CryptoVerif verifies that $M_1, \dots, M_m$ are of types $T_1, \dots, T_m$ respectively, and replaces $pid(M_1, \dots, M_m)$ with the expansion $P\{M_1/x_1, \dots, M_m/x_m\}$.

We can also define functions by macros: letfun $f(x_1 : T_1, \dots, x_m : T_m) = M$. If a term $f(M_1, \dots, M_m)$ occurs in the initial game, CryptoVerif verifies that $M_1, \dots, M_m$ are of types $T_1, \dots, T_m$ respectively, and replaces $f(M_1, \dots, M_m)$ with the expansion $M\{M_1/x_1, \dots, M_m/x_m\}$.

In the initial game, all bound variables that do not occur in V nor in defined conditions of find or if are renamed to distinct names, so that Invariant 1 is satisfied for these variables. The condition on input channels in Invariant 5 is always satisfied by definition of the language. CryptoVerif checks the rest of the invariants.

2.6 Subsets used inside the Sequence of Games

During the computation of the sequence of games, several properties are used by CryptoVerif, either required by some game transformations or guaranteed by others. We summarize them in this section.

Property 1 No function takes as argument or returns values of interval types. The types of the elements of tables, of the arguments of events, of values chosen by new $x[\widetilde{i}] : T$ are not interval types. The type T of the sent message in the (TOut) rule and of the receiving pattern in the (TIn) rule are not interval types.

This property is satisfied by all games manipulated by CryptoVerif, but not by processes that model the adversary. Combined with Invariants 7, 2, and 5, it implies that the terms of variable accesses $x[M_1, \dots, M_m]$ contain only replication indices and variables.

For processes that model security assumptions on primitives, the receiving variable can be of an interval type. (This is used for instance to specify the computational Diffie-Hellman assumption.)

Property 2 The newChannel c construct does not appear in games.

Property 3 The indices of channels are always the current replication indices.

These properties are also satisfied by all games manipulated by CryptoVerif, but not by processes that model the adversary.

$M, N ::=$	terms
i	replication index
$x[M_1, \dots, M_m]$	variable access
$f(M_1, \dots, M_m)$	function application
$FC ::=$	find condition
M	term
$\mathsf{let}\ p = M\ \mathsf{in}\ FC\ \mathsf{else}\ FC'$	assignment (pattern-matching)
$\mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ N$	assignment
$\mathsf{if}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ FC\ \mathsf{else}\ FC'$	conditional
$\mathsf{find}[unique?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}$ $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge FC'_j\ \mathsf{then}\ FC_j)\ \mathsf{else}\ FC''$	array lookup
$p ::=$	pattern
$x[\widetilde{i}] : T$	variable
$f(p_1, \dots, p_m)$	function application
$=M$	comparison with a term
$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq n} Q$	replication n times
$c[M_1, \dots, M_l](p); P$	input
$P ::=$	output process
$\overline{c[M_1, \dots, M_l]}\langle N \rangle; Q$	output
$\mathsf{new}\ x[\widetilde{i}] : T; P$	random number
$\mathsf{let}\ p = M\ \mathsf{in}\ P\ \mathsf{else}\ P'$	assignment
$\mathsf{if}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$	conditional
$\mathsf{find}[unique?]\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathsf{suchthat}$ $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge FC_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$	array lookup
$\mathsf{event}\ e(M_1, \dots, M_l); P$	event
$\mathsf{event\_abort}\ e$	event e and abort
yield	end

Figure 9: Subset after game expansion

Property 4 The constructs insert and get do not occur in the game.

This property is not valid in the initial game, but it is in all other games of the sequence produced by CryptoVerif. The very first game transformation applied by CryptoVerif, **ExpandTables**, encodes insert and get using find. The constructs insert and get are never introduced by subsequent game transformations, so this property remains valid in the rest of the sequence.

Property 5 The variables defined in conditions of find have pairwise distinct names.

This property is enforced by the transformation **AutoSArename** by renaming variables defined in conditions of find to distinct names. (This is easy since these variables do not have array accesses by Invariant 3.) Property 5 is required as a precondition by many game transformations, and may be broken by game transformations that duplicate code. Therefore, we apply **AutoSArename** after these game transformations.

Property 6 The terms M contain only replication indices, variables, and function applications except for conditions of find.

The grammar of the language taking into account this property as well as Properties 2 and 4 is shown in Figure 9. By Invariant 4, new and event do not occur in conditions of find, so new and event never occur as terms. Property 6 is enforced by the transformation **Expand** by converting other terms into processes. This transformation is applied on the initial game after **ExpandTables**. Property 6 is broken by the cryptographic transformation that relies on assumptions on primitives, so **Expand** is called again after this transformation. Most game transformations require Property 6 as a precondition.

2.7 Security Properties, Indistinguishability

A context is a process containing a hole $[\,]$. An evaluation context C is a context built from $[\,]$, newChannel $c; C$, $Q \mid C$, and $C \mid Q$. We use an evaluation context to represent the adversary. We denote by $C[Q]$ the process obtained by replacing the hole $[\,]$ in the context C with the process Q.

Definition 4 (Indistinguishability) Let Q and Q' be two processes and V a set of variables. Assume that Q and Q' satisfy Invariants 1 to 7 with public variables V, and the variables of V are defined in Q and Q', with the same types.

An evaluation context C is said to be *acceptable* for Q with public variables V if and only if $\mathrm{var}(C) \cap \mathrm{var}(Q) \subseteq V$, $\mathrm{vardef}(C) \cap V = \emptyset$, C and Q do not use any common table, and $C[Q]$ satisfies Invariants 1 to 7 with public variables V.

We write $Q \approx_p^V Q'$ when, for all evaluation contexts C acceptable for Q and Q' with public variables V and all distinguishers D that run in time at most t_D, $|\Pr[C[Q] : D] - \Pr[C[Q'] : D]| \leq p(C, t_D)$.

This definition formalizes that the probability that algorithms C and D distinguish the games Q and Q' is at most $p(C, t_D)$. The probability p typically depends on the runtime of C and D, but may also depend on other parameters, such as the number of queries to each oracle made by C. That is why p takes as arguments the whole algorithm C and the runtime of D.

The unusual requirement on variables of C comes from the presence of arrays and of the associated find construct which gives C direct access to variables of Q and Q': the context C is allowed to access variables of Q and Q' only when they are in V. (In more standard settings, the calculus does not have constructs that allow the context to access variables of Q and Q'.) When V is empty, we omit it and write $Q \approx_p Q'$.

When C is acceptable for Q with public variables V, and we transform Q into Q', we can rename the fresh variables of Q' (introduced by the game transformation) so that they do not occur in C. Then C is also acceptable for Q' with public variables V. (To establish this property, we use that the variables of V are defined in Q and Q', with the same types, so that, if $C[Q]$ is well-typed, then so is $C[Q']$.)

When C is acceptable for Q with public variables V, we have that $\mathrm{vardef}(C) \cap \mathrm{var}(Q) = \emptyset$, because $\mathrm{vardef}(C) \cap \mathrm{var}(Q) = \mathrm{vardef}(C) \cap \mathrm{var}(C) \cap \mathrm{var}(Q) \subseteq \mathrm{vardef}(C) \cap V = \emptyset$.

The following lemma is a straightforward consequence of Definition 4:

Lemma 11 *1. Reflexivity: $Q \approx_0^V Q$.*

2. Symmetry: If $Q \approx_p^V Q'$, then $Q' \approx_p^V Q$.

3. Transitivity: If $Q \approx_p^V Q'$ and $Q' \approx_{p'}^V Q''$, then $Q \approx_{p+p'}^V Q''$.

4. If $Q \approx_p^V Q'$ and C is an evaluation context acceptable for Q and Q' with public variables V, then $C[Q] \approx_{p'}^{V'} C[Q']$, where $p'(C', t_D) = p(C'[C[\,]], t_D)$ and $V' \subseteq V \cup \mathrm{var}(C)$.

Definition 5 (Indistinguishability with introduction of events) Let Q and Q' be two processes and V a set of variables. Assume that Q and Q' satisfy Invariants 1 to 7 with public variables V, and the variables of V are defined in Q and Q', with the same types.

Let $\mathsf{NonUnique}_Q = \bigvee\{e \mid \mathsf{find}[\mathsf{unique}_e] \text{ occurs in } Q\}$.

We write $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$ when $D^+ = e_1 \vee \ldots \vee e_m$ where $e_1, \ldots, e_m$ are Shoup events, the events that occur in Q are in EvUsed, $\mathit{EvUsed} \subseteq \mathit{EvUsed}'$, the events $e_1, \ldots, e_m$ and the events that occur in Q' but not in Q are in EvUsed' but not in EvUsed, and, for all evaluation contexts C acceptable for Q and Q' with public variables V that do not contain non-unique events that occur in Q nor events in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$, and all distinguishers D that run in time at most t_D,

$$\Pr[C[Q] : D \wedge \neg\mathsf{NonUnique}_Q] \le \Pr[C[Q'] : (D \vee D^+) \wedge \neg\mathsf{NonUnique}_{Q'}] + p(C, t_D)$$
$$\Pr[C[Q] : D \vee \mathsf{NonUnique}_Q] \ge \Pr[C[Q'] : D \vee \mathsf{NonUnique}_{Q'}] - p(C, t_D).$$

Intuitively, the events EvUsed are those used by CryptoVerif in the sequence of games until the game Q included, while the events EvUsed' are those used until Q'. Hence, EvUsed contains the events that occur in Q; EvUsed' contains EvUsed and the events that occur in Q'. The formula D^+ uses fresh Shoup events introduced during the transformation of Q into Q'; hence these events are in EvUsed' but not in EvUsed. More generally, all events that occur in Q' but not in Q are fresh events introduced in the transformation of Q into Q', so they are in EvUsed' but not in EvUsed.

We need to introduced distinct events for $\mathsf{find}[\mathsf{unique}_e]$ because we need to distinguish the non-unique events that occur in Q from those that occur in the context C. The context C must not contain the fresh events introduced in the transformation of Q into Q' (which can be guaranteed by choosing these fresh events appropriately).

Lemma 12 *Let $D_{\mathrm{false}}(\mathcal{E}v, a) = \mathrm{false}$ for all $\mathcal{E}v, a$.*

1. Link with indistinguishability: Suppose that Q and Q' do not contain non-unique events, the events that occur in Q are in EvUsed, and the events that occur in Q' also occur in Q. Then $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D_{\mathrm{false}}} Q', \mathit{EvUsed}$ if and only if $Q \approx_p^V Q'$.

2. Reflexivity: if the events that occur in Q are in EvUsed, then $Q, \mathit{EvUsed} \xrightarrow{V}_{0,D_{\mathrm{false}}} Q$, EvUsed.

3. Transitivity: If $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D_1^+} Q', \mathit{EvUsed}'$ and $Q', \mathit{EvUsed}' \xrightarrow{V}_{p',D_2^+} Q'', \mathit{EvUsed}''$, then $Q, \mathit{EvUsed} \xrightarrow{V}_{p'',D_1^+ \vee D_2^+} Q'', \mathit{EvUsed}''$, where $p''(C, t_D) = p(C, t_D) + p'(C, t_D + t_{D_1^+})$.

4. *If $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$ and C is a context acceptable for Q and Q' with public variables V such that C does not contain the non-unique events in Q and the events that occur in C are in EvUsed, then $C[Q], \mathit{EvUsed} \xrightarrow{V'}_{p',D^+} C[Q'], \mathit{EvUsed}'$, where $p'(C', t_D) = p(C'[C[\,]], t_D)$ and $V' \subseteq V \cup \mathrm{var}(C)$.*

5. *Renaming: if $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$ and EvUsed^+ is a set of events disjoint from EvUsed, then $Q, \mathit{EvUsed} \cup \mathit{EvUsed}^+ \xrightarrow{V}_{p',\sigma D^+} \sigma Q', \sigma\mathit{EvUsed}' \cup \mathit{EvUsed}^+$ where σ is a renaming of the events in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$ to events not in $\mathit{EvUsed} \cup \mathit{EvUsed}^+$, $p'(C, t_D) = p(\sigma^{-1}C, t_D)$, , and the distinguisher $D^+ \circ \sigma^{-1}$ is defined by $(D^+ \circ \sigma^{-1})(\mathcal{E}v, a) = D^+(\sigma^{-1}\mathcal{E}v, a)$.*

In the last property, the formula that expresses the probability p is often independent of the names of events; in this case, we have $p' = p$. If $D^+ = e_1 \vee \ldots \vee e_m$, then $D^+ \circ \sigma^{-1} = \sigma e_1 \vee \ldots \vee \sigma e_m$.

Proof Property 1: Since Q and Q' do not contain non-unique events, $\mathsf{NonUnique}_Q$ and $\mathsf{NonUnique}_{Q'}$ are always false, so given the hypothesis on EvUsed, $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D_{\text{false}}} Q'$, EvUsed reduces to: for all evaluation contexts C acceptable for Q and Q' with public variables V and all distinguishers D that run in time at most t_D,

$$\begin{aligned}\Pr[C[Q] : D] &\le \Pr[C[Q'] : D] + p(C, t_D)\\ \Pr[C[Q] : D] &\ge \Pr[C[Q'] : D] - p(C, t_D).\end{aligned}$$

which is exactly $Q \approx_p^V Q'$.

Property 2: Obvious.

Property 3: The events in Q are in EvUsed. We have $\mathit{EvUsed} \subseteq \mathit{EvUsed}' \subseteq \mathit{EvUsed}''$. The distinguisher $D_1^+ \vee D_2^+$ is a disjunction of Shoup events that occur in D_1^+ or in D_2^+; in the former case, they are in they are in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$; in the latter case, they are in they are in $\mathit{EvUsed}'' \setminus \mathit{EvUsed}'$; so in both cases they are in $\mathit{EvUsed}'' \setminus \mathit{EvUsed}$. The events that occur in Q'' but not in Q occur either in Q'' but not in Q' or in Q' but not in Q; in the former case, they are in $\mathit{EvUsed}'' \setminus \mathit{EvUsed}'$; in the latter case, they are in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$; so in both cases they are in $\mathit{EvUsed}'' \setminus \mathit{EvUsed}$.

Let C be any evaluation context acceptable for Q and Q'' with public variables V that does not contain non-unique events that occur in Q nor events in $\mathit{EvUsed}'' \setminus \mathit{EvUsed}$. After renaming the fresh variables of Q' that do not occur in Q and Q'' and the tables of Q' that do not occur in Q and Q'' so that they do not occur in C, C is also acceptable for Q' with public variables V. Since $\mathit{EvUsed} \subseteq \mathit{EvUsed}' \subseteq \mathit{EvUsed}''$, C does not contain events in $\mathit{EvUsed}'' \setminus \mathit{EvUsed}'$ nor in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$. Moreover, the non-unique events in Q' either occur in Q, or they occur in Q' but not Q, so they are in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$, so in both cases, they do not occur in C. Let D be any distinguisher D that runs in time at most t_D.

Then we have:

$$\begin{aligned}
&\Pr[C[Q] : D \wedge \neg\mathsf{NonUnique}_Q]\\
&\quad\le \Pr[C[Q'] : (D \vee D_1^+) \wedge \neg\mathsf{NonUnique}_{Q'}] + p(C, t_D) \quad \text{since } Q, \mathit{EvUsed} \xrightarrow{V}_{p,D_1^+} Q', \mathit{EvUsed}'\\
&\quad\le \Pr[C[Q''] : ((D \vee D_1^+) \vee D_2^+) \wedge \neg\mathsf{NonUnique}_{Q''}] + p'(C, t_{D \vee D_1^+}) + p(C, t_D)\\
&\qquad\qquad \text{since } Q', \mathit{EvUsed}' \xrightarrow{V}_{p',D_2^+} Q'', \mathit{EvUsed}''\\
&\quad\le \Pr[C[Q''] : (D \vee (D_1^+ \vee D_2^+)) \wedge \neg\mathsf{NonUnique}_{Q''}] + p(C, t_D) + p'(C, t_D + t_{D_1^+})\\
&\qquad\qquad \text{since } t_{D \vee D_1^+} \le t_D + t_{D_1^+}\\
&\quad\le \Pr[C[Q''] : (D \vee (D_1^+ \vee D_2^+)) \wedge \neg\mathsf{NonUnique}_{Q''}] + p''(C, t_D) \qquad \text{by definition of } p''
\end{aligned}$$

Similarly:

$$\begin{aligned}
&\Pr[C[Q] : D \vee \mathsf{NonUnique}_{Q}] \\
&\quad\geq \Pr[C[Q'] : D \vee \mathsf{NonUnique}_{Q'}] - p(C, t_D) \qquad \text{since } Q, \mathit{EvUsed} \xrightarrow{V}_{p,D_1^+} Q', \mathit{EvUsed}' \\
&\quad\geq \Pr[C[Q''] : D \vee \mathsf{NonUnique}_{Q''}] - p'(C, t_D) - p(C, t_D) \\
&\qquad\qquad \text{since } Q', \mathit{EvUsed}' \xrightarrow{V}_{p',D_2^+} Q'', \mathit{EvUsed}'' \\
&\quad\geq \Pr[C[Q''] : D \vee \mathsf{NonUnique}_{Q''}] - p''(C, t_D) \qquad \text{by definition of } p'', \text{ since } t_D \leq t_D + t_{D_1^+}
\end{aligned}$$

Therefore, we have $Q, \mathit{EvUsed} \xrightarrow{V}_{p'',D_1^+ \vee D_2^+} Q'', \mathit{EvUsed}''$.

Property 4: The events that occur in $C[Q]$ are either in C or in Q; the former case, they are in EvUsed by hypothesis; in the latter case, they are also in EvUsed since $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$. We have $\mathit{EvUsed} \subseteq \mathit{EvUsed}'$ since $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$. The events that occur in $C[Q']$ but not in $C[Q]$ are in Q' but not in Q, so they are in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$. The distinguisher D^+ is a disjunction of Shoup events that are in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$, since $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$.

Let C' be any evaluation context acceptable for $C[Q]$ and $C[Q']$ with public variables V' that does not contain non-unique events that occur in $C[Q]$ nor events in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$. We rename the variables of C' not in V' so that they are not in V; this renaming does not change the probabilities. Then $C'[C[\,]]$ is an evaluation context acceptable for Q and Q' with public variables V. Indeed,

$$\begin{aligned}
\mathrm{var}(C'[C[\,]]) \cap \mathrm{var}(Q) &= (\mathrm{var}(C') \cup \mathrm{var}(C)) \cap \mathrm{var}(Q) \\
&\subseteq (V' \cup \mathrm{var}(C)) \cap \mathrm{var}(Q) \\
&\qquad \text{since } \mathrm{var}(C') \cap \mathrm{var}(Q) \subseteq \mathrm{var}(C') \cap \mathrm{var}(C[Q]) \subseteq V' \\
&\subseteq (V \cup \mathrm{var}(C)) \cap \mathrm{var}(Q) \qquad \text{since } V' \subseteq V \cup \mathrm{var}(C) \\
&\subseteq V \qquad \text{since } \mathrm{var}(C) \cap \mathrm{var}(Q) \subseteq V
\end{aligned}$$

We have similarly $\mathrm{var}(C'[C[\,]]) \cap \mathrm{var}(Q') \subseteq V$. We also have $\mathrm{vardef}(C'[C[\,]]) \cap V = (\mathrm{vardef}(C') \cap V) \cup (\mathrm{vardef}(C) \cap V) = \emptyset$ since $\mathrm{vardef}(C) \cap V = \emptyset$ because C is an acceptable evaluation context for Q with public variables V and $\mathrm{vardef}(C') \cap V \subseteq \mathrm{vardef}(C') \cap V' = \emptyset$ because we have renamed the variables of C' not in V' so that they are not in V and C' is an acceptable evaluation context for $C[Q]$ and with public variables V'. Moreover, C and Q do not use any common table, and C' and $C[Q]$ do not use any common table so a fortiori C' and Q do not use any common table. Therefore, $C'[C[\,]]$ and Q do not use any common table. Similarly, $C'[C[\,]]$ and Q' do not use any common table. $C'[C[\,]]$ does not contain non-unique events that occur in Q, since C' does not contain non-unique events that occur in $C[Q]$ so a fortiori in Q, and C does not contain non-unique events that occur in Q. $C'[C[\,]]$ does not contain events in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$, since C' and C do not. By using the property $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$ with the context $C'[C[\,]]$, we get immediately the desired probability bounds.

Property 5: The events that occur in Q are in EvUsed, so a fortiori in $\mathit{EvUsed} \cup \mathit{EvUsed}^+$. We have $\mathit{EvUsed} \cup \mathit{EvUsed}^+ \subseteq \sigma\mathit{EvUsed}' \cup \mathit{EvUsed}^+$, since σ leaves the events of EvUsed unchanged, so $\mathit{EvUsed} = \sigma\mathit{EvUsed} \subseteq \sigma\mathit{EvUsed}'$. The distinguisher $D^+ \circ \sigma^{-1}$ is a disjunction of Shoup events that are in $\sigma(\mathit{EvUsed}' \setminus \mathit{EvUsed}) = \sigma\mathit{EvUsed}' \setminus \mathit{EvUsed} = (\sigma\mathit{EvUsed}' \cup \mathit{EvUsed}^+) \setminus (\mathit{EvUsed} \cup \mathit{EvUsed}^+)$. The events that occur in $\sigma Q'$ but not in $Q = \sigma Q$ are also in $\sigma(\mathit{EvUsed}' \setminus \mathit{EvUsed}) = (\sigma\mathit{EvUsed}' \cup \mathit{EvUsed}^+) \setminus (\mathit{EvUsed} \cup \mathit{EvUsed}^+)$.

Let C be any evaluation context acceptable for Q and $\sigma Q'$ with public variables V that do not contain non-unique events that occur in Q nor events in $\sigma\mathit{EvUsed}' \setminus \mathit{EvUsed}$. Then $\sigma^{-1}C$ is an evaluation context acceptable for Q and Q' with public variables V that do not contain non-unique events that occur in Q nor events in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$. Let D be a distinguisher that runs in time at most t_D. Then $D \circ \sigma$ also runs in time at most t_D. By applying the property $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$ with the context $\sigma^{-1}C$ and the distinguisher $D \circ \sigma$, we obtain:

$$\begin{aligned}
\Pr[(\sigma^{-1}C)[Q] : (D \circ \sigma) \wedge \neg \mathsf{NonUnique}_Q] &\leq \Pr[(\sigma^{-1}C)[Q'] : ((D \circ \sigma) \vee D^+) \wedge \neg \mathsf{NonUnique}_{Q'}] \\
&\quad + p(\sigma^{-1}C, t_D) \\
\Pr[(\sigma^{-1}C)[Q] : (D \circ \sigma) \vee \mathsf{NonUnique}_Q] &\geq \Pr[\sigma^{-1}C[Q'] : (D \circ \sigma) \vee \mathsf{NonUnique}_{Q'}] - p(\sigma^{-1}C, t_D).
\end{aligned}$$

Since renaming events does not change the probabilities of traces, by applying σ, we get:

$$\begin{aligned}
&\Pr[C[Q] : D \wedge \neg \mathsf{NonUnique}_Q] \leq \Pr[C[\sigma Q'] : (D \vee (D^+ \circ \sigma^{-1})) \wedge \neg \mathsf{NonUnique}_{\sigma Q'}] + p(\sigma^{-1}C, t_D) \\
&\Pr[C[Q] : D \vee \mathsf{NonUnique}_Q] \geq \Pr[C[\sigma Q'] : D \vee \mathsf{NonUnique}_{\sigma Q'}] - p(\sigma^{-1}C, t_D).
\end{aligned}$$

which yields the desired result. □

When CryptoVerif transforms a game G into a game G', in most cases, we have $G \approx_p^V G'$, where p is the probability difference coming from the transformation, and computed by CryptoVerif. However, there are two exceptions to this situation:

- transformations that exploit the uniqueness of $\mathsf{find}[\mathsf{unique}_e]$, which are valid only when event e is not executed. These events are taken into account by $\mathsf{NonUnique}_Q$.
- transformations that insert events using Shoup's lemma.

That is why, in general, when CryptoVerif transforms a game G into a game G', we have $G, \mathit{EvUsed} \xrightarrow{V}_{p,e_1 \vee \ldots \vee e_m} G', \mathit{EvUsed}'$, where $e_1, \ldots, e_m$ are the events introduced by Shoup's lemma during the transformation of G into G'.

2.7.1 Secrecy

Let us now define the secrecy properties that are proved by CryptoVerif.

Definition 6 (One-session secrecy) Let C be an evaluation context acceptable for $Q \mid Q_x$ with public variables V ($x \notin V$) that does not contain S nor $\overline{\mathsf{S}}$.

The advantage of the adversary C against the *one-session secrecy* of x in process Q is

$$\mathsf{Adv}_Q^{\mathsf{1-ses.secrecy}(x)}(C) = \Pr[C[Q \mid Q_x] : \mathsf{S}] - \Pr[C[Q \mid Q_x] : \overline{\mathsf{S}}]$$

where

$$\begin{aligned}
Q_x = {} & c_{s0}(); \mathsf{new}\ b : \mathit{bool}; \overline{c_{s0}}\langle\rangle; \\
& (c_s(u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \mathsf{if\ defined}(x[u_1, \ldots, u_m])\ \mathsf{then} \\
& \quad \mathsf{if}\ b\ \mathsf{then}\ \overline{c_s}\langle x[u_1, \ldots, u_m]\rangle\ \mathsf{else\ new}\ y : T; \overline{c_s}\langle y\rangle \\
& \mid c_s'(b' : \mathit{bool}); \mathsf{if}\ b = b'\ \mathsf{then\ event\_abort}\ \mathsf{S}\ \mathsf{else\ event\_abort}\ \overline{\mathsf{S}})
\end{aligned}$$

$c_{s0}, c_s, c_s' \notin \mathrm{fc}(Q)$, $u_1, \ldots, u_m, y, b, b' \notin \mathrm{var}(Q)$, S, $\overline{\mathsf{S}}$ do not occur in Q, and $\mathcal{E}(x) = [1, n_1] \times \ldots \times [1, n_m] \to T$.

The process Q *preserves the one-session secrecy of* x with public variables V ($x \notin V$) up to probability p when, for all evaluation contexts C acceptable for $Q \mid Q_x$ with public variables V that do not contain S nor $\overline{\mathsf{S}}$, $\mathsf{Adv}_Q^{\mathsf{1-ses.secrecy}(x)}(C) \leq p(C)$.

Intuitively, the adversary cannot guess the random bit b, that is, it cannot distinguish whether the process outputs the value of the secret ($b = \text{true}$) or outputs a random number ($b = \text{false}$). The adversary performs a single test query, modeled by Q_x. In more detail, in Q_x, we choose a random bit b; the adversary sends the indices $(u_1, \dots, u_m)$ on channel c_s to perform a test query on $x[u_1, \dots, u_m]$: if $b = \text{true}$, the test query sends back $x[u_1, \dots, u_m]$; if $b = \text{false}$, it sends back a random value y. Finally, the adversary should guess the bit b: it sends its guess b' on channel c'_s and, if the guess is correct, then event S is executed, and otherwise, event $\overline{\mathsf{S}}$ is executed. The probability of getting some information on the secret is the difference between the probability of S and the probability of $\overline{\mathsf{S}}$. (When the adversary always sends a guess on channel c'_s, we have $\Pr[C[Q \mid Q_x] : \overline{\mathsf{S}}] = 1 - \Pr[C[Q \mid Q_x] : \mathsf{S}]$, so the advantage of the adversary is $\mathsf{Adv}_Q^{\mathsf{1\text{-}ses.secrecy}(x)}(C) = \Pr[C[Q \mid Q_x] : \mathsf{S}] - \Pr[C[Q \mid Q_x] : \overline{\mathsf{S}}] = 2\Pr[C[Q \mid Q_x] : \mathsf{S}] - 1$, which is a more standard formula. By flipping a coin, the adversary can execute events S and $\overline{\mathsf{S}}$ with the same probability, that is why the probability that the adversary really guesses b is the difference between the probability of these two events. We need not take the absolute value of $\Pr[C[Q \mid Q_x] : \mathsf{S}] - \Pr[C[Q \mid Q_x] : \overline{\mathsf{S}}]$ because, when it is negative, we can obtain the opposite, positive value by considering an adversary that sends the guess $1 - b'$ instead of b'.)

Definition 7 (Secrecy) Let C be an evaluation context acceptable for $Q \mid R_x$ with public variables V $(x \notin V)$ that does not contain S nor $\overline{\mathsf{S}}$.

The advantage of the adversary C against the *secrecy* of x in process Q is

$$\mathsf{Adv}_Q^{\mathsf{Secrecy}(x)}(C) = \Pr[C[Q \mid R_x] : \mathsf{S}] - \Pr[C[Q \mid R_x] : \overline{\mathsf{S}}]$$

where

```
$R_x = c_{s0}(); \mathsf{new}\ b : bool; \overline{c_{s0}}\langle\rangle;$
    $(!^{i_s \le n_s}\ c_s[i_s](u_1 : [1, n_1], \dots, u_m : [1, n_m]); \mathsf{if\ defined}(x[u_1, \dots, u_m])\ \mathsf{then}$
     $\mathsf{if}\ b\ \mathsf{then}\ \overline{c_s[i_s]}\langle x[u_1, \dots, u_m]\rangle\ \mathsf{else}$
     $\mathsf{find}\ u'_s = i'_s \le n_s\ \mathsf{suchthat\ defined}(y[i'_s], u_1[i'_s], \dots, u_m[i'_s]) \wedge$
         $u_1[i'_s] = u_1 \wedge \dots \wedge u_m[i'_s] = u_m$
     $\mathsf{then}\ \overline{c_s[i_s]}\langle y[u'_s]\rangle$
     $\mathsf{else\ new}\ y : T; \overline{c_s[i_s]}\langle y\rangle$
    $\mid c'_s(b' : bool); \mathsf{if}\ b = b'\ \mathsf{then\ event\_abort}\ \mathsf{S}\ \mathsf{else\ event\_abort}\ \overline{\mathsf{S}})$
```

$c_{s0}, c_s, c'_s \notin \text{fc}(Q)$, $u_1, \dots, u_m, u'_s, y, b, b' \notin \text{var}(Q)$, S, $\overline{\mathsf{S}}$ do not occur in Q, $\mathcal{E}(x) = [1, n_1] \times \dots \times [1, n_m] \to T$.

The process Q *preserves the secrecy of* x with public variables V $(x \notin V)$ up to probability p when, for all evaluation contexts C acceptable for $Q \mid R_x$ with public variables V that do not contain S nor $\overline{\mathsf{S}}$, $\mathsf{Adv}_Q^{\mathsf{Secrecy}(x)}(C) \le p(C)$.

The replication bound n_s is chosen large enough so that it does not prevent communications that would otherwise occur.

Intuitively, the adversary cannot guess b, that is, it cannot distinguish whether the process outputs the value of the secret for several indices ($b = \text{true}$) or outputs independent random numbers ($b = \text{false}$). In this definition, the adversary can perform several test queries, modeled by R_x. This corresponds to the "real-or-random" definition of security [1]. (As shown in [1], this notion is stronger than the more standard approach in which the adversary can perform a single test query and some reveal queries, which always reveal $x[u_1, \dots, u_m]$.)

Lemma 13 *If Q preserves the secrecy of x with public variables V up to probability p and C is an acceptable evaluation context for Q with public variables V, then for all $V' \subseteq V \cup \text{var}(C)$, $C[Q]$ preserves the secrecy of x with public variables V' up to probability p' such that $p'(C') = p(C'[C])$.*

If $Q \approx_p^{V \cup \{x\}} Q'$ and Q preserves the secrecy of x with public variables V up to probability p', then Q' preserves the secrecy of x with public variables V up to probability p'' such that $p''(C) = p'(C) + 2 \times p(C[[\,] \mid R_x], t_{\mathsf{S}})$.

Proof Suppose that Q preserves the secrecy of x with public variables V ($x \notin V$) and C is an acceptable evaluation context for Q with public variables V. Let $V' \subseteq V \cup \mathrm{var}(C)$. Choose channels c_{s0}, c_s, c'_s, variables $u_1, \dots, u_m, u'_s, y, b, b'$, and events S, $\overline{\mathsf{S}}$ such that they do not occur in $C[Q]$. Let C' be an acceptable evaluation context for $C[Q] \mid R_x$ with public variables V' that does not contain S nor $\overline{\mathsf{S}}$. Then we have

$$\begin{aligned}
\mathsf{Adv}_{C[Q]}^{\mathsf{Secrecy}(x)}(C') &= \Pr[C'[C[Q] \mid R_x] : \mathsf{S}] - \Pr[C'[C[Q] \mid R_x] : \overline{\mathsf{S}}] \\
&= \Pr[C'[C[Q \mid R_x]] : \mathsf{S}] - \Pr[C'[C[Q \mid R_x]] : \overline{\mathsf{S}}] \\
&\le p(C'[C])
\end{aligned}$$

We can commute the context C with the parallel composition with R_x because the context C does not bind the channels of R_x. The context $C'[C]$ is an acceptable evaluation context for $Q \mid R_x$ with public variables V that does not contain S nor $\overline{\mathsf{S}}$: there is no common table between C and Q, and between C' and $C[Q] \mid R_x$, so a fortiori between C' and Q and R_x does not use tables, so there is no common table between $C'[C]$ and $Q \mid R_x$; moreover

$$\begin{aligned}
\mathrm{var}(C'[C]) \cap \mathrm{var}(Q \mid R_x) &= ((\mathrm{var}(C') \cap \mathrm{var}(Q \mid R_x)) \cup \mathrm{var}(C)) \cap \mathrm{var}(Q \mid R_x) \\
&\subseteq (V' \cup \mathrm{var}(C)) \cap \mathrm{var}(Q \mid R_x) \quad \text{since } \mathrm{var}(C') \cap \mathrm{var}(C[Q] \mid R_x) \subseteq V' \\
&\subseteq (V \cup \mathrm{var}(C)) \cap \mathrm{var}(Q \mid R_x) \qquad \text{since } V' \subseteq V \cup \mathrm{var}(C) \\
&\subseteq V \qquad \text{since } \mathrm{var}(C) \cap \mathrm{var}(Q) \subseteq V \text{ and } \mathrm{var}(C) \cap \mathrm{var}(R_x) = \emptyset
\end{aligned}$$

Suppose that $Q \approx_p^{V \cup \{x\}} Q'$ and Q preserves the secrecy of x with public variables V up to probability p'. Let C be an acceptable evaluation context for $Q' \mid R_x$ with public variables V that does not contain S nor $\overline{\mathsf{S}}$.

$$\begin{aligned}
\mathsf{Adv}_{Q'}^{\mathsf{Secrecy}(x)}(C) &= \Pr[C[Q' \mid R_x] : \mathsf{S}] - \Pr[C[Q' \mid R_x] : \overline{\mathsf{S}}] \\
&\le \Pr[C[Q \mid R_x] : \mathsf{S}] - \Pr[C[Q \mid R_x] : \overline{\mathsf{S}}] + \\
&\quad |\Pr[C[Q' \mid R_x] : \mathsf{S}] - \Pr[C[Q \mid R_x] : \mathsf{S}]| + \\
&\quad |\Pr[C[Q \mid R_x] : \overline{\mathsf{S}}] - \Pr[C[Q' \mid R_x] : \overline{\mathsf{S}}]| \\
&\le p'(C) + 2 \times p(C[[\,] \mid R_x], t_{\mathsf{S}})
\end{aligned}$$

since $t_{\mathsf{S}} = t_{\overline{\mathsf{S}}}$. Indeed, by renaming the variables and tables of Q that do not appear in Q' to variables and tables that also do not occur in C, C is also an acceptable evaluation context for $Q \mid R_x$ with public variables V. □

2.7.2 Correspondences

In this section, we define non-injective and injective correspondences.

Non-injective Correspondences A non-injective correspondence is a property of the form "if some events have been executed, then some other events have been executed at least once". Here, we generalize these correspondences to implications between logical formulae $\psi \Rightarrow \phi$, which may contain events. We use the following logical formulae:

$\phi ::=$	formula
M	term
$\mathsf{event}(e(M_1, \dots, M_m))$	event
$\phi_1 \wedge \phi_2$	conjunction
$\phi_1 \vee \phi_2$	disjunction

Terms $M, M_1, \ldots, M_m$ in formulae must contain only variables x without array indices and function applications, and their variables are assumed to be distinct from variables of processes. The formula M holds when M evaluates to true. The formula $\mathsf{event}(e(M_1, \ldots, M_n))$ holds when the event $e(M_1, \ldots, M_n)$ has been executed. The conjunction and disjunction are defined as usual. More formally, we write $\rho, \mathcal{E}v \vdash \phi$ when the sequence of events $\mathcal{E}v$ satisfies the formula ϕ, in the environment ρ that maps variables to bitstrings. We define $\rho, \mathcal{E}v \vdash \phi$ as follows:

$\rho, \mathcal{E}v \vdash M$ if and only if $\rho, \emptyset, M, \emptyset, \emptyset \xrightarrow{1}^{*} \rho, \emptyset, \mathrm{true}, \emptyset, \emptyset$
$\rho, \mathcal{E}v \vdash \mathsf{event}(e(M_1, \ldots, M_m))$ if and only if
  for all $j \leq m$, $\rho, \emptyset, M_j, \emptyset, \emptyset \xrightarrow{1}^{*} \rho, \emptyset, a_j, \emptyset, \emptyset$ and $e(a_1, \ldots, a_m) \in \mathcal{E}v$
$\rho, \mathcal{E}v \vdash \phi_1 \wedge \phi_2$ if and only if $\rho, \mathcal{E}v \vdash \phi_1$ and $\rho, \mathcal{E}v \vdash \phi_2$
$\rho, \mathcal{E}v \vdash \phi_1 \vee \phi_2$ if and only if $\rho, \mathcal{E}v \vdash \phi_1$ or $\rho, \mathcal{E}v \vdash \phi_2$

Formulae denoted by ψ are conjunctions of events.

Definition 8 The sequence of events $\mathcal{E}v$ *satisfies the correspondence* $\psi \Rightarrow \phi$, written $\mathcal{E}v \vdash \psi \Rightarrow \phi$, if and only if for all ρ defined on $\mathrm{var}(\psi)$ such that $\rho, \mathcal{E}v \vdash \psi$, there exists an extension ρ' of ρ to $\mathrm{var}(\phi)$ such that $\rho', \mathcal{E}v \vdash \phi$.

Intuitively, a sequence of events $\mathcal{E}v$ satisfies $\psi \Rightarrow \phi$ when, if $\mathcal{E}v$ satisfies ψ, then $\mathcal{E}v$ satisfies ϕ. The variables of ψ are universally quantified; those of ϕ that do not occur in ψ are existentially quantified.

Definition 9 We define a distinguisher $D(\mathcal{E}v, a) = \mathrm{true}$ if and only if $\mathcal{E}v \vdash \psi \Rightarrow \phi$, and we denote this distinguisher D simply by $\psi \Rightarrow \phi$.

The advantage of the adversary C against the *correspondence* $\psi \Rightarrow \phi$ in process Q is $\mathsf{Adv}_Q^{\psi \Rightarrow \phi}(C) = \Pr[C[Q] : \neg(\psi \Rightarrow \phi)]$, where C is an evaluation context acceptable for Q with any public variables that does not contain events used by $\psi \Rightarrow \phi$.

The process Q *satisfies the correspondence* $\psi \Rightarrow \phi$ with public variables V up to probability p if and only if for all evaluation contexts C acceptable for Q with public variables V that do not contain events used by $\psi \Rightarrow \phi$, $\mathsf{Adv}_Q^{\psi \Rightarrow \phi}(C) \leq p(C)$.

A process satisfies $\psi \Rightarrow \phi$ up to probability p when the probability that it generates a sequence of events $\mathcal{E}v$ that does not satisfy $\psi \Rightarrow \phi$ is at most $p(C)$, in the presence of an adversary represented by the context C. The events used by $\psi \Rightarrow \phi$ are the events that occur in the formula $\psi \Rightarrow \phi$.

Example 2 The correspondence

$$\mathsf{event}(e_B(x, y, z)) \Rightarrow \mathsf{event}(e_A(x, y, z)) \tag{1}$$

means that, with overwhelming probability, for all x, y, z, if $e_B(x, y, z)$ has been executed, then $e_A(x, y, z)$ has been executed.

The correspondence

$$\begin{array}{l}\mathsf{event}(e_1(x)) \wedge \mathsf{event}(e_2(x)) \Rightarrow \\ \quad \mathsf{event}(e_3(x)) \vee (\mathsf{event}(e_4(x, y)) \wedge \mathsf{event}(e_5(y, z)))\end{array}$$

means that, with overwhelming probability, for all x, if $e_1(x)$ and $e_2(x)$ have been executed, then $e_3(x)$ has been executed or there exists y such that both $e_4(x, y)$ and $e_5(x, y)$ have been executed.

Injective Correspondences Injective correspondences are properties of the form "if some event has been executed n times, then some other events have been executed at least n times". In order to model them in our logical formulae, we extend the grammar of formulae ϕ with injective events $\mathsf{inj\text{-}event}(e(M_1, \dots, M_m))$. The formula ψ is a conjunction of (injective or non-injective) events. The conditions on the number of executions of events apply only to injective events.

The definition of formula satisfaction is also extended: we indicate at which step each injective event has been executed, by a "pseudo-formula" ϕ^τ obtained from the formula ϕ by replacing terms and non-injective events with $\bot$ and injective events with the step τ at which they have been executed (that is, their index τ in the sequence of events $\mathcal{E}v$) or $\bot$ when their execution is not required. For example, if $\phi = \mathsf{inj\text{-}event}(e_1(x)) \wedge (\mathsf{inj\text{-}event}(e_2(x)) \vee \mathsf{inj\text{-}event}(e_3(x)))$, then ϕ^τ is of the form $\tau_1 \wedge (\tau_2 \vee \tau_3)$ where τ_1 is the execution step of $e_1(x)$ and either τ_2 is the execution step of $e_2(x)$ or τ_3 is the execution step of $e_3(x)$. (One of the steps τ_2 and τ_3 may be $\bot$, but not both.) We define formula satisfaction $\rho, \mathcal{E}v \vdash^{\phi^\tau} \phi$ as follows:

$\rho, \mathcal{E}v \vdash^{\bot} M$ if and only if $\rho, \emptyset, M, \emptyset, \emptyset \xrightarrow{1}^* \rho, \emptyset, \mathrm{true}, \emptyset, \emptyset$
$\rho, \mathcal{E}v \vdash^{\bot} \mathsf{event}(e(M_1, \dots, M_m))$ if and only if
for all $j \leq m$, $\rho, \emptyset, M_j, \emptyset, \emptyset \xrightarrow{1}^* \rho, \emptyset, a_j, \emptyset, \emptyset$ and $e(a_1, \dots, a_m) \in \mathcal{E}v$
$\rho, \mathcal{E}v \vdash^{\tau} \mathsf{inj\text{-}event}(e(M_1, \dots, M_m))$ if and only if $\tau \neq \bot$,
for all $j \leq m$, $\rho, \emptyset, M_j, \emptyset, \emptyset \xrightarrow{1}^* \rho, \emptyset, a_j, \emptyset, \emptyset$, and $e(a_1, \dots, a_m) = \mathcal{E}v(\tau)$
$\rho, \mathcal{E}v \vdash^{\phi_1^\tau \wedge \phi_2^\tau} \phi_1 \wedge \phi_2$ if and only if $\rho, \mathcal{E}v \vdash^{\phi_1^\tau} \phi_1$ and $\rho, \mathcal{E}v \vdash^{\phi_2^\tau} \phi_2$
$\rho, \mathcal{E}v \vdash^{\phi_1^\tau \vee \phi_2^\tau} \phi_1 \vee \phi_2$ if and only if $\rho, \mathcal{E}v \vdash^{\phi_1^\tau} \phi_1$ or $\rho, \mathcal{E}v \vdash^{\phi_2^\tau} \phi_2$

This definition differs from the case of non-injective correspondences in that we propagate the pseudo-formula ϕ^τ and, in the case of injective events, we make sure that the event has been executed at step τ by requiring that $\tau \neq \bot$ and $e(a_1, \dots, a_m) = \mathcal{E}v(\tau)$.

Given a function $\mathbb{F}$ that maps ψ^τ to ϕ^τ, the *projection* f of $\mathbb{F}$ to the leaf at occurrence o of ϕ is such that $f(\psi^\tau)$ is the leaf at occurrence o of $\mathbb{F}(\psi^\tau)$. For example, if $\mathbb{F}$ maps ψ^τ to ϕ^τ of the form $\tau_1 \wedge (\tau_2 \vee \tau_3)$, then $\mathbb{F}$ has three projections, which map ψ^τ to τ_1, τ_2, and τ_3 respectively. We say that $\mathbb{F}$ is *component-wise injective* when each projection f of $\mathbb{F}$ is such that $f(\psi_1^\tau) = f(\psi_2^\tau) \neq \bot$ implies $\psi_1^\tau = \psi_2^\tau$. (Ignoring the result $\bot$, f is injective.)

Definition 10 The sequence of events $\mathcal{E}v$ *satisfies the correspondence* $\psi \Rightarrow \phi$, written $\mathcal{E}v \vdash \psi \Rightarrow \phi$, if and only if there exists a component-wise injective $\mathbb{F}$ such that for all ρ defined on (ψ), for all ψ^τ such that $\rho, \mathcal{E}v \vdash^{\psi^\tau} \psi$, there exists an extension ρ' of ρ to (ϕ) such that $\rho', \mathcal{E}v \vdash^{\mathbb{F}(\psi^\tau)} \phi$.

Intuitively, a sequence of events $\mathcal{E}v$ satisfies $\psi \Rightarrow \phi$ when, if $\mathcal{E}v$ satisfies ψ with execution steps defined by ψ^τ, then $\mathcal{E}v$ satisfies ϕ with execution steps defined by $\mathbb{F}(\psi^\tau)$. The injectivity is guaranteed because $\mathbb{F}$ is component-wise injective. Definition 9 is unchanged for injective correspondences.

Example 3 The correspondence

$$\mathsf{inj\text{-}event}(e_B(x, y, z)) \Rightarrow \mathsf{inj\text{-}event}(e_A(x, y, z)) \tag{2}$$

means that, with overwhelming probability, each execution of $e_B(x, y, z)$ corresponds to a distinct execution of $e_A(x, y, z)$. In this case, ψ^τ is simply the execution step of $e_B(x, y, z)$ and ϕ^τ the execution step of $e_A(x, y, z)$. The function $\mathbb{F}$ is an injective function that maps the execution step of $e_B(x, y, z)$ to the execution step of $e_A(x, y, z)$. (This step is never $\bot$.)

The correspondence

$$\mathsf{event}(e_1(x)) \wedge \mathsf{inj\text{-}event}(e_2(x)) \Rightarrow \mathsf{inj\text{-}event}(e_3(x)) \vee \\ (\mathsf{inj\text{-}event}(e_4(x, y)) \wedge \mathsf{inj\text{-}event}(e_5(x, y)))$$

means that, with overwhelming probability, for all x, if $e_1(x)$ has been executed, then each execution of $e_2(x)$ corresponds to distinct executions of $e_3(x)$ or to distinct executions of $e_4(x,y)$ and $e_5(x,y)$. The function $\mathbb{F}$ maps $\bot \wedge \tau_2$ to $\tau_3 \vee (\tau_4 \wedge \tau_5)$, where τ_2, τ_3, τ_4, τ_5 are the execution steps of $e_2(x)$, $e_3(x)$, $e_4(x,y)$, $e_5(x,y)$ respectively (either τ_3 or τ_4 and τ_5 may be $\bot$). The projections of $\mathbb{F}$ map $\bot \wedge \tau_2$ to τ_3, τ_4, and τ_5 respectively.

When no injective event occurs in $\psi \Rightarrow \phi$, Definition 10 reduces to the definition of non-injective correspondences.

Property

Lemma 14 *If Q satisfies a correspondence corr with public variables V up to probability p and C is an acceptable evaluation context for Q with public variables V that does not contain events used in corr, then for all $V' \subseteq V \cup \mathrm{var}(C)$, $C[Q]$ satisfies a correspondence corr with public variables V' up to probability p' such that $p'(C') = p(C'[C])$.*

If $Q \approx_p^V Q'$ and Q satisfies a correspondence corr with public variables V up to probability p', then Q' satisfies corr with public variables V up to probability p'' such that $p''(C) = p'(C) + p(C, t_{corr})$.

Proof Suppose that Q satisfies a correspondence *corr* with public variables V and C is an acceptable evaluation context for Q with public variables V that does not contain events used in *corr*. Let $V' \subseteq V \cup \mathrm{var}(C)$. Let C' be an evaluation context acceptable for $C[Q]$ with public variables V' that does not contain events used by *corr*. We rename the variables of C' not in V' so that they are not in V; this renaming does not change the probabilities. We have

$$\mathsf{Adv}^{corr}_{C[Q]}(C') = \Pr[C'[C[Q]] : \neg corr] \leq p(C'[C])$$

because $C'[C]$ is an evaluation context acceptable for Q with public variables V: there is no common table between C and Q, and between C' and $C[Q]$, so a fortiori between C' and Q, so there is no common table between $C'[C]$ and Q; moreover

$$\begin{aligned}
\mathrm{var}(C'[C]) \cap \mathrm{var}(Q) &= ((\mathrm{var}(C') \cap \mathrm{var}(Q)) \cup \mathrm{var}(C)) \cap \mathrm{var}(Q) \\
&\subseteq (V' \cup \mathrm{var}(C)) \cap \mathrm{var}(Q) && \text{since } \mathrm{var}(C') \cap \mathrm{var}(C[Q]) \subseteq V' \\
&\subseteq (V \cup \mathrm{var}(C)) \cap \mathrm{var}(Q) && \text{since } V' \subseteq V \cup \mathrm{var}(C) \\
&\subseteq V && \text{since } \mathrm{var}(C) \cap \mathrm{var}(Q) \subseteq V
\end{aligned}$$

We also have $\mathrm{vardef}(C'[C[\,]]) \cap V = (\mathrm{vardef}(C') \cap V) \cup (\mathrm{vardef}(C) \cap V) = \emptyset$ since $\mathrm{vardef}(C) \cap V = \emptyset$ because C is an acceptable evaluation context for Q with public variables V and $\mathrm{vardef}(C') \cap V \subseteq \mathrm{vardef}(C') \cap V' = \emptyset$ because we have renamed the variables of C' not in V' so that they are not in V and C' is an acceptable evaluation context for $C[Q]$ and with public variables V'.

Suppose that $Q \approx_p^V Q'$ and Q satisfies a correspondence *corr* with public variables V up to probability p'. Let C be an evaluation context acceptable for Q' with public variables V that does not contain events used by *corr*. We have

$$\begin{aligned}
\mathsf{Adv}^{corr}_{Q'}(C) &= \Pr[C[Q'] : \neg corr] \\
&\leq \Pr[C[Q] : \neg corr] + |\Pr[C[Q'] : \neg corr] - \Pr[C[Q] : \neg corr]| \\
&\leq p'(C) + p(C, t_{corr})
\end{aligned}$$

Indeed, by renaming the variables and tables of Q that do not appear in Q' to variables and tables that also do not occur in C, C is also an acceptable evaluation context for Q with public variables V. □

2.7.3 Computation of Advantages

Definition 11 Let C be an evaluation context acceptable for Q with any public variables. We define

$$\mathsf{Adv}_Q(C, D) = \begin{cases} \Pr[C[Q] : D \wedge \neg\mathsf{NonUnique}_Q] & \text{if } \mathsf{S} \text{ does not occur in } D \\ \Pr[C[Q] : D \wedge \neg\mathsf{NonUnique}_Q] - \Pr[C[Q] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}_Q] & \text{if } D = \mathsf{S} \vee D' \end{cases}$$

We assume that D is written as a logical formula (for instance, one of the correspondence formulas defined previously). The phrase "S does not occur in D" means that S occurs in this formula. We consider $\vee$ as commutative and associative, so that $D = \mathsf{S} \vee D'$ means $D = D_1 \vee \ldots \vee D_l$ and $D_j = \mathsf{S}$ for some $j \leq l$. In the following lemmas, the events used by D are the events that occur in this formula.

Lemma 15 *1. In the initial game, if C is an evaluation context acceptable for Q with public variables V that does not contain events used by D, and S does not occur in D, $\Pr[C[Q] : D] = \mathsf{Adv}_Q(C, D)$.*

In the initial game, if C is an evaluation context acceptable for $Q \mid R_x$ with public variables V ($x \notin V$) that does not contain S nor $\overline{\mathsf{S}}$, then $\mathsf{Adv}_Q^{\mathsf{Secrecy}(x)}(C) = \mathsf{Adv}_Q(C[[\,] \mid R_x], \mathsf{S})$.

In the initial game, if C is an evaluation context acceptable for $Q \mid Q_x$ with public variables V ($x \notin V$) that does not contain S nor $\overline{\mathsf{S}}$, then $\mathsf{Adv}_Q^{\mathsf{1-ses.secrecy}(x)}(C) = \mathsf{Adv}_Q(C[[\,] \mid Q_x], \mathsf{S})$.

2. If $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$, S and $\overline{\mathsf{S}}$ are in EvUsed, and C is an evaluation context acceptable for Q and Q' with public variables V that does not contain non-unique events of Q nor events in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$, then

- *if S does not occur in D, $\mathsf{Adv}_Q(C, D) \leq p(C, t_D) + \mathsf{Adv}_{Q'}(C, D \vee D^+)$;*
- *if $D = \mathsf{S} \vee D'$, $\mathsf{Adv}_Q(C, D) \leq 2p(C, t_D) + \mathsf{Adv}_{Q'}(C, D \vee D^+)$.*

3. If C is an evaluation context acceptable for Q with any public variables, then $\mathsf{Adv}_Q(C, D \vee D') \leq \mathsf{Adv}_Q(C, D) + \mathsf{Adv}_Q(C, D')$, when S does not occur both in D and D'.

Proof Property 1: the initial game does not contain $\mathsf{find}[\mathsf{unique}_e]$ (since it does not contain find), so $\mathsf{NonUnique}_Q$ is always false, and $D \wedge \neg\mathsf{NonUnique}_Q = D$. Therefore, we have:

- In case S does not occur in D, $\Pr[C[Q] : D] = \mathsf{Adv}_Q(C, D)$.
- If C is an evaluation context acceptable for $Q \mid R_x$ with public variables V ($x \notin V$) that does not contain S nor $\overline{\mathsf{S}}$, then $\mathsf{Adv}_Q^{\mathsf{Secrecy}(x)}(C) = \Pr[C[Q \mid R_x] : \mathsf{S}] - \Pr[C[Q \mid R_x] : \overline{\mathsf{S}}] = \mathsf{Adv}_Q(C[[\,] \mid R_x], \mathsf{S})$.
- The case of one-session secrecy is similar to the case of secrecy.

Property 2, case S does not occur in D: since $Q, \mathit{EvUsed} \xrightarrow{V}_{p,D^+} Q', \mathit{EvUsed}'$ and C is acceptable for Q and Q' with public variables V and does not contain non-unique events in Q nor events in $\mathit{EvUsed}' \setminus \mathit{EvUsed}$, we have

$$\begin{aligned} \mathsf{Adv}_Q(C, D) &= \Pr[C[Q] : D \wedge \neg\mathsf{NonUnique}_Q] \\ &\leq p(C, t_D) + \Pr[C[Q'] : (D \vee D^+) \wedge \neg\mathsf{NonUnique}_{Q'}] \\ &\leq p(C, t_D) + \mathsf{Adv}_{Q'}(C, D \vee D^+). \end{aligned}$$

since S does not occur in D^+, because $\mathsf{S} \in \mathit{EvUsed}$.

Property 2, case $D = \mathsf{S} \vee D'$: we have

$$\begin{aligned}
\mathsf{Adv}_Q(C, D) &= \Pr[C[Q] : D \wedge \neg\mathsf{NonUnique}_Q] - \Pr[C[Q \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}_Q] \\
&\leq \Pr[C[Q'] : (D \vee D^+) \wedge \neg\mathsf{NonUnique}_Q] + p(C, t_D) \\
&\quad - \Pr[C[Q'] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}_Q] + p(C, t_{\overline{\mathsf{S}}}) \\
&\leq \mathsf{Adv}_{Q'}(C, D \vee D^+) + 2p(C, t_D)
\end{aligned}$$

since $t_{\overline{\mathsf{S}}} \leq t_D$.

Property 3, case S does not occur in D nor D':

$$\begin{aligned}
\mathsf{Adv}_Q(C, D \vee D') &= \Pr[C[Q] : (D \vee D') \wedge \neg\mathsf{NonUnique}_Q] \\
&= \Pr[C[Q] : (D \wedge \neg\mathsf{NonUnique}_Q) \vee (D' \wedge \neg\mathsf{NonUnique}_Q)] \\
&\leq \Pr[C[Q] : D \wedge \neg\mathsf{NonUnique}_Q] + \Pr[C[Q] : D' \wedge \neg\mathsf{NonUnique}_Q] \\
&\leq \mathsf{Adv}_Q(C, D) + \mathsf{Adv}_Q(C, D').
\end{aligned}$$

Property 3, case $D = \mathsf{S} \vee D''$ and S does not occur in D' (the other case is symmetric):

$$\begin{aligned}
\mathsf{Adv}_Q(C, D \vee D') &= \Pr[C[Q] : (D \vee D') \wedge \neg\mathsf{NonUnique}_Q] - \Pr[C[Q] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}_Q] \\
&\leq \Pr[C[Q] : D \wedge \neg\mathsf{NonUnique}_Q] - \Pr[C[Q] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}_Q] \\
&\quad + \Pr[C[Q] : D' \wedge \neg\mathsf{NonUnique}_Q] \\
&\leq \mathsf{Adv}_Q(C, D) + \mathsf{Adv}_Q(C, D')
\end{aligned}$$

□

This lemma allows one to bound the advantage of the adversary against secrecy and correspondences. Property 1 is used in the initial game, to express the desired probability as $\mathsf{Adv}_Q(C, D)$. Property 2 is used when a game Q is transformed into a game Q' during the proof. It allows one to bound the probability in Q from a bound in Q'. Property 3 is useful when distinct sequences of games are used for bounding the probabilities of the disjuncts D and D'. We bound these two probabilities $\mathsf{Adv}_Q(C, D)$ and $\mathsf{Adv}_Q(C, D')$ separately, then obtain a bound on $\mathsf{Adv}_Q(C, D \vee D')$ by taking the sum.

Notice that, in Lemma 15, Property 1, if S does not occur in D, the context C can be any evaluation context acceptable for Q with public variables V that does not contain events used by D (since, in the initial game Q, Q contains no non-unique event at all). In subsequent game transformations, the introduced events and variables can be renamed so that they do not occur in C, by Lemma 12, Property 5. Therefore, Lemma 15 allows one to bound $\Pr[C[Q] : D]$ for any context C allowed in Definition 9. Similarly, for secrecy, C can be any evaluation context acceptable for $Q \mid R_x$ with public variables V ($x \notin V$) that does not contain S nor $\overline{\mathsf{S}}$, so Lemma 15 allows one to bound $\mathsf{Adv}_Q^{\mathsf{Secrecy}(x)}(C)$ for any context C allowed in Definition 7. For one-session secrecy, C can be any evaluation context acceptable for $Q \mid Q_x$ with public variables V ($x \notin V$) that does not contain S nor $\overline{\mathsf{S}}$, so Lemma 15 allows one to bound $\mathsf{Adv}_Q^{\mathsf{1\text{-}ses.secrecy}(x)}(C)$ for any context C allowed in Definition 6.

More formally, consider the following cases:

- If we want to prove that Q_0 satisfies the correspondence $\psi \Rightarrow \phi$ with public variables V, we want to bound the probability $\Pr[C[Q_0] : \neg(\psi \Rightarrow \phi)]$ for any evaluation context C acceptable for Q_0 with public variables V that does not contain the events used by $\neg(\psi \Rightarrow \phi)$. We let $D_0 = \neg(\psi \Rightarrow \phi)$ and $C' = C$ be such a context. The special event S used for testing secrecy is supposed not to occur in D_0. By Lemma 15, Property 1, we have $\mathsf{Adv}_{Q_0}(C', D_0) = \Pr[C[Q_0] : \neg(\psi \Rightarrow \phi)]$.

- If we want prove that Q_0 preserves the secrecy of x with public variables V ($x \notin V$), let C be an evaluation context acceptable for $Q_0 \mid R_x$ with public variables V that does not contain the events $\mathsf{S}, \overline{\mathsf{S}}$. We want to bound the probability $\mathsf{Adv}_{Q_0}^{\mathsf{Secrecy}(x)}(C)$ that C breaks the secrecy of x. We let $D_0 = \mathsf{S}$, $V = \{x\}$, and $C' = C[[\,] \mid R_x]$. By Lemma 15, Property 1, we have $\mathsf{Adv}_{Q_0}(C', D_0) = \mathsf{Adv}_{Q}^{\mathsf{Secrecy}(x)}(C)$.

- The situation for one-session secrecy is similar to the case of secrecy, using Q_x instead of R_x.

The proof produced by CryptoVerif can be represented as a tree whose nodes are labeled by triples (D, Q, EvUsed) and whose edges are labeled by triples (D', p, D^+), where D^+ is a disjunction of Shoup events, and D and D' are disjunctions of Shoup events and possibly the initial formula D_0. The root of the tree is labeled by $(D_0, Q_0, \mathit{EvUsed}_0)$ such that EvUsed_0 is the set containing S, $\overline{\mathsf{S}}$, and the events used by D_0 or that occur in Q_0. When a node labeled by (D, Q, EvUsed) has sons labeled by $(D_1, Q_1, \mathit{EvUsed}_1)$, ..., $(D_l, Q_l, \mathit{EvUsed}_l)$ linked with edges labeled respectively (D'_1, p_1, D^+_1), ..., (D'_l, p_l, D^+_l), then $D = D'_1 \vee \ldots \vee D'_l$ (D is a disjunction of the form $e_1 \vee \ldots \vee e_m$ or $D_0 \vee e_1 \vee \ldots \vee e_m$, $D'_1, \ldots, D'_l$ are disjunctions that form a partition of the disjuncts of D), $D_j = D'_j \vee D^+_j$, and $Q, \mathit{EvUsed} \xrightarrow{V}_{p_j, D^+_j} Q_j, \mathit{EvUsed}_j$ for all $j \leq l$. When the proof is a basic sequence of games, each node has one son, which is the next game in the sequence, except the last game of the sequence which has no son. However, it may happen that distinct sequences of games are used to bound several events occuring in the game; in this case, there is a branching in the proof and a node has several sons. Examples of proof trees can be found in Figure 10; they are explained below.

By Lemma 12, Property 5, we build a similar tree such that, additionally, the events that occur in C' are in EvUsed_0 (by induction from the root to the leaves). We also rename the fresh variables introduced during the game transformations such that they do not occur in C'. Then we show by an easy induction that all nodes of this tree are labeled by (D, Q, EvUsed) such that EvUsed contains S, $\overline{\mathsf{S}}$, and the events that occur in C', and C' is an evaluation context acceptable for Q with public variables V that does not contain non-unique events that occur in Q. We associate to each node labeled by (D, Q, EvUsed) the advantage $\mathsf{Adv}_Q(C', D)$, and we show how to bound these advantages for all nodes in the tree. Suppose that, in this tree, a node labeled by (D, Q, EvUsed) has sons labeled by $(D_1, Q_1, \mathit{EvUsed}_1)$, ..., $(D_l, Q_l, \mathit{EvUsed}_l)$ linked with edges labeled respectively (D'_1, p_1, D^+_1), ..., (D'_l, p_l, D^+_l). Then we have

$$\mathsf{Adv}_Q(C', D) \leq \sum_{j=1}^{l} \mathsf{Adv}_Q(C', D'_j)$$

by Lemma 15, Property 3, since $D = D'_1 \vee \ldots \vee D'_l$. Moreover, if S does not occur in D, for all $j \leq l$,

$$\mathsf{Adv}_Q(C', D'_j) \leq p_j(C', t_{D'_j}) + \mathsf{Adv}_{Q_j}(C', D'_j \vee D^+_j) = p_j(C', t_{D'_j}) + \mathsf{Adv}_{Q_j}(C', D_j)$$

by Lemma 15, Property 2, since $Q, \mathit{EvUsed} \xrightarrow{V}_{p_j, D^+_j} Q_j, \mathit{EvUsed}_j$. Therefore,

$$\mathsf{Adv}_Q(C', D) \leq \sum_{j=1}^{l} \left(p_j(C', t_{D'_j}) + \mathsf{Adv}_{Q_j}(C', D_j) \right)$$

If $D = \mathsf{S} \vee D'$, then the event S occurs in exactly one formula $D'_1, \ldots, D'_l$, say in D'_1. We have

$$\mathsf{Adv}_Q(C', D'_1) \leq 2p_1(C', t_{D'_1}) + \mathsf{Adv}_{Q_1}(C', D'_1 \vee D^+_1) \leq 2p_1(C', t_{D'_1}) + \mathsf{Adv}_{Q_1}(C', D_1)$$
$$\mathsf{Adv}_Q(C', D'_j) \leq p_j(C', t_{D'_j}) + \mathsf{Adv}_{Q_j}(C', D'_j \vee D^+_j) = p_j(C', t_{D'_j}) + \mathsf{Adv}_{Q_j}(C', D_j) \text{ for } j \geq 2$$

$$\begin{tikzcd}
\boxed{e_0, G_0, \{e_0\}} \arrow[d, "{e_0, 0, e}"] \\
\boxed{e_0 \vee e, G_1, \{e_0, e\}} \arrow[d, "{e_0 \vee e, p, D_{\mathrm{false}}}"] \\
\boxed{e_0 \vee e, G_2, \{e_0, e\}}
\end{tikzcd}$$

(a) linear sequence

$$\begin{tikzcd}
& \boxed{e_0, G_0, \{e_0\}} \arrow[d, "{e_0, 0, e}"] & \\
& \boxed{e_0 \vee e, G_1, \{e_0, e\}} \arrow[d, "{e_0 \vee e, p, D_{\mathrm{false}}}"] & \\
& \boxed{e_0 \vee e, G_2, \{e_0, e\}} \arrow[dl, "{e_0, p_3, D_{\mathrm{false}}}"'] \arrow[dr, "{e, p_4, D_{\mathrm{false}}}"] & \\
\boxed{e_0, G_3, \{e_0, e\}} & & \boxed{e, G_4, \{e_0, e\}}
\end{tikzcd}$$

(b) example with branching

Figure 10: Examples of proof trees

by Lemma 15, Property 2, since $Q, \mathit{EvUsed} \xrightarrow{V}_{p_j, D_j^+} Q_j, \mathit{EvUsed}_j$. Therefore,

$$\mathsf{Adv}_Q(C', D) \leq 2p_1(C', t_{D'_1}) + \sum_{j=2}^{l} p_j(C', t_{D'_j}) + \sum_{j=1}^{l} \mathsf{Adv}_{Q_j}(C', D_j)$$

In all cases, we can then bound the advantage associated to a node from bounds on the advantages associated to its sons. Assuming that the advantage associated to the leaves of the tree is bounded, we can then bound the advantage associated to all nodes of the tree, by induction from the leaves to the root. At the root, we obtain a bound on $\mathsf{Adv}_{Q_0}(C', D_0)$ which yields the desired result.

Lemma 15 allows us to obtain more precise probability bounds than the standard computation of probabilities generally done by cryptographers, when we use Shoup's lemma [11]. By Shoup's lemma, if G' is obtained from G by inserting an event e and modifying the code executed after e, the probability of distinguishing G' from G is bounded by the probability of executing e: for all contexts C acceptable for G and G' (with any public variables) and all distinguishers D, $|\Pr[C[G] : D] - \Pr[C[G'] : D]| \leq \Pr[C[G'] : e]$. Hence,

$$\Pr[C[G] : D] \leq \Pr[C[G'] : e] + \Pr[C[G'] : D].$$

We improve over this computation of probabilities by considering e and D simultaneously instead of making the sum of the two probabilities:

$$\Pr[C[G] : D] \leq \Pr[C[G'] : D \vee e].$$

For example, suppose that we want to bound the probability of event e_0 in G_0; we transform G_0 into G_1 using Shoup's lemma, so that G_1 differs from G_0 only when G_1 executes event e, and we have $G_0, \mathit{EvUsed}_0 \rightarrow_{0,e} G_1, \mathit{EvUsed}_1$; then we transform G_1 into G_2, so that $G_1 \approx_p G_2$, and $G_1, \mathit{EvUsed}_1 \rightarrow_{p, D_{\mathrm{false}}} G_2, \mathit{EvUsed}_1$; and G_2 executes neither e_0 nor e. The corresponding proof tree is given in Figure 10(a). Let C be an evaluation context acceptable for G_0 without public variables that does not contain event e_0. Lemma 15 yields

$$\begin{aligned}
\Pr[C[G_0] : e_0] &= \mathsf{Adv}_{G_0}(C, e_0) \\
&\leq \mathsf{Adv}_{G_1}(C, e_0 \vee e) \\
&\leq p(C, t_{e_0 \vee e}) + \mathsf{Adv}_{G_2}(C, e_0 \vee e) = p(C, t_{e_0 \vee e})
\end{aligned}$$

If we suppose for simplicity that no $\mathsf{find}[\mathsf{unique}]$ occurs, so that $\mathsf{NonUnique}_{G_i}$ is always false, we have $\mathsf{Adv}_{G_i}(C, D) = \Pr[C[G_i] : D]$, so we can write the previous formulas simply using probabilities:

$$\Pr[C[G_0] : e_0] \leq \Pr[C[G_1] : e_0 \vee e] \leq p(C, t_{e_0 \vee e}) + \Pr[C[G_2] : e_0 \vee e] = p(C, t_{e_0 \vee e}).$$

In contrast, the standard computation of probabilities yields

$$\Pr[C[G_0] : e_0] \leq \Pr[C[G_1] : e_0] + \Pr[C[G_1] : e] \leq p(C, t_{e_0}) + p(C, t_e).$$

The runtime t_D of D is essentially the same for e_0, e, and $e_0 \vee e$, so $\Pr[C[G_0] : e_0] \leq p(C, t_D)$ by Lemma 15, while $\Pr[C[G_0] : e_0] \leq 2p(C, t_D)$ by the standard computation, so we have gained a factor 2. The probability that comes from the transformation of G_1 into G_2 is counted once (for distinguisher $e_0 \vee e$) instead of counting it twice (once for e_0 and once for e).

The standard computation of probabilities corresponds to applying point 3 of Lemma 15 to bound each probability separately and compute the sum, as soon as the considered distinguisher D has several disjuncts. Instead, we use point 3 of Lemma 15 only when the proof uses different sequences of games to bound the probabilities of the events, as in Figure 10(b).

2.8 Turing Machine Adversary

In CryptoVerif, the adversary is modeled as an evaluation context. However, usually, in cryptographic results, an adversary is a bounded-time probabilistic Turing machine. In this section, we explain how any bounded-time probabilistic Turing machine that communicates on channels can be represented as a CryptoVerif evaluation context.

Let Q_0 be the initial game that interacts with an adversary. Let $c_1, \ldots, c_k$ be the channels used in Q_0. Let T_{all} be the union of all types that occur in Q_0. Let T'_{all} be the type of pairs containing the encoding a channel as first component and an element of T_{all} as second component. The encoding of a channel is either the constant *yield* or a tuple of integers $(j, i_1, \ldots, i_{k'})$ with $1 \leq j \leq k$. (We assume that unambiguous tuples can be encoded as CryptoVerif values, and that the constant *yield* is different from a tuple.) Let d_0, d_1, and d_2 be channels that do not occur in Q_0.

Let Q_1 be a process that contains the parallel composition of processes

$$!^{i_1 \leq n_1} \ldots !^{i_{k'} \leq n_{k'}} c_j[i_1, \ldots, i_{k'}](x : T_{\mathsf{all}}).\overline{d_0}\langle((j, i_1, \ldots, i_{k'}), x)\rangle$$

for each output $\overline{c_j[i'_1, \ldots, i'_{k'}]}\langle N\rangle$ that occurs under $!^{i'_1 \leq n_1} \ldots !^{i'_{k'} \leq n_{k'}}$ in Q_0. Since, in the initial game Q_0, the channels of all outputs use the current replication indices as channel indices, as in $c_j[i'_1, \ldots, i'_{k'}]$, a single output is executed for each value of the indices and for each syntactic occurrence of the output, so the inputs in Q_1 can receive all outputs made by Q_0. The process Q_1 forwards all these outputs to the same channel d_0, with a message that specifies both the channel $c_j[i_1, \ldots, i_{k'}]$ on which Q_0 emitted (encoded as a bitstring) and the message x sent by Q_0.

In addition, Q_1 also contains the parallel composition of processes

$$!^{i_1 \leq n_1} \ldots !^{i_{k'} \leq n_{k'}} \mathit{yield}().\overline{d_0}\langle(\mathit{yield}, ())\rangle$$

for each occurrence of yield that occurs under $!^{i'_1 \leq n_1} \ldots !^{i'_{k'} \leq n_{k'}}$ in Q_0, to receive all outputs that come from the yield construct.

Let $C = \mathsf{newChannel}\ d_0; \mathsf{newChannel}\ d_1; \mathsf{newChannel}\ d_2; (\mathit{start}().\overline{d_1[1]}\langle s_0\rangle \mid Q_1 \mid Q_2 \mid [\,])$, where the process Q_2 is defined in Figure 11. Let us explain how the context C can simulate any Turing machine interacting with the process Q_0.

The current state of the Turing machine is sent on channel $d_1[i]$ where i is a loop index that starts at 1 and increases during execution. As shown in the semantics of CryptoVerif,

```
1     $Q_2 = !^{i \leq n} d_1[i](s : \mathit{bitstring});$
2         let $(s', o, v) = f(s)$ in
              *Lines 3–6 are repeated for each $j \leq k$ and each $k'$*
              *such that there is an input on channel $c_j[i'_1, \ldots, i'_{k'}]$ in $Q_0$.*
3         if $o = (j, k')$ then
4             let $(a_1, \ldots, a_{k'}, b) = v$ in $\overline{c_j[a_1, \ldots, a_{k'}]}\langle b \rangle;$
5             $d_0(s'' : T'_{\mathsf{all}}); \overline{d_1[i+1]}\langle f'(s', s'') \rangle$
6         else

7         if $o = \mathit{random}$ then
8             new $x : \mathit{bool}; \overline{d_1[i+1]}\langle f''(s', x) \rangle$
9         else
10        if $o = \mathit{abort}$ then
11            event_abort $e$
12        else
13            $\overline{d_2}\langle\rangle$
```

Figure 11: Looping process

upon startup, a message is sent on channel *start*. When C receives that message, it sends the initial state of the Turing machine s_0 on channel $d_1[1]$. This message is received by process Q_2 (line 1). Then Q_2 calls the function f on the current state s of the Turing machine (line 2). This function executes the Turing machine, until one of the following situations happens:

- The Turing machine sends a message b on a channel $c_j[a_1, \ldots, a_{k'}]$; in this case, f returns $(s', (j, k'), (a_1, \ldots, a_{k'}, b))$, where s' is the new state of the Turing machine. The test at line 3 is then going to succeed for the appropriate value of j, k', and the desired message is going to be sent at line 4. After receiving a message, the process Q_0 always replies by sending a message (except if it aborts). This message is going to be received by Q_1, which is going to forward on d_0 the channel and the received message. These channel and message are then received as s'' at line 5. Then $f'(s', s'')$ is the new state of the Turing machine after receiving that message. This state is sent on channel $d_1[i+1]$, which restarts a new iteration of Q_2.

- The Turing machine generates a fresh random bit; in this case, f returns $(s', \mathit{random}, ())$ where s' is the new state of the Turing machine. The test at line 7 is then going to succeed. At line 8, a random bit x is chosen. Then $f''(s', x)$ is the new state of the Turing machine with that random bit. This state is sent on channel $d_1[i+1]$, which restarts a new iteration of Q_2 as in the previous case.

- The Turing machine aborts; in this case, f returns $(s', \mathit{abort}, ())$. The test at line 10 is then going to succeed, and the process aborts at line 11. (The event e is any event not used elsewhere; the event is not really useful, it is present because the CryptoVerif language always executes an event before aborting.)

- The Turing machine stops; in this case, f returns $(s', \mathit{stop}, ())$. No test succeeds, so line 13 is executed. The process tries to send a message on channel d_2, but there is no input on this channel, so the process blocks.

The constants *random*, *abort*, and *stop* are assumed to be pairwise distinct, and distinct from all pairs.

The function f is a CryptoVerif primitive, because it can be implemented by a deterministic bounded-time Turing machine. (Recall that f stops when the initial probabilistic Turing machine makes a random choice, and the random choice is performed by CryptoVerif at lines 7–8.) Similarly, the function f' that computes the new state of the Turing machine from the old state and the received message, and the function f'' that computes the new state of the Turing machine from the old state and a random bit are CryptoVerif primitives.

The replication bound n (used in Q_2, line 1) is chosen large enough so that the loop never stops due to that bound: the Turing machine aborts or stops before the bound is reached. This is possible since the Turing machine runs in bounded time, so sends a bounded number of messages and chooses a bounded number of random bits.

Notice that, if Q_0 sends and receives messages on the same channels, it may happen that a message sent by Q_0 is immediately received by Q_0 without being intercepted by the adversary. In this case, since both Q_0 and Q_1 are going to listen on the same channels, the destination of the message (either the honest process Q_0 or the adversary Q_1) is chosen randomly with uniform probability, depending on the number of available receivers. Therefore, adding more copies of the receiving processes in Q_1 increases the probability that the adversary receives the message. Moreover, when the same channel is used for both inputs and outputs, the messages sent by Q_2 at line 4 may be received back by the adversary via Q_1, instead of being received by Q_0. We recommend avoiding this strange situation, by using distinct channels for inputs on the one hand and outputs on the other hand. More generally, we recommend using distinct channels for each input and output, so that the adversary gets full control of the network, as already mentioned page 7.

As a slight extension, it would still be possible to allow Q_0 to output on $c_j[i_1, \dots, i_{k'}]$ after receiving a message on the same channel $c_j[i_1, \dots, i_{k'}]$. In this case, a message sent by Q_0 on $c_j[i_1, \dots, i_{k'}]$ cannot be received by Q_0, because the input on $c_j[i_1, \dots, i_{k'}]$ is no longer available when the output on $c_j[i_1, \dots, i_{k'}]$ is performed by Q_0. Moreover, the problem that messages sent by Q_2 at line 4 may be received back by the adversary via Q_1, instead of being received by Q_0, can be avoided by putting the receiver process

$$c_j[a_1, \dots, a_{k'}](x : T_{\mathsf{all}}).\overline{d_0}\langle((j, a_1, \dots, a_{k'}), x)\rangle$$

after $\overline{c_j[a_1, \dots, a_{k'}]}\langle b\rangle$ in parallel with $d_0(s'' : T'_{\mathsf{all}}); \overline{d_1[i+1]}\langle f'(s', s'')\rangle$ in Q_2, instead of including

$$!^{i_1 \leq n_1} \dots !^{i_{k'} \leq n_{k'}} c_j[i_1, \dots, i_{k'}](x : T_{\mathsf{all}}).\overline{d_0}\langle((j, i_1, \dots, i_{k'}), x)\rangle$$

in Q_1.

The context C does not allow the Turing machine to execute events of its choice, while a CryptoVerif context can execute events. We could obviously extend the model to allow the Turing machine to execute events, but this is not needed for the cases we consider. Indeeed, if the adversary represented as CryptoVerif context executes events, these events can be deleted without changing the final result returned by the distinguisher: for correspondences, by Definition 9, the context is not allowed to contain events used by $\psi \Rightarrow \phi$, and all other events are ignored by the distinguisher $\neg(\psi \Rightarrow \phi)$; for one-session secrecy and secrecy, by Definitions 6 and 7, the context is not allowed to contain S nor $\overline{\mathsf{S}}$, and all other events are ignored by the distinguishers S and $\overline{\mathsf{S}}$.

To sum up, the context given in this section allows us to run any probabilistic bounded-time Turing machine as a CryptoVerif context, so CryptoVerif contexts are powerful enough to represent the adversaries usually considered by cryptographers.

Acknowledgments I warmly thank David Pointcheval for his advice and explanations of the computational proofs of protocols. This project would not have been possible without him. I also thank Jacques Stern for initiating this work. The design and implementation of CryptoVerif was partly done while I was at CNRS and at Ecole Normale Supérieure. It was partly supported by the ANR through the projects FormaCrypt (ARA SSIA 2005) and ProSe (VERSO 2010, decision number 2010-VERS-004).

References

[1] M. Abdalla, P.-A. Fouque, and D. Pointcheval. Password-based authenticated key exchange in the three-party setting. *IEE Proceedings Information Security*, 153(1):27–39, Mar. 2006.

[2] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT'00*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545, Berlin, Heidelberg, Dec. 2000. Springer.

[3] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *Advances in Cryptology – Eurocrypt 2006 Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, Berlin, Heidelberg, May 2006. Springer. Extended version available at `http://eprint.iacr.org/2004/331`.

[4] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 97–111, Los Alamitos, CA, July 2007. IEEE Computer Society Press. Extended version available as ePrint Report 2007/128, `http://eprint.iacr.org/2007/128`.

[5] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, Oct.–Dec. 2008.

[6] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations. In *7th International Conference on Availability, Reliability and Security (AReS 2012)*, pages 65–74, Los Alamitos, CA, Aug. 2012. IEEE Computer Society Press.

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

[8] J. C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 353(1–3):118–164, Mar. 2006.

[9] V. Shoup. A proposal for an ISO standard for public-key encryption, Dec. 2001. ISO/IEC JTC 1/SC27.

[10] V. Shoup. OAEP reconsidered. *Journal of Cryptology*, 15(4):223–249, Sept. 2002.

[11] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, Nov. 2004. Available at `http://eprint.iacr.org/2004/332`.