Proved Generation of Implementations from Computationally Secure Protocol Specifications

David Cadé and Bruno Blanchet

INRIA Paris-Rocquencourt, France
{david.cade,bruno.blanchet}@inria.fr

Abstract. In order to obtain implementations of security protocols proved secure in the computational model, we have previously implemented a compiler that takes a specification of the protocol in the input language of the computational protocol verifier CryptoVerif and translates it into an OCaml implementation. However, until now, this compiler was not proved correct, so we did not have real guarantees on the generated implementation. In this paper, we fill this gap. We prove that this compiler preserves the security properties proved by CryptoVerif: if an adversary has probability p of breaking a security property in the generated code, then there exists an adversary that breaks the property with the same probability p in the CryptoVerif specification. Therefore, if the protocol specification is proved secure in the computational model by CryptoVerif, then the generated implementation is also secure.

1 Introduction

The verification of security protocols is an important research area since the 1990s: the design of security protocols is notoriously error-prone, and errors can have serious consequences. Formal verification first focused on verifying formal specifications of protocols. However, verifying a specification does not guarantee that the protocol is correctly implemented from this specification. It is therefore important to make sure that the implementation is secure, and not only the specification. Moreover, two models were considered for verifying protocols. In the symbolic model, the so-called Dolev-Yao model, messages are terms. This abstract model facilitates automatic proofs. On the other hand, in the computational model, typically used by cryptographers, messages are bitstrings and attackers are polynomial-time probabilistic Turing machines. Proofs in the latter model are more difficult than in the former, but yield a much more precise analysis of the protocol. Therefore, we would like to obtain implementations of protocols proved secure in the computational model.

To reach this goal, we have proposed the following approach in [7]. We start from a formal specification of the protocol. In order to prove the specified protocol secure in the computational model, we rely on the automatic protocol verifier CryptoVerif [4, 5]. This verifier can prove secrecy and authentication properties. The generated proofs are proofs by sequences of games, like the manual proofs written by cryptographers. These games are formalized in a probabilistic process calculus. In order to obtain a proved implementation from the specification, we have written a compiler that takes a CryptoVerif specification and returns an implementation in the functional language OCaml (http://caml.inria.fr). This compiler starts from a CryptoVerif specification annotated with implementation details: the annotations specify how to divide the protocol in different roles, for example, key generation, server, and client, and how to implement the various cryptographic primitives and types. The compiler then generates an OCaml module for each role in the input file. In order to get a full implementation of the protocol, this module is combined with manually written network code, responsible in particular for sending and receiving messages from the network. From the point of view of security, the network code can be considered as part of the adversary, so we do not need to prove its security.

To make sure that the generated implementation is actually secure, we need to prove the correctness of our compiler. This proof was still missing in [7]. It is the topic of this paper. To make this proof, we needed a formal semantics of OCaml. We adapted the operational small-step semantics of a core part of OCaml by Owens et al. [13]. We added to this language support for simplified modules, multiple threads where only one thread can run at any given time, and communication between threads by a shared part of the store.

An adversary against the generated implementation is an OCaml program using the modules generated by our compiler. On the CryptoVerif side, an adversary is a process running in parallel with the verified protocol. In our proof, for each OCaml adversary, we construct a corresponding CryptoVerif adversary that simulates the behavior of the OCaml adversary. When the OCaml adversary calls one of the functions generated by our compiler, which comes from an oracle in the CryptoVerif process, the CryptoVerif adversary calls this oracle. Then we establish a precise correspondence between the traces of the CryptoVerif process with that CryptoVerif adversary and the traces of the OCaml program. This correspondence allows us to show that the probability of success of an attack is the same on the CryptoVerif side and on the OCaml side. Therefore, if CryptoVerif proves that the protocol is secure, then the generated OCaml implementation is also secure, and the bound on the probability of success of an attack computed by CryptoVerif is also valid for the implementation.

We have made several assumptions to get this proof; the important ones are:

A1. The cryptographic primitives are correct with respect to the assumptions made on them in the specification.

A2. The roles are executed in the order specified in CryptoVerif (e.g., in a key-exchange protocol, the key generation is called before the servers and clients).

A3. The adversary and the network code do not access files created by our implementation (e.g. private key files).

A4. The network code is a well-typed OCaml program, which does not use unsafe OCaml functions to bypass the type system.

A5. We represent bitstrings by the OCaml type `string`. We assume that the network code does not mutate `string`s passed to or received from generated code. This could be guaranteed by using an abstract type instead of `string`. In our semantics, `string`s are immutable values.

A6. Our semantics of threads is obeyed, which implies that two processes that read or write the same file are not run concurrently (which can be enforced using locks), and that one cannot fork in the middle of a role.

Related work. Several approaches have been considered in order to obtain proved implementations of security protocols. In the symbolic model, several approaches generate protocols from specifications, e.g. [12, 14]. Other approaches analyze implementations by extracting a specification verified by a symbolic protocol verifier, e.g. [3, 1], or analyze them by other tools such as the model-checker ASPIER [8], the general-purpose C verifier VCC [9] or typing [15].

In contrast, the following approaches provide computational security guarantees, by analyzing implementations. The tool FS2CV [11] translates a subset of F# to the input language of CryptoVerif, which can then prove the protocol secure. The tool F7, which uses a dependent type system to prove security properties on protocols implemented in F#, has been adapted to the computational model in [10]; it uses type annotations to help the proof. The symbolic execution approach of [1] provides computational security guarantees by applying a computational soundness result, which however restricts the class of protocols that can be considered. The tool of [2] generates a CryptoVerif model from a C implementation; however, it can analyze only a single execution path.

To the best of our knowledge, our approach is the first one for generating implementations with a computational proof. The work of [2] and ours are the only ones to provide an explicit bound on the probability of success of an attack against the verified protocol implementation.

Outline. Section 2 describes the common input language of CryptoVerif and of our compiler. Section 3 describes OCaml, the output language of our compiler. Section 4 describes the compiler itself. Sections 5 to 8 present our proof. The long version [6] provides additional details on the semantics of the CryptoVerif input language, on the compiler, and on the proof.

2 The CryptoVerif Input Language

This section presents the syntax and semantics of the CryptoVerif input language, as well as the annotations that specify implementation details.

Syntax and Informal Semantics. Let us first introduce the syntax of the CryptoVerif language in Fig. 1. The language is typed, and types T are subsets of $\mathit{bitstring}_\perp = \mathit{bitstring} \cup \{\perp\}$ where *bitstring* is the set of all bitstrings and $\perp$ is a symbol that is not a bitstring, used, for example, to represent the failure of a decryption. The boolean type $\mathit{bool} = \{\mathrm{true}, \mathrm{false}\}$ with true being the bitstring 1 and false 0, *bitstring*, and $\mathit{bitstring}_\perp$ are predefined.

Variables $x[i_1, \dots, i_m]$ represent arrays of bitstrings of a given type T indexed by the values of the indices i of the replications $\mathsf{foreach}\ i \leq N\ \mathsf{do}\ Q$ present above the definition of the variable. We call these indices *replication indices*, and we

$$
\begin{aligned}
M &::= x[i_1,\dots,i_m] \quad | \quad f(M_1,\dots,M_m) && \text{(Terms)}\\
Q &::= 0 \quad | \quad Q \mid Q' \quad | \quad \mathsf{foreach}\ i \le N\ \mathsf{do}\ Q \quad | \quad O[\widetilde{i}](x_1[\widetilde{i}],\dots,x_k[\widetilde{i}]) := P\\
&&& \text{(Oracle definitions)}\\
P &::= \mathsf{return}(M_1,\dots,M_k); Q \quad | \quad \mathsf{end} \quad | \quad x[i_1,\dots,i_m] \overset{R}{\leftarrow} T; P\\
&\quad | \quad x[i_1,\dots,i_m] \leftarrow M; P \quad | \quad \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P' \quad | \quad \mathsf{event}\ e(M_1,\dots,M_k); P\\
&\quad | \quad \mathsf{insert}\ \mathit{Tbl}(M_1,\dots,M_k); P\\
&\quad | \quad \mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}],\dots,x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'\\
&\quad | \quad \mathsf{let}\ (x_1[\widetilde{i}],\dots,x_{k'}[\widetilde{i}]) = O[M_1,\dots,M_l](M'_1,\dots,M'_k)\ \mathsf{in}\ P\ \mathsf{else}\ P'\\
&\quad | \quad \mathsf{let}\ x[\widetilde{i}] = \mathsf{loop}\ O[M_1,\dots,M_n](M')\ \mathsf{in}\ P\ \mathsf{else}\ P' && \text{(Oracle bodies)}
\end{aligned}
$$

Fig. 1. Syntax of the CryptoVerif language

abbreviate $i_1,\dots,i_m$ by $\widetilde{i}$. Each function f comes with its type $T_1 \times \dots \times T_m \to T$; all CryptoVerif functions are deterministic and efficiently computable. Some functions are predefined, and some are infix, like the equality test $=$ and boolean operations. The cryptographic primitives used in the protocol are represented by CryptoVerif functions. Terms M represent computations over bitstrings: they can be variable accesses $x[i_1,\dots,i_m]$ or function applications $f(M_1,\dots,M_m)$.

The oracle definitions Q represent the oracles that will become available to the adversary at this point. The nil construct 0 provides no oracle. The parallel composition $Q \mid Q'$ provides oracles in Q and Q'. The replication $\mathsf{foreach}\ i \le N\ \mathsf{do}\ Q$ provides N copies of Q, indexed by $i \in \{1,\dots,N\}$. The parameter N is unspecified and is used by CryptoVerif to express the maximum probability of breaking the protocol, which typically depends on the number of calls to the various oracles. The oracle definition $O[\widetilde{i}](x_1[\widetilde{i}] : T_1,\dots,x_k[\widetilde{i}] : T_k) := P$ makes available the oracle $O[\widetilde{i}]$, and when called by the adversary with arguments $a_1,\dots,a_k$, it executes the oracle body P with $x_j[\widetilde{i}]$ set to a_j.

The oracle bodies P represent the behavior of the oracle. A return statement $\mathsf{return}(M_1,\dots,M_k); Q$ returns the result of $M_1,\dots,M_k$ to the caller, and makes available oracles in Q. An end statement end returns to the caller on an error. A random number assignment $x[\widetilde{i}] \overset{R}{\leftarrow} T; P$ puts a uniformly chosen random value of type T in variable $x[\widetilde{i}]$, and continues with P. The type T must consist of all bitstrings of a given size. An assignment $x[\widetilde{i}] \leftarrow M; P$ puts the result of M in the variable $x[\widetilde{i}]$, and continues with P. A conditional statement $\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P'$ executes P if M evaluates to true and P' otherwise.

An insert statement $\mathsf{insert}\ \mathit{Tbl}(M_1,\dots,M_k); P$ inserts the result of $M_1,\dots,$ M_k into the table Tbl. Tables are lists of tuples, used for example to store tables of keys. A get statement $\mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}],\dots,x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$ searches for an element $a_1,\dots,a_k$ in the table Tbl such that the term M evaluates to true when $x_1[\widetilde{i}] = a_1,\dots,x_k[\widetilde{i}] = a_k$. If there is no such element, we continue with P', and otherwise we choose randomly one of the elements that correspond, store it in the variables $x_1[\widetilde{i}], \dots, x_k[\widetilde{i}]$, then continue with P. An event statement event $e(M_1, \dots, M_k); P$ is used to log events. Events serve for specifying security properties of protocols, but do not change the execution of the process.

An oracle call let $(x_1[\widetilde{i}], \dots, x_{k'}[\widetilde{i}]) = O[M_1, \dots, M_l](M'_1, \dots, M'_k)$ in P else P' calls oracle $O[M_1, \dots, M_l]$, stores its returned values in the variables $x_1[\widetilde{i}]$, $\dots, x_{k'}[\widetilde{i}]$, and continues with P if the oracle terminates with a return statement, and continues with P' if the oracle terminates with end.

A loop let $x[\widetilde{i}] =$ loop $O[M_1, \dots, M_n](M')$ in P else P' calls oracle O in a loop. Oracle O takes a unique argument (the internal state of the loop) and returns a pair containing a result of the same type and a boolean indicating whether the loop should continue or not. $O[M_1, \dots, M_n](M')$ is first called. If it returns (a_1, true), $O[M_1 + 1, M_2, \dots, M_n](a_1)$ is called. If it returns (a_2, true), $O[M_1 + 2, M_2, \dots, M_n](a_2)$ is called, and so on, until $O[M_1 + k, M_2, \dots, M_n](a_k)$ returns $(a_{k+1}, \mathrm{false})$. Then we run P with $x[\widetilde{i}]$ set to a_{k+1}. If O terminates with end, we run P'. Oracle call and loop statements cannot appear in the CryptoVerif process representing the protocol, but are used for representing the adversary.

Formal Semantics. The complete formal semantics of the language can be found in [6]. The semantics is defined as a reduction relation on semantic configurations, which are tuples of the form $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$. The environment E is a mapping from variables with their replication indices to bitstring values. The oracle body P is the oracle body currently running. The mapping $\mathcal{T}$ maps table names to their contents, which is the multiset of elements inserted in the table. The set $\mathcal{Q}$ contains the set of the callable oracle definitions. The list $\mathcal{R}$ is the call stack, which consists of triplets containing the variables with which the result should be bound and two oracle bodies, the first will be executed if the oracle returns a result with a return statement, and the second will be executed if the oracle returns on an end statement. The sequence $\mathcal{E}$ is the list of events $e(a_1, \dots, a_k)$ executed so far, by the construct event $e(M_1, \dots, M_k)$.

The notation $E, M \Downarrow a$ means that the term M evaluates to the bitstring a under the environment E. We say that an oracle definition $O[\widetilde{i}](x_1[\widetilde{i}], \dots, x_k[\widetilde{i}])$ $:= P$ is defined *at the beginning of* Q when this oracle definition is present in Q without entering into oracle bodies. The list $\mathrm{reduce}'(Q) = [(Q_1, b_1), \dots, (Q_n, b_n)]$ contains all oracle definitions at the beginning of Q, ordered from left to right, with the boolean b_i to true if Q_i is under a replication in Q, and false otherwise. The set $\mathrm{reduce}(Q)$ contains all oracle definitions present in $\mathrm{reduce}'(Q)$. The semantics is probabilistic: $\mathfrak{C} \rightarrow_p \mathfrak{C}'$ means that $\mathfrak{C}$ reduces into $\mathfrak{C}'$ with probability p. The initial configuration for running the oracle definition Q is $\mathfrak{C}_{\mathrm{i}}(Q) = \emptyset, \mathsf{let}\ x = O_{\mathrm{start}}()\ \mathsf{in}\ \mathsf{return}(x)\ \mathsf{else}\ \mathsf{end}, \emptyset, \mathrm{reduce}(Q), \emptyset, \emptyset$, which starts by calling oracle O_{start}. The oracle definition Q typically contains a protocol in parallel with an adversary.

Annotations. In order to compile a CryptoVerif process into an implementation, we added annotations to the language, to specify implementation details.

First, we separate the parts of the process that correspond to different roles, such as client and server, which will be included in different OCaml programs in the generated implementation. We annotate processes to specify roles: the beginning of *role* is specified in oracle definitions $role\{Q$; the end of *role* is specified by a closing brace $\}$ between a $\mathsf{return}(\ldots)$ and its following oracle definition Q. We denote by $Q(role)$ the part of the process corresponding to the role *role*.

The process for a role $Q(role)$ may have free variables, but CryptoVerif requires that these free variables be defined under no replication, so that they can be passed from the process that defines them to the process $Q(role)$, which uses them, simply by storing each variable in a file. (There must be a single value to store, not one for each value of the replication indices.) The user must also declare, for each free variable $x[\,]$ in a role, the file f in which the variable will be stored. Let files be the set of these pairs $(x[\,], f)$. Let also tables be the set of pairs (Tbl, f) such that the table Tbl will be stored in file f.

Finally, the user annotations provide, for each CryptoVerif type T, the corresponding OCaml type $\mathbb{G}_T(T)$ as well as several OCaml functions: a function $\mathbb{G}_{\mathsf{random}}(T) : \mathsf{unit} \rightarrow \mathbb{G}_T(T)$ that generates random numbers uniformly in T (when T is used in a random number generation), serialization and deserialization functions $\mathbb{G}_{\mathsf{ser}}(T) : \mathbb{G}_T(T) \rightarrow \mathsf{string}$ and $\mathbb{G}_{\mathsf{deser}}(T) : \mathsf{string} \rightarrow \mathbb{G}_T(T)$ (when T is written or read from tables and files), and a predicate function $\mathbb{G}_{\mathsf{pred}}(T) : \mathbb{G}_T(T) \rightarrow \mathsf{bool}$ that returns true if its argument corresponds to an element of type T and false otherwise (when T is present in the interface of the oracle definitions). The user annotations also provide, for each CryptoVerif function f, a corresponding OCaml function $\mathbb{G}(f)$. We assume that these functions are all provided in an OCaml module μ_{prim}.

Requirements. CryptoVerif verifies the following requirements:

1. Variables are renamed so that each variable has a single definition. The indices $\widetilde{i}$ of a variable $x[\widetilde{i}]$ are always the indices of replications above the definition of x.
2. The processes are well-typed. (In particular, functions receive arguments of their expected types. See [4] for a similar type system.)
3. Oracles with the same name can be defined only in different branches of an if or get construct.
   We define types of oracles as follows. The type of a $\mathsf{return}(M_1, \ldots, M_n); Q$ statement consists of the types of $M_1, \ldots, M_n$ and the type of the oracle definitions at the beginning of Q. The type of an oracle definition consists of the role that it starts (if it starts a role), the oracle name, the bounds of the replications above that oracle definition, the types of the arguments of the oracle, and the common type of its return statements.
   An oracle may have several return statements, but they must be of the same type. When there are several definitions of an oracle with the same name O, they must be of the same type.

Item 1 makes sure that a distinct array cell is used in each copy of a process, so that all values of the variables during execution are kept in memory. (This helps in cryptographic proofs.) To lighten notations, we often omit the indices since they are determined by Item 1. Item 3 guarantees that the various definitions of an oracle are consistent, and can in fact be compiled into a single function in OCaml. Furthermore, for simplicity, we also require the following points:

1. All oracle definitions are included in a role.
2. No replication occurs directly under a replication.

We can encode nested replications by adding a dummy oracle between the two replications. These assumptions are relaxed in our implementation.

3 The OCaml Language

We do not repeat the syntax of OCaml, which is standard (see the manual at http://caml.inria.fr). To define its formal semantics, we adapted the semantics by Scott Owens et al. [13]. This semantics is a small step, operational semantics of the core part of the OCaml language. We modified it in several ways.

First, this semantics substitutes directly variables with their values. Instead, we define an environment Env that maps variables to their values. This way, it is easier to relate the OCaml state to the CryptoVerif state which also contains an environment. Then, we need to define explicit closures $\mathsf{function}[Env, pm]$ where pm is a pattern matching. A pattern matching is a list of tuples containing a pattern and an expression, which is denoted $pat_1 \to e_1 \mid \ldots \mid pat_m \to e_m$. When matching a value v, this executes the first expression e_j such that the pattern pat_j matches v. We also need to add an explicit call stack $Stack$. The stack is a list of pairs (Env, C), where C is an *evaluation context*, that is, an expression with a hole $[\cdot]$, such that the expression inside the hole can be immediately evaluated. For instance, $e\ [\cdot]$ and $[\cdot]\ v$ are evaluation contexts, so we evaluate the argument of applications first, and when it becomes a value v, we evaluate the function. In order to evaluate an expression $C[e]$, that is, C with e in the hole, we push the context C on the stack with the current environment, evaluate the expression e until it becomes a value v, and finally pop the context C from the stack, inserting the obtained value in it, yielding $C[v]$. As usual, the contents of references are stored in a *store*, which maps locations to their current values. Hence, the semantic configuration of an OCaml program is $\langle Env, pe, Stack, store\rangle$, where pe is the program or expression currently evaluated.

Moreover, a security protocol typically involves several programs running in parallel on different machines. We model this by considering several threads. Each thread has a configuration $Th_i = \langle Env_i, pe_i, Stack_i, store_i\rangle$ and the semantic configuration becomes

$$[Th_1, \ldots, Th_n], globalstore, j\ ,$$

where j is the number of the thread currently being executed, and $globalstore$ is a store for locations shared between threads. We use it to model the communication between threads by storing messages in shared locations, and to store the files containing private data from the CryptoVerif process (free variables of roles and tables). In practice, these files may for example be copied from one machine to another by the user. The values in the global store contain no closure and no reference. Let S_g be the set of locations of the global store and $S_{priv} \subset S_g$ be the ones reserved for private CryptoVerif data. (The latter cannot be used by the adversary, following Assumption A3.) We define the new primitives $\mathsf{addthread}(pe)$, which starts the program pe in a new thread, and $\mathsf{schedule}(j)$, which stops execution of the current thread and continues execution of the thread j. The primitive addthread does not allow using the same local store in several threads, which corresponds to forbidding fork in the middle of a role. Moreover, we reduce only the active thread, and we change threads only with schedule. So we can only change threads in code defined by the adversary, because neither the primitives nor the generated modules use schedule. So a call to an oracle cannot be interleaved with other threads. This corresponds to Assumption A6: if multiple oracles cannot interleave reads and writes in the same table file, one can reconstruct a well-defined call order for these oracles in the CryptoVerif process, which processes one oracle call after another, so that the calls can be simulated in our semantics.

In order to represent random choices, we add the primitive random (), which returns a random boolean true or false with equal probability.

OCaml programs typically contain several modules. A module named μ consists of an OCaml program pe_μ and its interface $\mathit{interface}_\mu$ that is the set of OCaml identifiers defined in μ and usable in other modules. A correct OCaml program is then of the form $pe_{\mu_1};; \dots ;; pe_{\mu_n}$, where, for all $i \leq n$, the free variables of μ_i are defined in the interfaces of μ_j with $j < i$.

Finally, we instrument OCaml code in three ways. First, we add a new kind of functions and closures tagfunction that behave exactly in the same way as the regular closures. We use these to differentiate closures coming from our generated code and closures coming from the adversary.

Second, we need to be able to match CryptoVerif events, so we add to the semantic configuration an element *events* that contains the list of the events executed until now. We add the primitive event $e(e_1, \dots, e_k)$ that adds the event $e(v_1, \dots, v_k)$ to *events* where $e_1, \dots, e_k$ evaluate to the values $v_1, \dots, v_k$ respectively. Events serve in specifying security properties of protocols, so they appear in generated code, but cannot be used by the adversary.

Third, the roles of a CryptoVerif process cannot be executed in any order: if a role is defined after the return from an oracle, it can be executed only after the previous oracle has returned. For instance, we can run a server only after generating its keys. We need to enforce this constraint also in the OCaml program. Each CryptoVerif role *role* is translated by our compiler into an OCaml module μ_{role}. We add to the OCaml configuration the multiset of callable modules $\mathcal{MI}$ that contains tuples (μ_{role}, b) of a module μ_{role} and a boolean b, indicating, if true, that the module can be called any number of times and if false that the module can be called only once. The construct addthread is then modified to reject new programs that contain a module that cannot be called. We add the primitive $\mathsf{return}(\mathcal{MI}', e)$ that adds to the module list $\mathcal{MI}$ the generated modules present in $\mathcal{MI}'$, and returns the result of e. This primitive is useful to add modules newly defined at the return from an oracle.

$$\mathbb{G}(x \overset{R}{\leftarrow} T; P) = \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{random}}(T_x)\ ()\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x); \mathbb{G}(P)) \tag{Random}$$

$$\mathbb{G}(x \leftarrow M; P) = \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x); \mathbb{G}(P)) \tag{Let}$$

$$\mathbb{G}(\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P') = \mathsf{if}\ \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{then}\ \mathbb{G}(P)\ \mathsf{else}\ \mathbb{G}(P') \tag{If}$$

$$\frac{[(Q_1, b_1), \dots, (Q_l, b_l)] = \mathrm{reduce}'(Q)}{\mathbb{G}(\mathsf{return}(N_1, \dots, N_k); Q) = (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \dots, \mathbb{G}_{\mathsf{O}}(Q_l, b_l), \mathbb{G}_{\mathsf{M}}(N_1), \dots, \mathbb{G}_{\mathsf{M}}(N_k))} \tag{Return1}$$

$$\mathbb{G}(\mathsf{return}(N_1, \dots, N_k)\}; Q) = (\mathsf{return}(\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q), (\mathbb{G}_{\mathsf{M}}(N_1), \dots, \mathbb{G}_{\mathsf{M}}(N_k)))) \tag{Return2}$$

$$\mathbb{G}(\mathsf{end}) = (\mathsf{raise\ Match\_failure}) \tag{End}$$

$$\mathbb{G}(\mathsf{event}\ e(M_1, \dots, M_k)) = \mathsf{event}\ e(\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_k)) \tag{Event}$$

$$\begin{aligned}
&\mathbb{G}_{\mathsf{O}}(Q, \mathrm{false}) = \mathsf{let}\ \mathit{token} = \mathsf{ref}\ \mathrm{true}\ \mathsf{in}\ \mathsf{tagfunction}\ \mathit{pm}_{\mathrm{false}}(Q) \\
&\mathit{pm}_{\mathrm{false}}(O(x_1 : T_1, \dots, x_k : T_k) := P) = (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k)) \to \\
&\qquad \mathsf{if}\ (!\mathit{token})\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \dots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k)) \\
&\qquad \mathsf{then}\ (\mathit{token} := \mathrm{false}; \mathbb{G}(P))\ \mathsf{else\ raise\ Bad\_Call}
\end{aligned} \tag{Oracle1}$$

$$\begin{aligned}
&\mathbb{G}_{\mathsf{O}}(Q, \mathrm{true}) = \mathsf{tagfunction}\ \mathit{pm}_{\mathrm{true}}(Q) \\
&\mathit{pm}_{\mathrm{true}}(O(x_1 : T_1, \dots, x_k : T_k) := P) = (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k)) \to \\
&\qquad \mathsf{if}\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \dots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{then}\ \mathbb{G}(P) \\
&\qquad \mathsf{else\ raise\ Bad\_Call}
\end{aligned} \tag{Oracle2}$$

Fig. 2. Translation function $\mathbb{G}$, excerpt

Hence, the instrumented semantic configuration is

$$\mathcal{C} = [\mathit{Th}_1, \dots, \mathit{Th}_n], \mathit{globalstore}, j, \mathcal{MI}, \mathit{events}$$

We have shown that this instrumentation does not alter the semantics of OCaml: an instrumented program behaves exactly in the same way as that program with the instrumentation deleted, provided only allowed roles are executed, as assumed by Assumption A2. Below, when the current thread of $\mathcal{C}$ is $\mathit{Th}_j = \langle \mathit{Env}_j, \mathit{pe}_j, \mathit{Stack}_j, \mathit{store}_j \rangle$, we denote by $\mathcal{C}_{\mathit{pe}}(\mathcal{C}) = \mathit{pe}_j$ and $\mathcal{C}_{\mathit{store}}(\mathcal{C}) = \mathit{store}_j$ the current program and store of $\mathcal{C}$.

4 Translation

Our compiler translates each CryptoVerif role *role* into an OCaml module μ_{role} and each CryptoVerif oracle into a function. Let $\mathbb{G}_{\mathsf{var}}$ be an injective function taking a CryptoVerif variable name and returning an OCaml variable name. The function $\mathbb{G}_{\mathsf{M}}$ transforms a term M into an OCaml expression as follows: $\mathbb{G}_{\mathsf{M}}(x[\widetilde{i}]) = \mathbb{G}_{\mathsf{var}}(x)$ and $\mathbb{G}_{\mathsf{M}}(f(M_1, \dots, M_m)) = \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_m))$. To translate an oracle, we translate the body of the oracle using the function $\mathbb{G}$ defined in Fig. 2, except the translation of get and insert, which is shown in [6]. Most cases are straightforward.

After defining a variable, we store it in a file if needed, using $\mathbb{G}_{\mathsf{file}}(x)$ which is $()$ when $x[]$ is not present in files, and $f := \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x)$ when $(x[], f)$ is present in files.

For the return case, if the return is not at the end of a role, we return the closures corresponding to the oracles defined after the return (Return 1). Otherwise, we update the set of available roles using the primitive return introduced in the previous section (Return 2). We let $\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q)$ be the set of pairs (μ_{role}, b) where *role* is defined at the beginning of Q, and the boolean b is true if the role *role* is under a replication and false otherwise.

An oracle $O(x_1, \dots, x_n) := P$ is transformed in a closure by the function $\mathbb{G}_{\mathsf{O}}$ as shown in Fig. 2. When the oracle O is not under replication (second argument of $\mathbb{G}_{\mathsf{O}}$ false, in (Oracle 1)), we use a boolean token *token* to make sure that it can be called only once: *token* is initially true, it is set to false in the first call. In subsequent calls, an exception will be raised. The translation of an oracle always checks that the arguments are correct values for their CryptoVerif types.

Finally, we generate an OCaml module μ_{role} for each role *role* in the CryptoVerif process. This module provides a single function init which returns the functions implementing the oracles defined at the beginning of $Q(role)$, so its interface is $interface_{\mu_{role}} = \{\mu_{role}.\mathsf{init}\}$ and its program is

$$\begin{array}{l}
pe_{\mu_{role}} = \mathsf{let}\ \mu_{role}.\mathsf{init} = \mathsf{let}\ token = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}\ pm_{role}\\
pm_{role} = () \to \mathsf{if}\ (!token)\ \mathsf{then}\\
\qquad\qquad\qquad\quad \mathbb{G}_{\mathsf{read}}(x_1[\,])\ \mathsf{in}\ \dots \mathbb{G}_{\mathsf{read}}(x_m[\,])\ \mathsf{in}\\
\qquad\qquad\qquad\quad (token := \text{false};\ (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \dots, \mathbb{G}_{\mathsf{O}}(Q_k, b_k)))\\
\qquad\qquad\qquad \mathsf{else\ raise\ Bad\_Call}
\end{array}$$

where $[(Q_1, b_1), \dots, (Q_n, b_n)] = \text{reduce}'(Q(role))$ and $x_1[\,], \dots, x_k[\,]$ are the free variables of $Q(role)$, which are the variables we need to retrieve from the files. These variables are read by $\mathbb{G}_{\mathsf{read}}(x) = \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{deser}}(T_x)\ !(f)$ where $(x[\,], f) \in \mathsf{files}$.

The generated modules are included in manually-written programs that represent the full implementation of the protocol, for instance a client and a server. In particular, these programs are responsible for sending the result of oracles to the network and receiving messages to be passed as arguments to oracles. We consider that these programs are run by the adversary using the $\mathsf{addthread}$ primitive. We represent the adversary by an OCaml program pe_0. For simplicity, we require that the programs added by $\mathsf{addthread}$ in pe_0 contain modules in the following order $pe_{\mu_{\mathsf{prim}}};; pe_{\mu_{role_1}};; \dots;; pe_{\mu_{role_k}};; pe$ where pe contains no generated module. We assume that pe_0 uses the generated modules only inside $\mathsf{addthread}$, and that pe_0 is a well-typed OCaml program. (The network code is well-typed by Assumption A4. The adversary itself is any probabilistic Turing machine, which can be implemented by a well-typed OCaml program.) The program pe_0 is run in the initial OCaml configuration $\mathcal{C}_0(Q_0, pe_0)$ defined as follows:

$$\mathcal{C}_0(Q_0, pe_0) = [\langle \emptyset, pe_0, [\,], \emptyset \rangle], globalstore_0, 1, \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_0)$$

where $\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_0)$ is the set of modules available at the beginning of the execution and $globalstore_0 = \{x \mapsto [\,] \mid x \in S_g\}$ is the initial value of global store. We use the empty list $[\,]$ as initial value, representing that the file has not been created yet, or that the table is empty.

5 Correctness of the Translation of Oracle Bodies

Let us first define precisely the notion of trace.

Definition 1 (Traces). *A* trace $\mathcal{CT}$ *is a sequence of reductions:* $\mathcal{CT} = \mathcal{C}_0 \rightarrow_{p_1} \cdots \rightarrow_{p_n} \mathcal{C}_n$*. The trace* $\mathcal{CT}$ *is* complete *when there is no possible reduction from its last configuration* $\mathcal{C}_n$*. The probability of the trace* $\mathcal{CT}$ *is* $\Pr[\mathcal{CT}] = p_1 \times \cdots \times p_n$*. The probability of a set of traces is the sum of the probabilities of its elements.*

We write $\mathcal{C} \rightarrow_p^* \mathcal{C}'$ *when there exists a trace from* $\mathcal{C}$ *to* $\mathcal{C}'$ *and* p *is the probability of the set of all traces from* $\mathcal{C}$ *to* $\mathcal{C}'$ *that contain a single occurrence of* $\mathcal{C}'$*. The notation* $\mathcal{C} \rightarrow^* \mathcal{C}'$ *means* $\mathcal{C} \rightarrow_1^* \mathcal{C}'$*.*

We denote OCaml traces by $\mathcal{CT}$ *and CryptoVerif traces by* $\mathfrak{CT}$*.*

For expressions that do not use addthread, return, event, nor schedule operations and do not use the store nor the global store, we use the shortened OCaml configuration $Env, pe, Stack$. The rest of the OCaml configuration is unchanged. We make the following assumptions on the code of primitives.

Assumption 1. 1. *There are no* addthread, return, event, *nor* schedule *operations and no mention of global store locations in* $pe_{\mu_{\mathsf{prim}}}$*.*
2. *There are no operations that touch the store (*ref, !, :=*) and no mention of store locations in* $pe_{\mu_{\mathsf{prim}}}$*.*
3. *There exists* Env_{prim} *such that for all programs* pe*, we have* $\emptyset, pe_{\mu_{\mathsf{prim}}};; pe, [\,] \rightarrow^* Env_{\mathsf{prim}}, pe, [\,]$*.*

Item 2 may not be realistic (the primitives often use the store internally). This assumption is made here only for simplicity; it can be relaxed at the cost of a much more complex proof: when the primitives use the store, we need to make sure that the part of the store used by the primitives is disjoint from the one used by the rest of the program. We are finishing this proof. Item 3 requires that there are no random choices during the initialization of primitives, so that the obtained environment is always the same. This is not restrictive since random choices are allowed during calls to primitives.

To establish the correspondence between CryptoVerif values and OCaml values, we define a function $\mathbb{G}_{\mathsf{val}_T}$, which maps each CryptoVerif bitstring a to its associated value v in OCaml. For a given type T, $\mathbb{G}_{\mathsf{val}_T}$ must be a bijection between T and the set of OCaml values of type $\mathbb{G}_T(T)$ satisfying the predicate function $\mathbb{G}_{\mathsf{pred}}(T)$. We extend this function to events by $\mathbb{G}_{\mathsf{ev}}(e(a_1, \dots, a_j)) = e(\mathbb{G}_{\mathsf{val}_{T_1}}(a_1), \dots, \mathbb{G}_{\mathsf{val}_{T_j}}(a_j))$ if e is of type $T_1 \times \cdots \times T_j$. This function is naturally extended to lists of events. The next assumption states that the primitives have been correctly implemented, following Assumption A1.

Assumption 2 (Correct primitives). *For each CryptoVerif function* f *of type* $T_1 \times \cdots \times T_n \rightarrow T$*, for all CryptoVerif values* $a_1, \dots, a_n$ *of types* $T_1, \dots, T_n$*,* $Env_{\mathsf{prim}}, \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{val}_{T_1}}(a_1), \dots, \mathbb{G}_{\mathsf{val}_{T_n}}(a_n)), [\,] \rightarrow^* Env', \mathbb{G}_{\mathsf{val}_T}(f(a_1, \dots, a_n)), [\,]$*.*

For each CryptoVerif type T *used for getting fresh values, for each value* $a \in T$*,* $Env_{\mathsf{prim}}, \mathbb{G}_{\mathsf{random}}(T)\ (), [\,] \rightarrow_{1/|T|}^* Env', \mathbb{G}_{\mathsf{val}_T}(a), [\,]$*.*

For each CryptoVerif type T used to check predicates, for each value v of the OCaml type $\mathbb{G}_T(T)$, $Env_{\mathsf{prim}}, \mathbb{G}_{\mathsf{pred}}(T)\ v, [\,] \rightarrow^* Env', v', [\,]$ where $v' = \mathrm{true}$ if $\mathbb{G}^{-1}_{\mathsf{val}_T}(v)$ exists, and $v' = \mathrm{false}$ otherwise.

For each CryptoVerif type T used for serialization/deserialization, for each value $a \in T$, there exists an OCaml string value $\mathsf{ser}(T, a)$, such that

$$Env_{\mathsf{prim}}, \mathbb{G}_{\mathsf{ser}}(T)\ \mathbb{G}_{\mathsf{val}_T}(a), [\,] \rightarrow^* Env', \mathsf{ser}(T, a), [\,]$$
$$\textit{and } Env_{\mathsf{prim}}, \mathbb{G}_{\mathsf{deser}}(T)\ \mathsf{ser}(T, a), [\,] \rightarrow^* Env', \mathbb{G}_{\mathsf{val}_T}(a), [\,]$$

We denote by $\mathsf{fv}(P)$ the free variables of the oracle body P, with their indices, defined as usual. Let us define the minimal environments and global stores corresponding to CryptoVerif variables and tables.

Definition 2 (Minimal environments and stores).

$$Env(E, P) = \{\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}_{T_x}}(a) \mid x[\widetilde{a}] \in \mathsf{fv}(P), E, x[\widetilde{a}] \Downarrow a\} \quad \text{(Environment)}$$

$$\begin{aligned} &\mathit{globalstore}(E, \mathcal{T}) = \big\{ \\ &\quad \{f \mapsto \mathsf{ser}(T_x, \mathbb{G}_{\mathsf{val}_{T_x}}(a)) \mid (x[\,], f) \in \mathsf{files}, E, x[\,] \Downarrow a\} \cup \\ &\quad \{f \mapsto [\,] \mid (x[\,], f) \in \mathsf{files}, x \textit{ not defined in } E\} \cup \qquad \text{(Global store)} \\ &\quad \{f_1 \mapsto l_1, \ldots, f_k \mapsto l_k\} \\ &\mid \{(Tbl_1, f_1), \ldots, (Tbl_k, f_k)\} = \mathsf{tables}, l_i \in \mathbb{G}_{\mathsf{tbl}}(\mathcal{T}(Tbl_i))\big\} \end{aligned}$$

where $\mathbb{G}_{\mathsf{tbl}}(t)$ is the set of OCaml lists corresponding to the table contents t, defined as follows. Suppose the table has type $T = T_1, \ldots, T_l$. Let $\mathbb{G}_{\mathsf{tblel}}(b_1, \ldots, b_l)$ be the OCaml value $(\mathsf{ser}(T_1, \mathbb{G}_{\mathsf{val}_{T_1}}(b_1)), \ldots, \mathsf{ser}(T_l, \mathbb{G}_{\mathsf{val}_{T_l}}(b_l)))$ corresponding to the table element $(b_1, \ldots, b_l)$. If $t = \{a_1, \ldots, a_k\}$, $\mathbb{G}_{\mathsf{tbl}}(t)$ is the set of all lists containing elements $\mathbb{G}_{\mathsf{tblel}}(a_1), \ldots, \mathbb{G}_{\mathsf{tblel}}(a_k)$ in any order.

We also define $Env(E, Q)$ and $Env(E, M)$ in the same way as $Env(E, P)$.

The *globalstore* function above returns the set of all the possible minimal global stores in which the contents of the files and the tables is correct with respect to the CryptoVerif configuration.

The next lemma shows that CryptoVerif terms are correctly translated into OCaml. It is proved by induction on the syntax of the term M.

Lemma 1 (Term reduction). *For all CryptoVerif terms M of type T and all CryptoVerif environments E, if $Env \supseteq Env_{\mathsf{prim}} \cup Env(E, M)$ and $E, M \Downarrow a$, then $Env, \mathbb{G}_{\mathsf{M}}(M), Stack \rightarrow^* Env', \mathbb{G}_{\mathsf{val}_T}(a), Stack$.*

The next lemma shows that CryptoVerif oracle bodies are correctly translated into OCaml.

Lemma 2 (Inner reduction). *Consider a CryptoVerif configuration $\mathfrak{C}$ whose program part P is not in a return, end, call or loop form. Suppose we have n reductions beginning at this configuration:*

$$\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_{p_i} \mathfrak{C}_i = E_i, P_i, \mathcal{Q}, \mathcal{T}_i, \mathcal{R}, \mathcal{E}_i$$

for $i \leq n$. Let $\mathcal{C}$ be an OCaml configuration such that $\mathcal{C} = [\mathit{Th}_1, \dots, \mathit{Th}_j, \dots, \mathit{Th}_m], \mathit{globalstore}, j, \mathcal{MI}, \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$ with $\mathit{Th}_j = \langle \mathit{Env}, \mathbb{G}(P), \mathit{Stack}, \mathit{store} \rangle$, $\mathit{Env}_{\mathsf{prim}} \cup \mathit{Env}(E, P) \subseteq \mathit{Env}$, and $G \subseteq \mathit{globalstore}$ for some $G \in \mathit{globalstore}(E, \mathcal{T})$.

Then there exist m disjoint sets of OCaml traces $\mathcal{CTS}_1, \dots, \mathcal{CTS}_m$, all starting at $\mathcal{C}$, such that none of these traces is a prefix of another of these traces, $\Pr[\mathcal{CTS}_i] = p_i$, and the last configurations of traces in $\mathcal{CTS}_i$ are of the form $[\mathit{Th}_1, \dots, \mathit{Th}'_j, \dots, \mathit{Th}_m], \mathit{globalstore}', j, \mathcal{MI}, \mathbb{G}_{\mathsf{ev}}(\mathcal{E}_i)$ with $\mathit{Th}'_j = \langle \mathit{Env}', \mathbb{G}(P_i), \mathit{Stack}, \mathit{store} \rangle$, $\mathit{Env}_{\mathsf{prim}} \cup \mathit{Env}(E_i, P_i) \subseteq \mathit{Env}'$, and $G' \subseteq \mathit{globalstore}'$ for some Env', $\mathit{globalstore}'$, and some $G' \in \mathit{globalstore}(E_i, \mathcal{T}_i)$.

This lemma is proved by cases on the process P. We use Lemma 1 when we need to evaluate a term. Similar lemmas can be proved for return and end, adapting the form of the final configuration. The oracle bodies that we translate into OCaml do not contain calls nor loops.

6 Simulation of OCaml Code

In this section, we show how to simulate in CryptoVerif any OCaml program pe_0 corresponding to an adversary interacting with the protocol implementation generated from the CryptoVerif process Q_0. Basically, we run the OCaml program pe_0 inside a CryptoVerif primitive f (which is possible since these primitives can represent any deterministic Turing machine). When pe_0 needs to call an oracle of Q_0, the primitive returns and the call is made by CryptoVerif. When pe_0 needs to generate a random number, this generation is performed by CryptoVerif.

We assume that the OCaml program pe_0 runs in bounded time, so makes a bounded number of oracle calls. When oracle O is under replication, we let N_O be the maximum number of calls to $\mathbb{G}_{\mathsf{O}}(Q, \mathrm{true})$ for each call to the oracle defined above O. We use N_O as the bound of the replication above O. When a role *role* is under replication, we let N_{role} be the maximum number of executions of $\mathsf{addthread}(pe)$ for some pe that contains μ_{role} for each call to the oracle defined above *role*. We use N_{role} as the bound of the replication above *role*. These replication bounds are chosen such that the OCaml program never exhausts the number of oracle calls allowed by the CryptoVerif process.

From the OCaml program pe_0, we build a CryptoVerif adversary $Q_{\mathrm{adv}}(Q_0, pe_0)$ given in Fig. 3. The initial CryptoVerif configuration is then $\mathfrak{C}_0(Q_0, pe_0) = \mathfrak{C}_{\mathrm{i}}(Q_0 \mid Q_{\mathrm{adv}}(Q_0, pe_0))$. In Fig. 3, we use a let construct with pattern matching, which can easily be encoded in the syntax shown in Fig. 1, by defining functions for creating tuples and their corresponding projections. Let $O_1, \dots, O_n$ be the oracle names in Q_0. We define n constants $o_1, \dots, o_n$ which are used to designate the oracles $O_1, \dots, O_n$ respectively, o_R which corresponds to a random choice, and o_S which corresponds to the end of the OCaml program.

The adversary is mainly encoded by the function f. This function takes as argument the bitstring representation $s = \mathsf{repr}(\mathcal{CS})$ of a simulator configuration $\mathcal{CS} = \mathcal{C}, \mathcal{RI}, \mathcal{I}$, which consists of an OCaml configuration $\mathcal{C}$ (without $\mathcal{MI}$ and *events*) and sets $\mathcal{RI}$ and $\mathcal{I}$ that finitely represent the callable oracles $\mathcal{Q}$ of CryptoVerif. The set $\mathcal{RI}$ represents the callable roles with their replication indices. Its

```
$Q_{\mathrm{adv}}(Q_0, pe_0) = (O_{\mathrm{start}}() =$ let $r =$ loop $O_{loop}(s_0(Q_0, pe_0))$ in end else end)
                  $\mid Q_c(Q_0, pe_0)$

$Q_c(Q_0, pe_0) =$ foreach $i' \leq N'$ do $O_{loop}[i'](s : bitstring) :=$
        let $(s', o, i, args) = f(s)$ in
        if $o = o_S$ then
            return$(s', \mathrm{false})$
        else if $o = o_1$ then
            let $(a_{1_1}, \dots, a_{1_{m_1}}) = args$ in let $(i_{1_1}, \dots, i_{1_{n_1}}) = i$ in
            $(r_{1_1}, \dots, r_{1_{k_1}}) \leftarrow O_1[i_{1_1}, \dots, i_{1_{n_1}}](a_{1_1}, \dots, a_{1_{m_1}})$;
                return$(f'_{O_1}(s', (r_{1_1}, \dots, r_{1_{k_1}})), \mathrm{true})$
            else return$(f''_{O_1}(s'), \mathrm{true})$
        else if $o = o_2$ then ...
        else if $o = o_R$ then
            $b_R \overset{R}{\leftarrow} bool$; return$(f_R(s', b_R), \mathrm{true})$
```

Fig. 3. The program $Q_c(Q_0, pe_0)$

elements are as follows: $role\big[[a, +\infty[, \widetilde{a'}\big]$ means that role $role$ is under replication and the roles $role[1, \widetilde{a'}]$ to $role[a-1, \widetilde{a'}]$ have been used, and the roles $role[a, \widetilde{a'}]$ to $role[N_\mu, \widetilde{a'}]$ are callable; $role[\widetilde{a}]$ means that $role$ is not under replication and $role[\widetilde{a}]$ is callable. Similarly, the set $\mathcal{I}$ represents the callable oracles (inside an already started role), using $O\big[[a, +\infty[, \widetilde{a'}\big]$ to mean that the oracles $O[a, \widetilde{a'}]$ to $O[N_O, \widetilde{a'}]$ are usable, and $O[\widetilde{a}]$ to mean that oracle $O[\widetilde{a}]$ can be called.

The OCaml configuration $\mathcal{C}$ is slightly modified with respect to a standard OCaml configuration. It uses the additional constructs $\mathsf{call}(O[\widetilde{a}])$ and $\mathsf{call}(O[\_, \widetilde{a}])$ which are functional values and correspond to our generated closures for oracle O (the latter for oracles under replication, for which the first index is the first available index value; the former for oracles not under replication). The semantics of $\mathsf{call}(O[\widetilde{a}])\ (v_1, \dots, v_k)$ is defined as follows. If the required oracle $O[\widetilde{a}]$ is not in $\mathcal{I}$, or the number of arguments of O is not k, or $v_1, \dots, v_k$ are not values of the types expected by O, then it reduces into raise Bad_Call, as the OCaml translation of the oracle does. Otherwise it blocks; the oracle call succeeds but will be handled outside f by the process $Q_c(Q_0, pe_0)$ as we detail below. The semantics of $\mathsf{call}(O[\_, \widetilde{a}])$ is defined similarly. The semantics of random is modified so that random () blocks; the random number generation is handled outside f by the process $Q_c(Q_0, pe_0)$. The semantics of addthread is also modified so that it updates the sets of available roles and oracles $\mathcal{RI}$ and $\mathcal{I}$, and it replaces the program of generated modules μ_{role} from CryptoVerif code with:

$$\begin{aligned} program'(role[\widetilde{a}]) &= \mathsf{let}\ \mu_{role}.\mathsf{init} = \mathsf{let}\ token = \mathsf{ref}\ \mathsf{true}\ \mathsf{in}\ \mathsf{tagfunction}\ pm'_{role[\widetilde{a}]} \\ pm'_{role[\widetilde{a}]} &= () \rightarrow \mathsf{if}\ (!token)\ \mathsf{then}\ (token := \mathrm{false}; (\mathsf{call}(O_1[\widetilde{a}]), \dots, \mathsf{call}(O_k[\widetilde{a}]))) \\ &\qquad\qquad\qquad \mathsf{else\ raise\ Bad\_Call} \end{aligned}$$

where $\widetilde{a}$ are the smallest replication indices such that $role[\widetilde{a}]$ is present in $\mathcal{RI}$, $O_1, \dots, O_k$ are the oracles defined at the beginning of $Q(role)$, which are supposed not to be under replication. (When O_j is under replication, we use the form $\mathsf{call}(O_j[\_,\widetilde{a}])$ instead of $\mathsf{call}(O_j[\widetilde{a}])$.) All oracle calls are then performed using call rather than directly in OCaml.

Then $f(\mathsf{repr}(\mathcal{CS}))$ is defined as follows. Let $\mathcal{CS}'$ be the configuration such that $\mathcal{CS} \to^* \mathcal{CS}'$ and $\forall \mathcal{CS}'', \mathcal{CS}' \not\to \mathcal{CS}''$, using the semantics outlined above.

- If $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{call}(O[\widetilde{a}])\ (v_1, \dots, v_l)$, let $T_1, \dots, T_l$ be the type of the arguments of the oracle O and let o be the constant associated to O. We define
$$f(\mathsf{repr}(\mathcal{CS})) = (\mathsf{repr}(\mathcal{CS}'), o, \widetilde{a}, (\mathbb{G}^{-1}_{\mathsf{val}_{T_1}}(v_1), \dots, \mathbb{G}^{-1}_{\mathsf{val}_{T_l}}(v_l)))\,.$$
The case $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{call}(O[\_,\widetilde{a}])\ (v_1, \dots, v_l)$ can be defined similarly, calling $O[a', \widetilde{a}]$ when $O\big[[a', +\infty[, \widetilde{a}\big]$ is in the component $\mathcal{I}$ of $\mathcal{CS}'$.
- If $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{random}\ ()$, we define $f(\mathsf{repr}(\mathcal{CS})) = (\mathsf{repr}(\mathcal{CS}'), o_R, (), ())$.
- Otherwise, we define $f(\mathsf{repr}(\mathcal{CS})) = (\mathsf{repr}(\mathcal{CS}'), o_S, (), ())$.

The function f can be implemented by a deterministic Turing machine (since the random choices are handled outside f), so it can be used as a CryptoVerif primitive.

When f returns $(\mathsf{repr}(\mathcal{CS}'), o, \widetilde{a}, (a_1, \dots, a_l))$, the CryptoVerif process $Q_c(Q_0, pe_0)$ performs the corresponding oracle call $O[\widetilde{a}](a_1, \dots, a_l)$ (lines (4)–(6) of Fig. 3). Similarly, when f returns $(\mathsf{repr}(\mathcal{CS}'), o_R, (), ())$, the process $Q_c(Q_0, pe_0)$ performs a random choice (lines (7)–(8)), and when f returns $(\mathsf{repr}(\mathcal{CS}'), o_S, (), ())$, the process $Q_c(Q_0, pe_0)$ terminates (lines (2)–(3); the corresponding OCaml program also terminates).

The functions f'_O and f''_O replace, in the simulator configuration, the call expression by the result returned by the oracle, or raise the $\mathsf{Match\_failure}$ exception, respectively. More formally,

- $f'_O(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I}), (r_1, \dots, r_n)) = \mathsf{repr}(\mathcal{C}', \mathcal{RI}, \mathcal{I}')$ where the current program is $\mathcal{C}_{pe}(\mathcal{C}) = \mathsf{call}(O[\widetilde{a}])\ (v_1, \dots, v_l)$, $T_1, \dots, T_n$ are the types of the return value of O, the oracles defined after the return from O in Q_0 are $O_1, \dots, O_k$, $\mathcal{C}'$ is the configuration $\mathcal{C}$ where the current expression is replaced with the translated result $(\mathsf{call}(O_1[\widetilde{a}]), \dots, \mathsf{call}(O_k[\widetilde{a}]), \mathbb{G}_{\mathsf{val}_{T_1}}(r_1), \dots, \mathbb{G}_{\mathsf{val}_{T_n}}(r_n))$, $\mathcal{I}' = \mathcal{I} \setminus \{O[\widetilde{a}]\} \cup \{O_1[\widetilde{a}], \dots, O_k[\widetilde{a}]\}$ is obtained from $\mathcal{I}$ by removing the called oracle $O[\widetilde{a}]$ and adding the newly available oracles $O_1[\widetilde{a}], \dots, O_k[\widetilde{a}]$. (For simplicity, the previous notations assume that the oracles $O, O_1, \dots, O_k$ are not under replication; they can be easily adapted to situations in which these oracles are under replication. We also assumed that the return of oracle O does not terminate a role. When it terminates a role, the oracles $O_1, \dots, O_k$ are not returned nor added to $\mathcal{I}$, but the roles that start after the return from O are added to $\mathcal{RI}$.)
- $f''_O(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I})) = \mathsf{repr}(\mathcal{C}', \mathcal{RI}, \mathcal{I} \setminus \{(O[\widetilde{a}]\})$ where $\mathcal{C}_{pe}(\mathcal{C}) = \mathsf{call}(O[\widetilde{a}])\ (v_1, \dots, v_l)$ and $\mathcal{C}'$ is the configuration $\mathcal{C}$ in which the current expression is replaced with $\mathsf{raise\ Match\_failure}$. (The case $\mathcal{C}_{pe}(\mathcal{C}) = \mathsf{call}(O[\_,\widetilde{a}])\ (v_1, \dots, v_l)$ can be defined similarly.)
- $f_R(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I}), b) = \mathsf{repr}(\mathcal{C}'(b), \mathcal{RI}, \mathcal{I})$ where $\mathcal{C}'(b)$ is the configuration $\mathcal{C}$ in which the current expression is replaced with $\mathbb{G}_{\mathsf{val}_{bool}}(b)$.

Finally, the initial state of the simulator is

$$s_0(Q_0, pe_0) = \mathsf{repr}(([\langle \emptyset, pe_0, [\,], \emptyset \rangle], \mathit{globalstore}_0, 1), \mathcal{RI}_0(Q_0), \emptyset)$$

where $\mathcal{RI}_0(Q_0)$ is the set of the initially callable roles of Q_0.

7 Correctness of the Simulation

This section proves the correctness of the simulator, by showing a precise relation between the state of the simulator and the state of the OCaml program. Basically, the OCaml configuration is obtained by replacing the call present in the simulator configuration with the corresponding closures. We show that the tables and files in the OCaml global store correspond to the CryptoVerif tables $\mathcal{T}$ and environment E, and that the OCaml events match the CryptoVerif events.

Definition 3 (Preliminary definitions). *Let us first define the set of oracles $\mathcal{O}(\mathcal{I})$ and $\mathcal{O}(\mathcal{RI})$ represented by $\mathcal{I}$ and $\mathcal{RI}$ respectively:*

$$\begin{aligned}
\mathcal{O}(\mathcal{I}) &= \{O[b, \widetilde{a'}] \mid O\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{I}, a \le b \le N_O\} \cup \{O[\widetilde{a}] \mid O[\widetilde{a}] \in \mathcal{I}\}\\
\mathcal{O}(\mathcal{RI}) &= \{O[b, \widetilde{a}] \mid \mathit{role}[\widetilde{a}] \in \mathcal{RI}, O \textit{ defined under replication at the beginning}\\
&\qquad \textit{of role in } Q_0, 1 \le b \le N_O\}\\
&\quad \cup \{O[\widetilde{a}] \mid \mathit{role}[\widetilde{a}] \in \mathcal{RI}, O \textit{ defined not under replication at the beginning of}\\
&\qquad \textit{role in } Q_0\}\\
&\quad \cup \{O[b, \widetilde{a'}] \mid \mathit{role}\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{RI}, O \textit{ defined at the beginning of role in}\\
&\qquad Q_0,\ a \le b \le N_O\}
\end{aligned}$$

We say that $\mathcal{I}$ and $\mathcal{RI}$ represent the set of callable processes $\mathcal{Q}$, and we write $\mathcal{Q} \leftrightarrow \mathcal{RI}, \mathcal{I}$, when $\mathcal{Q}$ contains exactly one element $O[\widetilde{a}](\widetilde{x}) := P$ for each $O[\widetilde{a}]$ in $\mathcal{O}(\mathcal{I}) \cup \mathcal{O}(\mathcal{RI})$. In this case, we denote by $\mathcal{Q}(O[\widetilde{a}])$ this element of $\mathcal{Q}$.

Given a simulator thread $Th = \langle Env, pe, Stack, store \rangle$, $\mathcal{O}_{\mathsf{call}}(Th)$ is the set of oracles $O[\widetilde{a}]$ not under replication such that $\mathsf{call}(O[\widetilde{a}])$ occurs in Th outside tagfunction *functions or closures. Let* $\mathsf{rolelist}(Th)$ *be the set of roles $\mathit{role}[\widetilde{a}]$ such that* $\mathsf{tagfunction}[Env, pm'_{\mathit{role}[\widetilde{a}]}]$ *occurs in thread Th, and the store of Th binds $Env(token)$ to* true. *Let* $\mathsf{rolefunlist}(Th)$ *be the set of roles $\mathit{role}[\widetilde{a}]$ such that $\mathit{program}'(\mathit{role}[\widetilde{a}])$ occurs in Th.*

We also define $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ (resp. $\mathsf{rolelist}(\mathcal{CS}), \mathsf{rolefunlist}(\mathcal{CS})$*) as the union of $\mathcal{O}_{\mathsf{call}}(Th)$ (resp.* $\mathsf{rolelist}(Th), \mathsf{rolefunlist}(Th)$*) for all threads present in $\mathcal{CS}$.*

The set $\mathcal{O}_{\mathsf{willbeavailable}}(\mathcal{CS})$ contains the set of oracles that can eventually become available in $\mathcal{O}(\mathcal{I})$ or $\mathcal{O}(\mathcal{RI})$ in a future configuration, and that are not available now. More precisely, $\mathcal{O}_{\mathsf{willbeavailable}}(\mathcal{CS})$ is the set of $O'[\widetilde{a'}, \widetilde{a}]$ such that there is an oracle O defined above O' in Q_0 such that $O[\widetilde{a}] \in \mathcal{O}(\mathcal{I}) \cup \mathcal{O}(\mathcal{RI}) \cup \mathcal{O}(\mathsf{rolelist}(\mathcal{CS})) \cup \mathcal{O}(\mathsf{rolefunlist}(\mathcal{CS}))$.

The function replaceinitpm *replaces everywhere in a simulator thread the pattern matchings corresponding to role initialization of the simulator by the OCaml module initialization for the given role. Formally,* $\mathsf{replaceinitpm}(Th)$ *replaces each occurrence of $pm'_{\mathit{role}[\widetilde{a}]}$ in Th with pm_{role}.*

Suppose that $\mathcal{Q} \leftrightarrow \mathcal{RI},\mathcal{I}$*, and* l_{tok} *is a function that associates to each oracle* $O[\widetilde{a}]$ *the location of its token. We define the set of closures that correspond to an oracle:*

- $\mathsf{correctclosures}(O[\widetilde{a}],\mathcal{I},E,\mathcal{Q},l_{\mathsf{tok}}) = \{\mathsf{tagfunction}[Env, pm_{\text{false}}(\mathcal{Q}(O[\widetilde{a}]))] \mid Env_{\mathsf{prim}} \cup Env(E,\mathcal{Q}(O[\widetilde{a}])) \subseteq Env, Env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\}$ *if* $O[\widetilde{a}] \in \mathcal{I}$*;*
- $\mathsf{correctclosures}(O[\widetilde{a}],\mathcal{I},E,\mathcal{Q},l_{\mathsf{tok}}) = \{\mathsf{tagfunction}[Env, pm_{\text{false}}(Q)] \mid$ *for any* $Q, Env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\}$ *if* $O[\widetilde{a}] \notin \mathcal{I}$*;*
- $\mathsf{correctclosures}(O[\_,\widetilde{a}],\mathcal{I},E,\mathcal{Q},l_{\mathsf{tok}}) = \{\mathsf{tagfunction}[Env, pm_{\text{true}}(\mathcal{Q}(O[a',\widetilde{a}]))] \mid Env_{\mathsf{prim}} \cup Env(E,\mathcal{Q}(O[a',\widetilde{a}])) \subset Env\}$ *if* $O\big[[a',+\infty[,\widetilde{a}\big] \in \mathcal{I}$*;*
- $\mathsf{correctclosures}(O[\_,\widetilde{a''}],\mathcal{I},E,\mathcal{Q},l_{\mathsf{tok}}) = \emptyset$ *if* $O\big[[a',+\infty[,\widetilde{a''}\big] \notin \mathcal{I}$*.*

The function $\mathsf{replacecalls}$ *replaces in its argument the calls* $\mathsf{call}(O[\widetilde{a}])$ *with closures that correspond to the oracle.*

$\mathsf{replacecalls}(\langle Env, pe, Stack, store\rangle,\mathcal{I},E,\mathcal{Q},l_{\mathsf{tok}}) = \{\langle Env', \sigma(pe), \sigma(Stack), store'\rangle \mid Env'$ *is any environment if* pe *is a value* v *or an exception* $\mathsf{raise}\ v$*,* $Env' = \sigma(Env)$ *otherwise;* $store' \supseteq \sigma(store)$ *with* σ *a function that replaces each occurrence of a* $\mathsf{call}(R)$ *with an element of* $\mathsf{correctclosures}(R,\mathcal{I},E,\mathcal{Q},l_{\mathsf{tok}})\}$

The function $\mathsf{gettokens}$ *returns the store part corresponding to the tokens of the closures of oracles not under replication:*

$$\begin{aligned}\mathsf{gettokens}(\mathcal{I},\mathcal{O},l_{\mathsf{tok}}) = {} & \{l_{\mathsf{tok}}(O[\widetilde{a}]) \mapsto \text{true} \mid O[\widetilde{a}] \in \mathcal{O} \cap \mathcal{I}\} \\ & \cup \{l_{\mathsf{tok}}(O[\widetilde{a}]) \mapsto \text{false} \mid O[\widetilde{a}] \in \mathcal{O} \setminus \mathcal{I}\}\end{aligned}$$

Definition 4 (Relation between CryptoVerif and OCaml states). *Let* $\mathcal{CS}$ *be a configuration of the simulator,* $E,\mathcal{T},\mathcal{Q},\mathcal{E}$ *be parts of a CryptoVerif configuration, and* $\mathcal{C}$ *be an OCaml configuration. We say that* $\mathcal{CS},E,\mathcal{T},\mathcal{Q},\mathcal{E} \equiv \mathcal{C}$ *when the following properties are all true:*

1. $\mathcal{CS} = ([Th_1,\ldots,Th_n], globalstore, i),\mathcal{RI},\mathcal{I}$*.*
2. $\mathcal{C} = [Th'_1,\ldots,Th'_n], globalstore', i, \mathcal{MI}, events$*.*
3. $\mathcal{Q} \leftrightarrow \mathcal{RI},\mathcal{I}$*.*
4. *The environment* E *contains values for every free variable in the oracle definitions present in* $\mathcal{Q}$*.*
5. *There exists an injective function* l_{tok} *that associates to each oracle* $O[\widetilde{a}]$ *a store location that does not occur in* $\mathcal{CS}$ *such that*
   - $\forall i \le n$*,* $Th'_i \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th_i),\mathcal{I},E,\mathcal{Q},l_{\mathsf{tok}})$*,*
   - $\forall i \le n, \mathsf{gettokens}(\mathcal{I},\mathcal{O}_{\mathsf{call}}(Th_i),l_{\mathsf{tok}}) \subseteq store'_i$ *where* $store'_i$ *is the store part of* Th'_i*.*
6. *There exists an injective function* l'_{tok} *that associates to each role* $role[\widetilde{a}]$ *a store location such that for all* i*, for all closures* $\mathsf{tagfunction}[Env, pm'_{role[\widetilde{a}]}]$*, present in thread* Th_i*,* $l'_{\mathsf{tok}}(role[\widetilde{a}]) = Env(token)$*.*
7. $\forall l \in S_{priv}, globalstore(l) = [\,]$*.*
8. $G' \subseteq globalstore'$ *for some* $G' \in globalstore(E,\mathcal{T})$*.*
9. $\forall l \notin S_{priv}, globalstore(l) = globalstore'(l)$*.*

10. $\mathcal{MI} = \{(\mu_{role}, \text{false}) \mid role[\widetilde{a}] \in \mathcal{RI}\} \cup \{(\mu_{role}, \text{true}) \mid role\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{RI}\}$.
11. $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$.
12. *The sets* $\mathcal{O}(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$, $\mathcal{O}(\mathcal{RI})$ *and* $\mathcal{O}_{\mathsf{willbeavailable}}(\mathcal{CS})$ *are pairwise disjoint.*
13. *The* $3n$ *sets* $\mathcal{O}_{\mathsf{call}}(Th_i)$, $\mathcal{O}(\mathsf{rolefunlist}(Th_i))$, *and* $\mathcal{O}(\mathsf{rolelist}(Th_i))$ *for* $i \leq n$ *are pairwise disjoint, and are all included in* $\mathcal{O}(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$.

Item 3 is an invariant on the CryptoVerif side: it relates the CryptoVerif available oracles in $\mathcal{Q}$ to elements of the simulator configuration. Item 4 is also an invariant of the CryptoVerif semantics.

Item 5 relates the threads of the simulator and of the OCaml semantics. Basically, the simulator uses call while OCaml uses the corresponding $\mathsf{tagfunction}$. The tokens that determine whether oracles can be called are absent from the simulator: the value of these tokens is determined from $\mathcal{I}$ by the function $\mathsf{gettokens}$, and we require that they are present in the OCaml store with their correct value.

Item 6 assures that all instances of a closure of a given role initialization $role[\widetilde{a}]$ share the same store location for their tokens.

Items 7 to 9 relate the values of the global store in the simulator and in the OCaml semantics. The public part of the global store is the same on both sides (Item 9). The private part (files and tables) is empty in the simulator, since this part is handled by CryptoVerif itself (Item 7). We require that the private part of the OCaml global store corresponds to the CryptoVerif configuration (Item 8).

Item 10 relates the OCaml set of callable modules $\mathcal{MI}$ and the simulator set of callable roles $\mathcal{RI}$, and Item 11 relates the OCaml and CryptoVerif events.

Finally, Items 12 and 13 are required to keep the injections of Items 5 and 6.

Given a CryptoVerif configuration $\mathfrak{C}$, it is easy to recover the corresponding simulator configuration $\mathcal{CS}$. The next definition formalizes this point.

Definition 5 (Simulator configuration). *Let* $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ *be a CryptoVerif configuration. If* $E(r[\,])$ *is defined (we are at the end of the loop), then* $\mathcal{CS}(\mathfrak{C}) = \mathsf{repr}^{-1}(E(r[\,]))$. *Otherwise, if* $E(s[\alpha])$ *is defined and* $E(s'[\alpha])$ *is not defined (we are just before line (1) of Fig. 3 in iteration number* α*), then* $\mathcal{CS}(\mathfrak{C}) = \mathsf{repr}^{-1}(E(s[\alpha]))$. *Otherwise, if* $E(s'[\alpha])$ *is defined and* $E(s[\alpha+1])$ *is not defined (we are after line (1) in iteration number* α*), then* $\mathcal{CS}(\mathfrak{C}) = \mathsf{repr}^{-1}(E(s'[\alpha]))$.

Definition 6 (Invariant). *Let* $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ *be a CryptoVerif configuration and* $\mathcal{C}$ *be an OCaml configuration. We say that* $\mathfrak{C} \equiv \mathcal{C}$ *if and only if* $\mathcal{CS}(\mathfrak{C}), E, \mathcal{T}, \mathcal{Q}, \mathcal{E} \equiv \mathcal{C}$.

We say that a CryptoVerif configuration $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ is at a *checkpoint* when the process P corresponds to the process at lines marked (1), (2), (4), (7) of Fig. 3 and the stack $\mathcal{R}$ contains one element or P is end and $\mathcal{R}$ is empty. The following lemma shows that the invariant of Definition 6 is preserved at checkpoints during execution.

Lemma 3. *Let* $\mathfrak{CT}_1, \ldots, \mathfrak{CT}_n$ *be CryptoVerif traces starting at* $\mathfrak{C}$*, such that* $\mathfrak{C}$ *and the last configuration of these traces are at checkpoints, and there is no other checkpoint in these traces. Let* $\mathcal{C}$ *be an OCaml configuration such that* $\mathfrak{C} \equiv \mathcal{C}$.

Then there exist disjoint sets of OCaml traces $\mathcal{CTS}_1, \dots, \mathcal{CTS}_n$ all starting at $\mathcal{C}$ such that none of these traces is a prefix of another of these traces, for all $i \leq n$, $\Pr[\mathfrak{CT}_i] = \Pr[\mathcal{CTS}_i]$, and if $\mathfrak{C}'$ is the last configuration of $\mathfrak{CT}_i$ and $\mathcal{C}'$ is the last configuration of a trace in $\mathcal{CTS}_i$, then $\mathfrak{C}' \equiv \mathcal{C}'$. Furthermore, if $\mathfrak{C}'$ cannot be reduced, then neither can $\mathcal{C}'$.

From this lemma, we can prove:

Proposition 1. *Let $\mathfrak{CT}_1, \dots, \mathfrak{CT}_n$ be complete CryptoVerif traces starting at $\mathfrak{C}_0(Q_0, pe_0)$. Then there exist disjoint sets of complete OCaml traces $\mathcal{CTS}_1, \dots, \mathcal{CTS}_n$ all starting at $\mathcal{C}_0(Q_0, pe_0)$ such that for all $i \leq n$, $\Pr[\mathfrak{CT}_i] = \Pr[\mathcal{CTS}_i]$, and if $\mathfrak{C}$ is the last configuration of $\mathfrak{CT}_i$ and $\mathcal{C}$ is the last configuration of a trace in $\mathcal{CTS}_i$, then $\mathfrak{C} \equiv \mathcal{C}$.*

8 Security Result

CryptoVerif security properties are defined using distinguishers D which are functions that take a list of events $\mathcal{E}$ and return true or false. We denote by $\Pr[\mathfrak{C} : D]$ the probability of the set of complete CryptoVerif traces starting at $\mathfrak{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = \text{true}$. For instance, to show that a protocol Q_0 satisfies a correspondence of the form "for all a, if $e_1(a)$ has been executed, then $e_2(a)$ has also been executed", we define D by $D(\mathcal{E}) = \text{true}$ if and only if the correspondence does not hold, that is, $\mathcal{E}$ contains $e_1(a)$ but not $e_2(a)$ for some a. Then we bound the probability $\Pr[\mathfrak{C}_{\mathrm{i}}(Q_0 \mid Q_{\mathrm{adv}}) : D]$, that is, the probability that the adversary Q_{adv} breaks the correspondence in Q_0. We can also define secrecy using events and distinguishers [5]. We use a similar definition in OCaml: $\Pr[\mathcal{C} : D]$ is the probability of the set of complete OCaml traces starting at $\mathcal{C}$ and such that the list of events *events* in their last configuration satisfies $D(\mathbb{G}_{\mathsf{ev}}^{-1}(\mathit{events})) = \text{true}$. Our main theorem is then:

Theorem 1 (Security result). $\Pr[\mathfrak{C}_0(Q_0, pe_0) : D] = \Pr[\mathcal{C}_0(Q_0, pe_0) : D]$.

This theorem is easy to prove from Proposition 1, noticing that, when $\mathfrak{C} \equiv \mathcal{C}$ and the events of $\mathfrak{C}$ are $\mathcal{E}$, the events of $\mathcal{C}$ are $\mathbb{G}_{\mathsf{ev}}(\mathcal{E})$ (Definition 4, Item 11).

In other words, the adversary pe_0 against the OCaml program has the same probability of breaking the security property as the adversary $Q_{\mathrm{adv}}(Q_0, pe_0)$ against the CryptoVerif process. If CryptoVerif bounds the probability $\Pr[\mathfrak{C}_0(Q_0, pe_0) : D]$, then the same bound also holds for the generated implementation.

As detailed in [7], CryptoVerif shows that our model of the SSH Transport Layer Protocol guarantees the authentication of the server to the client and the secrecy of the session keys. By Theorem 1, our generated implementation of this protocol satisfies the same properties, provided assumptions A1 to A6 hold.

9 Conclusion

We have proved that our compiler preserves security. Therefore, by using CryptoVerif, we can prove the desired security properties on the protocol specification, and then by using our compiler, we get a runnable implementation of the protocol, which satisfies the same security properties as the specification. Making such a proof is also useful because it clarifies the assumptions needed to ensure that the implementation is secure (Assumptions A1 to A6 in our case). The proof technique presented in this paper, simulating any adversary by a CryptoVerif process, is also useful to show that any Turing machine can be encoded as a CryptoVerif adversary, which is important for the validity of the verification by CryptoVerif. We have done the proof by hand. Formalizing it using a proof assistant (e.g. Coq) would be interesting future work.

Acknowledgments. This work was partly done while the authors were at École Normale Supérieure, Paris. It was partly supported by the ANR project ProSe (decision ANR 2010-VERS-004).

References

1. Aizatulin, M., Gordon, A.D., Jürjens, J.: Extracting and verifying cryptographic models from C protocol code by symbolic execution. In: CCS'11. pp. 331–340. ACM, New York (2011)
2. Aizatulin, M., Gordon, A.D., Jürjens, J.: Computational verification of C protocol implementations by symbolic execution. In: CCS'12. pp. 712–723. ACM, New York (2012)
3. Bhargavan, K., Fournet, C., Gordon, A., Tse, S.: Verified interoperable implementations of security protocols. ACM TOPLAS 31(1) (2008)
4. Blanchet, B.: A computationally sound mechanized prover for security protocols. IEEE Transactions on Dependable and Secure Computing 5(4), 193–207 (2008)
5. Blanchet, B.: Automatically verified mechanized proof of one-encryption key exchange. In: CSF'12. pp. 325–339. IEEE, Los Alamitos (2012)
6. Cadé, D., Blanchet, B.: Proved generation of implementations from computationally secure protocol specifications, available at http://prosecco.gforge.inria.fr/personal/dcade/post2013full.pdf
7. Cadé, D., Blanchet, B.: From computationally-proved protocol specifications to implementations. In: ARES'12. pp. 65–74. IEEE, Los Alamitos (2012)
8. Chaki, S., Datta, A.: ASPIER: An automated framework for verifying security protocol implementations. In: CSF'09. pp. 172–185. IEEE, Los Alamitos (2009)
9. Dupressoir, F., Gordon, A.D., Jürjens, J., Naumann, D.A.: Guiding a general-purpose C verifier to prove cryptographic protocols. In: CSF'11. pp. 3–17. IEEE, Los Alamitos (2011)
10. Fournet, C., Kohlweiss, M., Strub, P.Y.: Modular code-based cryptographic verification. In: CCS'11. pp. 341–350. ACM, New York (2011)
11. http://msr-inria.inria.fr/projects/sec/fs2cv/
12. Milicia, G.: χ-spaces: Programming security protocols. In: NWPT'02 (2002)
13. Owens, S.: A sound semantics for OCaml light. In: Drossopoulou, S. (ed.) ESOP'08. LNCS, vol. 4960, pp. 1–15. Springer, Heidelberg (2008)
14. Pironti, A., Sisto, R.: Provably correct Java implementations of spi calculus security protocols specifications. Computers and Security 29(3), 302–314 (2010)
15. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bharagavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: ICFP'11. pp. 266–278. ACM, New York (2011)