Proved Generation of Implementations from Computationally Secure Protocol Specifications*

David Cadé and Bruno Blanchet
INRIA Paris-Rocquencourt
23 avenue d'Italie, 75013 Paris, France
{david.cade,bruno.blanchet}@inria.fr

August 23, 2013

Abstract

In order to obtain implementations of security protocols proved secure in the computational model, we previously proposed the following approach: we write a specification of the protocol in the input language of the computational protocol verifier CryptoVerif, prove it secure using CryptoVerif, then generate an OCaml implementation of the protocol from the CryptoVerif specification using a specific compiler that we have implemented. 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.

Keywords: cryptographic protocol, computational model, implementation, compiler, CryptoVerif, OCaml, verification

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.

*This paper is an extended version of the work originally presented at the 2nd Conference on Principles of Security and Trust (POST 2013), Rome, Italy, March 2013 [9].

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. In contrast, 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 [8]. 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 [5, 7, 6]. 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 [8]. 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. [15]. 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. There-

fore, 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 obtain this proof; the most 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. The network code does not mutate bitstrings passed to or received from generated code. This property can be guaranteed by representing bitstrings by an immutable OCaml type. However, the most natural type for representing bitstrings is the OCaml type `string`, which is mutable. Immutable strings can be implemented in OCaml using an abstract type instead of `string`. In our semantics, strings 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. [14, 16]. Other approaches analyze implementations by extracting a specification verified by a symbolic protocol verifier, e.g. [4, 1], or analyze them by other tools such as the model-checker ASPIER [10], the general-purpose C verifier VCC [11], or typing [3, 17].

In contrast, the following approaches provide computational security guarantees, by analyzing implementations. The tool FS2CV [13] translates a subset of F# to the input language of CryptoVerif, which can then prove the protocol secure. The tool F7 [3], which uses a dependent type system to prove security properties on protocols implemented in F#, has been adapted to the computational model in [12]; 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 instrumentation of the OCaml semantics that we use to facilitate our proof. Section 5 describes the compiler itself. Section 6 presents our proof. Because of the length of the proof, the details of the proof of some lemmas are postponed to the appendix.

Notations. Let us introduce some basic notations. When f is a function, we denote by $\mathrm{Dom}(f)$ the domain of f, that is, the set of elements x such that $f(x)$ is defined. We denote by $f[x \mapsto y]$ the function f' defined by $f'(x) = y$ and $f'(x') = f(x')$ for $x' \neq x$. When f_1 and f_2 are functions with disjoint domains, we denote by $f_1 \cup f_2$ the function f' defined by $f'(x) = f_1(x)$ if $x \in \mathrm{Dom}(f_1)$ and $f'(x) = f_2(x)$ if $x \in \mathrm{Dom}(f_2)$. When f_1 and f_2 are functions, we write $f_1 \subseteq f_2$ (or $f_2 \supseteq f_1$) when $\mathrm{Dom}(f_1) \subseteq \mathrm{Dom}(f_2)$ and, for all $x \in \mathrm{Dom}(f_1)$, we have $f_2(x) = f_1(x)$. We denote by $\emptyset$ the function whose domain is the empty set $\emptyset$.

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. CryptoVerif supports two input languages: the channel and oracle front-ends. The channel front-end [5] uses channels to pass data between the adversary and the protocol, and the oracle front-end [7] defines oracles that can be called by the adversary. In this paper, we focus on the oracle front-end, which is closer to the syntax of games used by cryptographers; oracles are also easier to translate into OCaml functions. (Our compiler also supports the channel front-end.) We adapt the semantics given in [5] for the channel front-end to the oracle front-end.

2.1 Syntax and Informal Semantics

Let us first introduce the syntax of the CryptoVerif language in Figure 1. The language is typed, and types T are subsets of $\mathit{bitstring}_\perp \stackrel{\text{def}}{=} \mathit{bitstring} \cup \{\perp\}$ 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 $\mathit{bool} \stackrel{\text{def}}{=} \{\mathrm{true}, \mathrm{false}\}$, where true is the bitstring 1 and false 0, and the types *bitstring* and $\mathit{bitstring}_\perp$ are predefined.

Variables $x[i_1, \ldots, i_m]$ 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

$M ::=$	terms
$\quad x[\widetilde{i}]$	variable access
$\quad f(M_1, \ldots, M_m)$	function application
$Q ::=$	oracle definitions
$\quad 0$	nil
$\quad Q \mid Q'$	parallel composition
$\quad \mathsf{foreach}\ i \leq N\ \mathsf{do}\ Q$	replication N times
$\quad O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P$	oracle definition
$P ::=$	oracle bodies
$\quad \mathsf{return}(M_1, \ldots, M_k); Q$	return
$\quad \mathsf{end}$	end
$\quad x[\widetilde{i}] \stackrel{R}{\leftarrow} T; P$	random number
$\quad x[\widetilde{i}] \leftarrow M; P$	assignment
$\quad \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P'$	conditional
$\quad \mathsf{insert}\ Tbl(M_1, \ldots, M_k); P$	insert in table
$\quad \mathsf{get}\ Tbl(x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$	get from table
$\quad \mathsf{event}\ e(M_1, \ldots, M_k); P$	event
$\quad \mathsf{let}\ (x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) = O[M_1, \ldots, M_l](M'_1, \ldots, M'_k)\ \mathsf{in}\ P\ \mathsf{else}\ P'$	oracle call
$\quad \mathsf{let}\ x[\widetilde{i}] : T = \mathsf{loop}\ O[M_1, \ldots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P'$	loop

Figure 1: Syntax of the CryptoVerif language

abbreviate $i_1, \ldots, i_m$ by $\widetilde{i}$. Each function f comes with its type $T_1 \times \cdots \times T_m \to T$; 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 $x[i_1, \ldots, i_m]$ or function applications $f(M_1, \ldots, M_m)$.

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 $\mathsf{foreach}\ i \leq N\ \mathsf{do}\ Q$ provides N copies of Q, indexed by $i \in \{1, \ldots, N\}$. The bound 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 $O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P$ makes available the oracle $O[\widetilde{i}]$; when $O[\widetilde{i}]$ is called by the adversary with arguments $a_1, \ldots, a_k$, 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 $\mathsf{return}(M_1, \ldots, M_k); Q$ returns the result of $M_1, \ldots, M_k$ to the caller, and makes

available oracles in Q. An end statement end returns to the caller with an error. A random number assignment $x[\widetilde{i}] \stackrel{R}{\leftarrow} T; P$ stores a uniformly chosen random value of type T in variable $x[\widetilde{i}]$, and continues by executing P. The type T must consist of all bitstrings of a given size; in this case, we say that T is a *fixed-length* type. An assignment $x[\widetilde{i}] \leftarrow M; P$ puts the result of M in the variable $x[\widetilde{i}]$, and continues by executing 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 $\mathsf{insert}\ Tbl(M_1, \ldots, M_k); P$ inserts the result of $M_1, \ldots, M_k$ into the table Tbl. Tables are lists of tuples, used for example to store tables of keys. Each table Tbl has a type $T_1 \times \cdots \times T_k$, which means that Tbl contains k-uples $a_1, \ldots, a_k$ such that a_j is of type T_j for all $j \leq k$. A get statement $\mathsf{get}\ Tbl(x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$ searches for an element $a_1, \ldots, a_k$ in the table Tbl such that the term M evaluates to true when $x_1[\widetilde{i}] = a_1, \ldots, x_k[\widetilde{i}] = a_k$. If there is no such element, we continue by executing P', and otherwise we choose randomly one of the elements that correspond, store it in the variables $x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]$, then execute P. An event statement $\mathsf{event}\ e(M_1, \ldots, M_k); P$ 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 $\mathsf{let}\ (x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) = O[M_1, \ldots, M_l](M'_1, \ldots, M'_k)$ $\mathsf{in}\ P\ \mathsf{else}\ P'$ calls oracle $O[M_1, \ldots, M_l]$ with arguments $M'_1, \ldots, M'_k$, stores its returned values in the variables $x_1[\widetilde{i}], \ldots, x_{k'}[\widetilde{i}]$, and continues by executing P if the oracle terminates with a return statement, or continues by executing P' if the oracle terminates with end.

A loop $\mathsf{let}\ x[\widetilde{i}] : T = \mathsf{loop}\ O[M_1, \ldots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P'$ calls oracle O in a loop. Oracle O takes a unique argument (the internal state of the loop) and returns a pair containing the modified internal state of the loop and a boolean indicating whether the loop should continue or not. $O[M_1, \ldots, M_l](M')$ is first called. If it returns (a_1, true), $O[M_1 + 1, M_2, \ldots, M_l](a_1)$ is called. If it returns (a_2, true), $O[M_1 + 2, M_2, \ldots, M_l](a_2)$ is called, and so on, until $O[M_1 + k, M_2, \ldots, M_l](a_k)$ 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.

Example 1 Let us consider a simple protocol in which the first participant Alice generates a nonce m, sends it to the second participant Bob with a signature of the nonce under Alice's signature key sk. Bob then verifies that the signature is correct using Alice's public key pk. This protocol can be described by the

following CryptoVerif process:

$$
\begin{array}{l}
\mathrm{Ostart}() := \\
\quad rk \xleftarrow{R} keyseed; pk \leftarrow \mathrm{pkgen}(rk); sk \leftarrow \mathrm{skgen}(rk); \\
\quad \mathsf{return}(pk); (\mathsf{foreach}\ i_1 \leq N_1\ \mathsf{do}\ P_A \mid \mathsf{foreach}\ i_2 \leq N_2\ \mathsf{do}\ P_B) \\
P_A \stackrel{\mathrm{def}}{=} \mathrm{OA}() := \\
\quad m \xleftarrow{R} nonce; s \xleftarrow{R} seed; \mathsf{return}(m, \mathrm{sign}(m, sk, s)) \\
P_B \stackrel{\mathrm{def}}{=} \mathrm{OB}(m' : nonce, s' : signature) := \\
\quad \mathsf{if}\ \mathrm{check}(m', pk, s')\ \mathsf{then}\ \mathsf{return}()\ \mathsf{else}\ \mathsf{end}
\end{array}
$$

The only callable oracle at the beginning is the oracle Ostart, which generates the signature key pair (pk, sk) by first generating a random seed rk and applying the key generation algorithms pkgen and skgen to it. We return to the attacker the public key, so that the attacker can check whether a signature signed with the signature key sk is correct. When the oracle Ostart returns, one can call the oracle OA N_1 times, and the oracle OB N_2 times.

The oracle OA generates a random nonce m and a random seed s, and returns the nonce m and the signature of the nonce m under the signature key sk with the random seed s.

The oracle OB takes as arguments a nonce m' and a signature s', which should be the elements returned by a call to oracle OA, and checks using the function check whether the signature s' is indeed a correct signature of the message m' under the signature key sk by using the public key pk. If the signature is correct, the oracle returns normally. Otherwise, the oracle terminates with end.

2.2 Formal Semantics

We present the semantics of the language in Figures 2, 3, and 4. 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 array cells $x[\widetilde{a}]$ to their contents, where x is a variable, $\widetilde{a}$ gives the value of its replication indices, and the contents of $x[\widetilde{a}]$ is a bitstring value.
- The oracle body P is the oracle body currently running.
- The mapping $\mathcal{T}$ maps table names to their contents, which is the list 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

Terms:

$$E, a \Downarrow a \tag{Cst}$$

$$\frac{x[a_1, \ldots, a_m] \in \mathrm{Dom}(E)}{E, x[a_1, \ldots, a_m] \Downarrow E(x[a_1, \ldots, a_m])} \tag{Var}$$

$$\frac{\begin{array}{c}\forall j \leq m, E, M_j \Downarrow a_j \qquad f : T_1 \times \cdots \times T_m \to T \\ \forall j \leq m, a_j \in T_j\end{array}}{E, f(M_1, \ldots, M_m) \Downarrow f(a_1, \ldots, a_m)} \tag{Fun}$$

Oracle definitions:

$$\mathrm{reduce}(0) \stackrel{\mathrm{def}}{=} \emptyset \tag{Nil}$$

$$\mathrm{reduce}(Q_1 \mid Q_2) \stackrel{\mathrm{def}}{=} \mathrm{reduce}(Q_1) \cup \mathrm{reduce}(Q_2) \tag{Par}$$

$$\mathrm{reduce}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) \stackrel{\mathrm{def}}{=} \bigcup_{a=1}^{n} \mathrm{reduce}(Q\{a/i\}) \tag{Repl}$$

$$\begin{array}{c}\mathrm{reduce}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\mathrm{def}}{=} \\ \{(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P\}\end{array} \tag{Oracle}$$

Figure 2: Semantics (1)

first will be executed if the oracle returns a result with a return statement, and the second will be executed if the oracle terminates with an end statement.

- The list $\mathcal{E}$ is the list of events $e(a_1, \ldots, a_k)$ executed so far, by the construct event $e(M_1, \ldots, M_k)$.

During execution, terms may be reduced into constant bitstrings, so we add constant bitstrings a to the grammar of terms M. The notation $E, M \Downarrow a$ means that the term M evaluates to the bitstring a under the environment E. This relation is defined by rules (Cst), (Var), and (Fun) in Figure 2. The set $\mathrm{reduce}(Q)$, also defined in Figure 2, contains all oracle definitions provided by the oracle definition Q, with replication indices instantiated to all their possible values.

The semantics is defined by probabilistic reduction rules between configurations: $\mathfrak{C} \to_p \mathfrak{C}'$ means that $\mathfrak{C}$ reduces into $\mathfrak{C}'$ with probability p. This relation is defined in Figures 3 and 4. We use the following notations for lists. Let $[\,]$ be the empty list, and $x :: l$ be the list obtained by adding the element x to the list l. Let $[x_1; \ldots; x_k]$ be the list $x_1 :: \ldots :: x_k :: [\,]$. Let $[x \in l \mid \mathit{Prop}(x)]$ be the list containing all elements x of l that satisfy the property $\mathit{Prop}(x)$, in the same

Oracle bodies (1):

$$\frac{T \text{ fixed-length type} \qquad a \in T}{E, x[\widetilde{a'}] \xleftarrow{R} T; P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_{\frac{1}{|T|}} E[x[\widetilde{a'}] \mapsto a], P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}} \quad \text{(New)}$$

$$\frac{E, M \Downarrow a}{E, x[\widetilde{a'}] \leftarrow M; P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_1 E[x[\widetilde{a'}] \mapsto a], P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}} \quad \text{(Let)}$$

$$\frac{E, M \Downarrow \text{true}}{E, \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_1 E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}} \quad \text{(If1)}$$

$$\frac{E, M \Downarrow \text{false}}{E, \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}} \quad \text{(If2)}$$

$$\frac{\forall j \le k, \quad E, M_j \Downarrow a_j}{\begin{array}{l} E, \mathsf{insert}\ Tbl(M_1, \ldots, M_k); P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_1 \\ \quad E, P, \mathcal{T}[Tbl \mapsto (a_1, \ldots, a_k) :: \mathcal{T}(Tbl)], \mathcal{Q}, \mathcal{R}, \mathcal{E} \end{array}} \quad \text{(Insert)}$$

$$\frac{\begin{array}{c} l = [(a_1, \ldots, a_k) \in \mathcal{T}(Tbl) \mid E[x_1[\widetilde{a'}] \mapsto a_1, \ldots, x_k[\widetilde{a'}] \mapsto a_k], M \Downarrow \text{true}] \\ (a_1^0, \ldots, a_k^0) \in l \\ S = \{1 \le j \le |l| \mid \mathsf{nth}(l, j) = (a_1^0, \ldots, a_k^0)\} \end{array}}{\begin{array}{l} E, \mathsf{get}\ Tbl(x_1[\widetilde{a'}], \ldots, x_k[\widetilde{a'}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \\ \quad \to_{\sum_{j \in S} \mathrm{among}(\{1,\ldots,|l|\}, j)} \\ \quad\quad E[x_1[\widetilde{a'}] \mapsto a_1^0, \ldots, x_k[\widetilde{a'}] \mapsto a_k^0], P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \end{array}} \quad \text{(Get1)}$$

$$\frac{[(a_1, \ldots, a_k) \in \mathcal{T}(Tbl) \mid E[x_1[\widetilde{a'}] \mapsto a_1, \ldots, x_k[\widetilde{a'}] \mapsto a_k], M \Downarrow \text{true}] = [\,]}{\begin{array}{l} E, \mathsf{get}\ Tbl(x_1[\widetilde{a'}], \ldots, x_k[\widetilde{a'}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_1 \\ \quad E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \end{array}} \quad \text{(Get2)}$$

Figure 3: Semantics (2)

Oracle bodies (2):

$$\frac{\begin{array}{c}\forall i \le l, E, M_i \Downarrow a'_i \qquad \widetilde{a'} = a'_1, \ldots, a'_l \qquad \forall j \le k, E, N_j \Downarrow b_j \\ \exists x'_1, \ldots, x'_k, P'' \text{ such that} \\ Q_0 = (O[\widetilde{a'}](x'_1[\widetilde{a'}] : T'_1, \ldots, x'_k[\widetilde{a'}] : T'_k) := P'') \in \mathcal{Q} \\ E' = E[x'_1[\widetilde{a'}] \mapsto b_1, \ldots, x'_k[\widetilde{a'}] \mapsto b_k]\end{array}}{\begin{array}{l}E, \mathsf{let}\ (x_1[\widetilde{a}] : T_1, \ldots, x_{k'}[\widetilde{a}] : T_{k'}) = O[M_1, \ldots, M_l](N_1, \ldots, N_k) \\ \quad \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \quad \to_1 \\ \quad E', P'', \mathcal{T}, \mathcal{Q} \setminus \{Q_0\}, ((x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]), P, P') :: \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Call)}$$

$$\frac{\forall j \le k, E, N_j \Downarrow b_j \qquad \mathcal{Q}' = \mathrm{reduce}(Q'')}{\begin{array}{l}E, \mathsf{return}(N_1, \ldots, N_k); Q'', \mathcal{Q}, ((x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), P, P') :: \mathcal{R}, \mathcal{E} \\ \quad \to_1 E[x_1[\widetilde{a}] \mapsto b_1, \ldots, x_k[\widetilde{a}] \mapsto b_k], P, \mathcal{T}, \mathcal{Q} \cup \mathcal{Q}', \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Return)}$$

$$E, \mathsf{end}, \mathcal{T}, \mathcal{Q}, ((x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]), P, P') :: \mathcal{R}, \mathcal{E} \to_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \quad \text{(End)}$$

$$\frac{\forall j \le l, E, M_j \Downarrow a_j}{E, \mathsf{event}\ ev(M_1, \ldots, M_l); P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_1 E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, e(a_1, \ldots, a_l) :: \mathcal{E}} \quad \text{(Event)}$$

$$\frac{\begin{array}{c}\forall i \le l, E, M_i \Downarrow a'_i \qquad E, M' \Downarrow c \\ \text{the last replication above the definition of } O \text{ is } \mathsf{foreach}\ i_1 \le N_1 \qquad a'_1 \le N_1\end{array}}{\begin{array}{l}E, \mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[M_1, \ldots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \\ \quad \to_1 E, \\ \quad\quad (\mathsf{let}\ (r'_{a'_1,r}[\widetilde{a}] : T, b_{a'_1,r}[\widetilde{a}] : bool) = O[a'_1, \ldots, a'_l](c)\ \mathsf{in} \\ \quad\quad\quad \mathsf{if}\ b_{a'_1,r}[\widetilde{a}]\ \mathsf{then} \\ \quad\quad\quad\quad (\mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[a'_1 + 1, a'_2, \ldots, a'_l](r'_{a'_1,r}[\widetilde{a}] : T) \\ \quad\quad\quad\quad\ \mathsf{in}\ P\ \mathsf{else}\ P') \\ \quad\quad\quad \mathsf{else}\ r[\widetilde{a}] \leftarrow r'_{a'_1,r}[\widetilde{a}]; P \\ \quad\quad \mathsf{else}\ P'), \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Loop1)}$$

$$\frac{\begin{array}{c}\forall i \le l, E, M_i \Downarrow a'_i \qquad E, M' \Downarrow c \\ \text{the last replication above the definition of } O \text{ is } \mathsf{foreach}\ i_1 \le N_1 \qquad a'_1 > N_1\end{array}}{\begin{array}{l}E, \mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[M_1, \ldots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \\ \quad \to_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Loop2)}$$

Figure 4: Semantics (3)

order as in l. This construct is defined by induction on lists:

$$[x \in [\,] \mid \mathit{Prop}(x)] \stackrel{\text{def}}{=} [\,],$$
$$[x \in y :: l \mid \mathit{Prop}(x)] \stackrel{\text{def}}{=} \begin{cases} [x \in l \mid \mathit{Prop}(x)] & \text{if } \neg \mathit{Prop}(y), \\ y :: [x \in l \mid \mathit{Prop}(x)] & \text{otherwise}. \end{cases}$$

The concatenation of lists $l_1 @ l_2$ is the list containing all elements of l_1 followed by all elements of l_2. The membership test $x \in l$ is true when l contains the element x, and false otherwise. Let $|l|$ be the length of the list l, and $\mathsf{nth}(l, n)$ be the nth element of list l.

The rule (New) evaluates $x[\widetilde{a'}] \stackrel{R}{\leftarrow} T$ by choosing an element $a \in T$ and storing it in $E(x[\widetilde{a'}])$. The element $a \in T$ is chosen uniformly, so the probability of each choice is $1/|T|$ and this is possible only when T is a fixed-length type. The rule (Let) evaluates the term M and stores its value in $E(x[\widetilde{a'}])$. The rules (If1) and (If2) are straightforward.

The rules (Insert), (Get1), and (Get2) deal with tables of keys. The rule (Insert) evaluates the inserted element and adds it to the table Tbl, by adding it to the list $\mathcal{T}(\mathit{Tbl})$. The rules (Get1) and (Get2) compute the list of elements that satisfy the condition of the get. When this list is empty, the else branch is taken by rule (Get2). When this list is not empty, the rule (Get1) chooses an element of this list l, stores it in $E(x_1[\widetilde{a'}]), \ldots, E(x_k[\widetilde{a'}])$, and takes the in branch. The j-th element of the list l is chosen with probability $\mathrm{among}(\{1, \ldots, |l|\}, j)$, where $\mathrm{among}(S, b)$ is the probability that the element $b \in S$ is chosen among elements of the set S, according to an almost uniform distribution: we require that, for every set S, $\sum_{b \in S} \mathrm{among}(S, b) = 1$, $\mathrm{among}(S, b) > 0$ for all $b \in S$, and $\sum_{b \in S} \left|\mathrm{among}(S, b) - \frac{1}{|S|}\right| \leq \epsilon$ for some $\epsilon > 0$. Indeed, probabilistic Turing machines can choose random elements uniformly only in sets of cardinal a power of 2. For other sets, they can choose random elements with a probability distribution as close as we wish to uniform, that is, we can make ϵ as small as we wish in the formula above. In case the same element $a_1^0, \ldots, a_k^0$ occurs several times in the list l, the probability of choosing that element is the sum of the probabilities of all its occurrences. The probability of choosing $a_1^0, \ldots, a_k^0$ is then close to $m/|l|$, where m is the number of times this element appears in l.

The rule (Call) implements the oracle call $\mathsf{let}\ (x_1[\widetilde{a}] : T_1, \ldots, x_{k'}[\widetilde{a}] : T_{k'}) = O[M_1, \ldots, M_l](N_1, \ldots, N_k)\ \mathsf{in}\ P\ \mathsf{else}\ P'$. It evaluates the indices $M_1, \ldots, M_l$ of the oracle to call into $\widetilde{a'}$ and its arguments $N_1, \ldots, N_k$ into $b_1, \ldots, b_k$; after evaluation, we want to call the oracle $O[\widetilde{a'}](b_1, \ldots, b_k)$. Then, it looks for the definition Q_0 of the oracle $O[\widetilde{a'}]$ in the callable oracles $\mathcal{Q}$. It calls Q_0 by removing it from the callable oracles, storing $b_1, \ldots, b_k$ in the arguments of Q_0, and running its body P''. The element $(x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]), P, P')$ is pushed on the stack $\mathcal{R}$: $x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]$ are the variables in which the return value of Q_0 should be stored, P is the process to execute when Q_0 returns, and P' is the process to execute when Q_0 terminates with end. The rule (Return) pops an element $((x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]), P, P')$ from the stack, stores the return value in $x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]$, and executes P. It adds to the set of callable oracles $\mathcal{Q}$ the

oracles $\mathcal{Q}'$ defined in the oracle definition Q'' located after the return statement. The rule (End) also pops an element $((x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]), P, P')$ from the stack, but executes the process P'. The rule (Event) adds the executed event to the list of events $\mathcal{E}$.

The rules (Loop1) and (Loop2) implement the loop statement. The rule (Loop1) performs one iteration of the loop. To that effect, it creates two fresh variable names $r'_{a'_1,r}$ and $b_{a'_1,r}$, calls the oracle O and stores its return values in these variables. When the boolean $b_{a'_1,r}[\widetilde{a}]$ returned by O is false, it ends the loop and continues by executing P with the result $r[\widetilde{a}]$ bound to the value of $r'_{a'_1,r}[\widetilde{a}]$. When $b_{a'_1,r}[\widetilde{a}]$ is true, it reruns the loop. If the oracle O terminates with an end statement, it ends the loop and continues by executing P'. The rule (Loop2) handles the case in which the loop stops by reaching the bound N_1 of the loop index.

The initial configuration for running the oracle definition Q is $\mathfrak{C}_{\mathrm{i}}(Q_0) \stackrel{\mathrm{def}}{=} \emptyset, \mathsf{let}\ x[\,] : \mathit{bitstring} = O_{\mathrm{start}}()\ \mathsf{in}\ \mathsf{return}(x)\ \mathsf{else}\ \mathsf{end}, \mathcal{T}_0, \mathrm{reduce}(Q_0), \emptyset, [\,]$, where $\mathcal{T}_0(\mathit{Tbl}) = [\,]$ for all tables Tbl. This configuration starts by calling oracle O_{start}. The oracle definition Q_0 typically contains a protocol in parallel with an adversary.

CryptoVerif verifies the following requirements on Q_0:

Property 2.1 *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.*

Property 2.2 *The processes are well-typed. (In particular, functions and oracles receive arguments of their expected types. For brevity, we do not detail the type system; see [5] for a similar type system.)*

Property 2.3 *Oracles with the same name can be defined only in different branches of an* if *or* get *construct. In an oracle definition $O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P$, the indices $\widetilde{i}$ are always the indices of replications above that oracle definition.*

Property 2.4 *We define types of oracles as follows. The type of a* return$(M_1, \ldots, M_k); Q$ *statement consists of the types of $M_1, \ldots, M_k$ and the list of types of the oracle definitions at the beginning of Q, ordered from left to right. The type of an oracle definition consists of 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.*

Property 2.1 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 Property 2.1. Property 2.2 requires

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 bitstrings may appear at each point of the protocol. Property 2.4 guarantees that the various definitions of an oracle are consistent, and can in fact be compiled into a single function in OCaml. Property 2.3 guarantees that there exists a single callable definition for each oracle. This property is formalized by the following lemma, proved in Appendix A.

Lemma 2.5 (Oracle name and indices unicity) *If the configuration $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ is reachable from the initial configuration $\mathfrak{C}_{\mathrm{i}}(Q_0)$ by reductions $\to_p$, then the set of callable oracles $\mathcal{Q}$ contains at most one oracle with a given name O and given replication indices $\widetilde{a}$.*

This lemma proves that the rule (Call) is deterministic. Therefore, all rules are deterministic, except the rules (New) and (Get1) which may make probabilistic choices.

Another interesting point is that, if a configuration $\mathfrak{C}$ reduces into another configuration, then the sum of the probabilities of all the possible reductions from $\mathfrak{C}$ is 1:

$$\sum_{\{\mathfrak{C}' | \mathfrak{C} \to_{p(\mathfrak{C}')} \mathfrak{C}'\}} p(\mathfrak{C}') = 1\,.$$

Definition 2.6 (Traces) *Let us denote traces with the symbol $\mathfrak{CT}$. A trace is a sequence of reductions $\mathfrak{CT} = \mathfrak{C}_0 \to_{p_1} \cdots \to_{p_n} \mathfrak{C}_n$ where $\mathfrak{C}_0, \ldots, \mathfrak{C}_n$ are semantic configurations such that $\mathfrak{C}_i \to_{p_{i+1}} \mathfrak{C}_{i+1}$ for $i = 0, \ldots, n-1$.*

A complete *trace is a trace such that there is no possible reduction from its last configuration.*

The probability of the trace $\mathfrak{CT}$ is $\Pr[\mathfrak{CT}] = p_1 \times \cdots \times p_n$. When the traces in a set of traces $\mathfrak{CTS}$ are not prefix of one another, the probability of $\mathfrak{CTS}$ is the sum of the probabilities of its elements.

The notation $\mathfrak{C} \to_p^* \mathfrak{C}'$ means that there exists a trace beginning at $\mathfrak{C}$ and ending at $\mathfrak{C}'$, and p is the probability of the set of all traces beginning at $\mathfrak{C}$ and stopping at their first occurrence of $\mathfrak{C}'$.

The notation $\mathfrak{C} \to_p^+ \mathfrak{C}'$ means that $\mathfrak{C} \to_p^* \mathfrak{C}'$ and $\mathfrak{C} \neq \mathfrak{C}'$, that is, all traces from $\mathfrak{C}$ to $\mathfrak{C}'$ have at least one step.

The notation $\mathfrak{C} \to^* \mathfrak{C}'$ means $\mathfrak{C} \to_1^* \mathfrak{C}'$. We denote the number of steps in the trace $\mathfrak{CT}$ as $|\mathfrak{CT}| = n$.

Intuitively, when traces in $\mathfrak{CTS}$ are not prefix of one another, they correspond to disjoint cases, so the probability of $\mathfrak{CTS}$ is the sum of probabilities of the traces in $\mathfrak{CTS}$. (When $\mathfrak{CT}$ is a prefix of $\mathfrak{CT}'$, the trace $\mathfrak{CT}'$ is a particular case of $\mathfrak{CT}$.)

In CryptoVerif, since for every reduction with a probabilistic choice, the environment E is modified so that we can determine from E which reduction

was used, and one cannot remove elements from E, there will be at most one trace from one configuration to another. However, the notations of Definition 2.6 are also used for OCaml where there could be several configurations reducing to the same configuration, so they support this situation.

2.3 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 by oracle definitions role{Q; the end of role is specified by a closing brace } between a return(...) 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. Storing variables in files is useful for variables that are communicated across roles, for example long-term keys that are set in a key generation program and later used by client and server programs.) 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.

Example 2 Let us annotate the protocol of Example 1.

$$\begin{array}{l}
\mathsf{keygen}[pk > \mathit{pkfile}, sk > \mathit{skfile}]\{\ \mathrm{Ostart}() := \\
\quad rk \xleftarrow{R} \mathit{keyseed}; pk \leftarrow \mathrm{pkgen}(rk); sk \leftarrow \mathrm{skgen}(rk); \\
\quad \mathsf{return}(pk)\ \}; (\mathsf{foreach}\ i_1 \leq N_1\ \mathsf{do}\ P_A \mid \mathsf{foreach}\ i_2 \leq N_2\ \mathsf{do}\ P_B) \\
P_A \stackrel{\mathrm{def}}{=} \mathsf{alice}\{\ \mathrm{OA}() := \\
\quad m \xleftarrow{R} \mathit{nonce}; s \xleftarrow{R} \mathit{seed}; \mathsf{return}(m, \mathrm{sign}(m, sk, s)) \\
P_B \stackrel{\mathrm{def}}{=} \mathsf{bob}\{\ \mathrm{OB}(m' : \mathit{nonce}, s' : \mathit{signature}) := \\
\quad \mathsf{if}\ \mathrm{check}(m', pk, s')\ \mathsf{then}\ \mathsf{return}()\ \mathsf{else}\ \mathsf{end}
\end{array}$$

We divide this process into three parts. First, the key generation part is represented by the role keygen, which contains just the oracle Ostart. The annotation $pk > \mathit{pkfile}, sk > \mathit{skfile}$ means that we store the public key pk in the file *pkfile* so that the oracle OB can access it, and analogously, we store the secret key sk in the file *skfile* so that the oracle OA can access it. In order words, files $= \{(pk[\,], \mathit{pkfile}), (sk[\,], \mathit{skfile})\}$.

The role alice, which contains the oracle OA, corresponds to the role of Alice and the role bob, which contains the oracle OB, corresponds to the role of Bob. For these two roles, there is no need to write the closing braces } because there is nothing after them.

Finally, the user annotations provide, for each CryptoVerif type T, the corresponding OCaml type $\mathbb{G}_T(T)$ as well as several OCaml functions:

- The function $\mathbb{G}_{\mathsf{random}}(T) : \mathsf{unit} \to \mathbb{G}_T(T)$ generates random numbers uniformly in T (when T is used in a random number generation).
- The serialization function $\mathbb{G}_{\mathsf{ser}}(T) : \mathbb{G}_T(T) \to \mathsf{string}$ converts an element of type $\mathbb{G}_T(T)$ to an OCaml string. The deserialization function $\mathbb{G}_{\mathsf{deser}}(T) : \mathsf{string} \to \mathbb{G}_T(T)$ performs the inverse operation. When deserialization fails, it must raise the exception Bad_file; this exception is raised only when a file has been corrupted. These functions are present when values of type T are written or read from tables and files.
- The predicate function $\mathbb{G}_{\mathsf{pred}}(T) : \mathbb{G}_T(T) \to \mathsf{bool}$ 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 $f : T_1 \times \cdots \times T_m \to T$, a corresponding OCaml function $\mathbb{G}_{\mathsf{f}}(f) : \mathbb{G}_T(T_1) \times \cdots \times \mathbb{G}_T(T_m) \to \mathbb{G}_T(T)$. We assume that these functions are all provided in an OCaml module μ_{prim}.

CryptoVerif verifies the following properties:

Property 2.7 *There is a single occurrence of each role* role. *If an oracle O has a* return$(x_1, \ldots, x_k)\}; Q$ *where Q contains a role definition, then there is only one* return *statement for the oracle O in the whole initial process.*

This property guarantees that we know which process to compile for a given role, and which roles start after the return from a given oracle.

Property 2.8 *There are no nested roles.*

Furthermore, for simplicity, we also assume the following points:

Assumption 2.9 *All oracle definitions are included in a role.*

Assumption 2.10 *No replication occurs above a parallel composition or a replication. When the definition of a role* role *is under replication* foreach $i \leq N$ do role$\{Q$*, its contents Q consists of an oracle definition $O[\widetilde{i}](\ldots) := \ldots$ or of a parallel composition of such oracle definitions (without replication).*

A process can be transformed so that no replication occurs above a parallel composition by distributing the replications into the parallel compositions: foreach $i \leq N$ do $(P_1 \mid P_2)$ can be transformed into (foreach $i_1 \leq N_1$ do P_1) | (foreach $i_2 \leq N_2$ do P_2). We can encode nested replications by adding a dummy

oracle between the two replications: the process foreach $i \leq N$ do foreach $j \leq N'$ do P can be transformed into foreach $i \leq N$ do $O() :=$ return(); foreach $j \leq N'$ do P.

By Properties 2.3, 2.4, and 2.7, there cannot be, in the same process, a definition of an oracle O directly under replication and another definition of the same oracle O not directly under replication. Hence, we can use the phrase "O is under replication" unambiguously. Moreover, by Property 2.4, the bound of the replication above a definition of an oracle O is the same for all definitions of O.

Assumption 2.11 *For each oracle O under replication, we let N_O be the bound of the replication above the definition of O. For each role* role *under replication, we let N_{role} be the bound of the replication above the definition of* role*. All these bounds N_O and N_{role} are pairwise distinct.*

This assumption allows us to be more precise when counting the number of times an oracle has been called, by using a distinct bound for each oracle.

These assumptions are relaxed in our implementation.

3 The OCaml Language

This section presents the OCaml language, the target language of our compiler, by giving its syntax and semantics. We omit some constructs, such as loops and type constructors, which are not used by our compiler. The subset that we consider is still Turing complete, so we do not lose expressivity by removing these constructs. To define the formal semantics, we adapted the semantics by Scott Owens et al. [15]. This semantics is a small step operational semantics of the core part of the OCaml language. We modified it in several ways, as detailed below.

3.1 Syntax and Informal Semantics

Figure 5 summarizes the syntax of our subset of OCaml. For brevity, we ignore types in this syntax.

Pattern-matching is a central feature of OCaml. A pattern *pat* describes the form of a value to be matched. When we match a value v with a pattern *pat*, if the value is of the correct form, then we bind each variable x occurring in the pattern *pat* to the corresponding part of v. Patterns must be linear, that is, no variable can occur more than once inside a pattern. When we match a value v with the pattern matching $pat_1 \rightarrow e_1 \mid \ldots \mid pat_n \rightarrow e_n$, we match v sequentially to the patterns $pat_1, \ldots, pat_n$. If the first pattern that matches v is pat_i, then we evaluate e_i. If no pattern matches v, then we raise the exception Match_failure.

The basic operations of the language are implemented by primitives *prim*. We write binary primitives in infix notation: for example, we write $v_1 = v_2$ rather than $(=)\ v_1\ v_2$. We consider the following primitives: not is the boolean

pat ::=	pattern
x	variable
$\_$	universal pattern
$(pat_1, \ldots, pat_n)$	tuple
$pat_1 :: pat_2$	list constructor
pm ::=	pattern matching
$pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n$	pattern matching
e ::=	expression
$prim$	primitive
x	variable
l	location
c	constant ([ ], (), 0, false, ...)
$(e_1, \ldots, e_n)$	tuple
$e_1 :: e_2$	list constructor
function pm	function
$e_1\ e_2$	application
$e_1; e_2$	sequence
if e_1 then e_2 else e_3	if
match e with pm	pattern matching
try e with pm	try
let $pat = e_1$ in e_2	let
let rec $x_1 =$ function pm_1 and ... and $x_n =$ function pm_n in e	let rec
function$[env, pm]$	closure
letrec$[env, \{x_1 \mapsto$ function $pm_1, \ldots, x_n \mapsto$ function $pm_n\}$ in $x_i]$	let rec closure, $1 \leq i \leq n$
addthread$(program)$	addition of a thread
schedule(e)	schedule
d ::=	definition
let $pat = e$	let
let rec $x_1 =$ function pm_1 and ... and $x_n =$ function pm_n	letrec
definitions ::=	definitions
ε	empty definition list
d;; *definitions*	definition list
program ::=	program
definitions	list of definitions
raise e	exception

Figure 5: OCaml syntax

$v ::=$	value
$\mathit{prim}\ v_1\ \ldots\ v_j$	partially applied primitives (prim is n-ary and $0 \leq j < n$)
c	constant ([ ], (), 0, false, ...)
l	location
$(v_1, \ldots, v_n)$	tuple
$v_1 :: v_2$	list constructor
$\mathsf{function}[env, pm]$	closure
$\mathsf{letrec}[env, \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}\ \mathsf{in}\ x_i]$	let rec closure

Figure 6: OCaml values

negation, (=) is the equality test, raise e raises the exception e. We use primitives to manage references, which are mutable memory cells. We represent memory cells by locations l; we also use special locations to represent files. The reference creation ref v creates a new location l, store the value v in l, and returns the location l. The assignment $l := v$ replaces the contents of the location l with the value v. The dereference $!l$ returns the contents of the location l. We also introduce a primitive for random number generation: random () returns a random boolean, true or false, with equal probability. This primitive was not present in [15]. It makes the semantics probabilistic. The language also includes primitives to manage other native types such as integers (e.g., addition and multiplication) and strings (e.g., concatenation, extraction of substrings, and conversion between integers in $\{0, \ldots, 255\}$ and one-character strings). Strings are immutable values in our semantics. In contrast, in OCaml, values of type `string` are mutable. Our strings could be implemented in OCaml as an abstract type, on which only operations that do not mutate strings are implemented.

Most expressions are standard. Constants c can be integers, strings, boolean values true or false, the empty list [ ], the unit constant (), and exceptions. The expression function pm defines a function. When this function is applied to a value v, it matches that value using the pattern matching pm. The application $e_1\ e_2$ applies the function e_1 to the argument e_2. The sequence operation $e_1; e_2$ evaluates e_1, ignoring its result (but obviously keeping its side effects), then evaluates e_2. The matching operation match e with pm evaluates e and matches the result of e using the pattern matching pm. The try construct try e with pm returns the result of e if e does not raise exceptions; if e raises an exception v matched by a pattern in pm, it returns the result of match v with pm; if e raises an exception v that is not matched by a pattern in pm, it also raises the exception v. The let binding let $pat = e_1$ in e_2 evaluates e_1, matches the result with the pattern pat, which binds the variables in pat, and finally evaluates e_2. When the pattern matching fails, it raises the exception Match_failure. This construct is equivalent to match e_1 with $pat \rightarrow e_2$. The let rec binding let rec $x_1 =$ function pm_1 and ... and $x_n =$ function pm_n in e defines n mutually recursive

functions $x_1, \ldots, x_n$, and evaluates the expression e using these functions.

Closures are not present in the initial program, but they serve to represent functional values internally. The closure $\mathsf{function}[env, pm]$ comes from the function $\mathsf{function}\ pm$. It contains the code of the function (pm), and an environment env that maps the free variables of pm to their values. Closures allow one to evaluate functions using the values that the free variables of the function had at the definition of the function. (In other words, OCaml uses static variable binding.) The let rec closure $\mathsf{letrec}[env, \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}\ \mathsf{in}\ x_i]$ is similar, but for mutually recursive functions. It records several mutually recursive bindings together.

A security protocol typically involves several programs running in parallel on different machines. We model this situation by considering several threads. To manage threads, we introduce two new expressions, $\mathsf{addthread}(program)$ and $\mathsf{schedule}(e)$. The expression $\mathsf{addthread}(program)$ creates a new thread that runs the program $program$. The expression $\mathsf{schedule}(e)$ stops execution of the current thread and continues execution of the thread number e. (Threads are designated by integer numbers. The initial thread, started at the beginning of the program, has number 1. The threads created by subsequent calls to addthread have numbers starting at 2 and increasing by one each time a new thread is created.)

We define the list expression $[e_1; e_2; \ldots; e_n]$ as syntactic sugar for $e_1 :: (e_2 :: \ldots :: (e_n :: [\,]) \ldots)$. The expression $e \,\&\&\, e'$ is syntactic sugar for $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}$ false, and $e \,||\, e'$ is syntactic sugar for $\mathsf{if}\ e\ \mathsf{then}$ true $\mathsf{else}\ e'$.

A program is a list of top level definitions d, or the raising of an exception. We omit the final ε in a sequence of definitions when it is not empty.

Expressions reduce into values or exceptional values. As summarized in Figure 6, the values v are either functional values like closures, or tuples and lists of constants c or locations l. An exceptional value is $\mathsf{raise}\ v$, where v is an exception value (a constant).

3.2 Formal Semantics

We define step by step the semantics of the various constructs of the language.

3.2.1 Pattern matching

We define the predicate matches in Figure 7: we have $v\ \mathsf{matches}\ pat \triangleright env$ when the value v matches the pattern pat, and the environment env is a mapping from the variables of pat to their values, computed by the pattern matching. The operation $env \oplus env' \stackrel{\text{def}}{=} env_{|\overline{\mathrm{Dom}(env')}} \cup env'$ adds the bindings of env' to those of env; when a variable is bound in both environments, the binding of env' is kept. Since patterns are linear, in Figure 7, the operation $env \oplus env'$ is always used with environments env and env' that have disjoint domains; the general case is used below. We also define $v\ \mathsf{matches}\ pat$ as $\exists env, v\ \mathsf{matches}\ pat \triangleright env$.

$$v \text{ matches } x \triangleright \{x \mapsto v\} \qquad \text{(Variable)}$$

$$v \text{ matches } \_ \triangleright \emptyset \qquad \text{(Any)}$$

$$\frac{\forall 1 \leq i \leq n, v_i \text{ matches } pat_i \triangleright env_i}{(v_1, \ldots, v_n) \text{ matches } (pat_1, \ldots, pat_n) \triangleright \bigoplus_{i=1}^{n} env_i} \qquad \text{(Tuple)}$$

$$\frac{v_1 \text{ matches } pat_1 \triangleright env_1 \qquad v_2 \text{ matches } pat_2 \triangleright env_2}{v_1 :: v_2 \text{ matches } pat_1 :: pat_2 \triangleright env_1 \oplus env_2} \qquad \text{(List)}$$

Figure 7: Matches predicate

3.2.2 Primitives

The semantics of primitives is defined in Figure 8. This semantics is defined by rules of the form $prim\ v_1\ \ldots\ v_n \xrightarrow{L}_p e$ where *prim* is an n-ary primitive. Such a rule means that $prim\ v_1\ \ldots\ v_n$ reduces to e with probability p. In contrast to [15], the semantics is probabilistic, because of the presence of the primitive random. The probability p is omitted when it is 1. The label L is used to reflect the operations on locations. It is empty when locations are unaffected. The label ref $v = l$ means that a new location l is created, with contents v. The label $!l = v$ means that the current contents of location l is v. The label $l := v$ means that the contents of the location l is changed into v. The rules are straightforward; they reflect the semantics defined informally in Section 3.1. One is not allowed to test equality between functional values, so we use the predicate funval, also defined in Figure 8, to test whether a value is functional, and raise the exception Invalid_argument when we try to test equality between functional values. There is no rule for the primitive raise: raise v is an exceptional value, it does not reduce.

3.2.3 Expressions and Programs

The semantics of [15] substitutes 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. Because of this change, we also need to add an explicit call stack *stack*. The stack is a list of pairs (env, C), where C is a minimal *evaluation context*, that is, an expression with a hole $[\cdot]$, such that the expression inside the hole can be immediately evaluated. We define a minimal evaluation context as:

$C_m ::=$	minimal expression evaluation context
$e\ [\cdot]$	apply
$[\cdot]\ v$	apply function
let $pat = [\cdot]$ in e	let
$[\cdot]; e$	sequence
if $[\cdot]$ then e_1 else e_2	if

Functional values:

$$\frac{\mathit{prim} \text{ is an } n\text{-ary primitive} \qquad 0 \leq j < n}{\mathsf{funval}(\mathit{prim}\ v_1\ \ldots\ v_j)} \quad \text{(Primitive)}$$

$$\mathsf{funval}(\mathsf{function}[\mathit{env}, \mathit{pm}]) \quad \text{(Function)}$$

$$\mathsf{funval}(\mathsf{letrec}[\mathit{env}, \{x_1 \mapsto \mathsf{function}\ \mathit{pm}_1, \ldots, x_n \mapsto \mathsf{function}\ \mathit{pm}_n\}\ \mathsf{in}\ x_i]) \quad \text{(Let rec)}$$

Primitives:

not

$$\mathsf{not}\ \mathsf{false} \longrightarrow \mathsf{true} \quad \text{(Not1)}$$

$$\mathsf{not}\ \mathsf{true} \longrightarrow \mathsf{false} \quad \text{(Not2)}$$

$(=)$

$$\frac{\mathsf{funval}(v) \text{ or } \mathsf{funval}(v')}{v = v' \longrightarrow \mathsf{raise}\ \mathsf{Invalid\_argument}} \quad \text{(Funval)}$$

$$c = c \longrightarrow \mathsf{true} \quad \text{(Constant1)}$$

$$\frac{c \neq c'}{c = c' \longrightarrow \mathsf{false}} \quad \text{(Constant2)}$$

$$v_1 :: v_2 = v_1' :: v_2' \longrightarrow v_1 = v_1' \mathbin{\&\&} v_2 = v_2' \quad \text{(List1)}$$

$$v_1 :: v_2 = [\,] \longrightarrow \mathsf{false} \quad \text{(List2)}$$

$$[\,] = v_1' :: v_2' \longrightarrow \mathsf{false} \quad \text{(List3)}$$

$$(v_1, \ldots, v_n) = (v_1', \ldots, v_n') \longrightarrow v_1 = v_1' \mathbin{\&\&} \ldots \mathbin{\&\&} v_n = v_n' \quad \text{(Tuples)}$$

ref

$$\mathsf{ref}\ v \xrightarrow{\mathsf{ref}\ v=l} l \quad \text{(New ref)}$$

$(:=)$

$$l := v \xrightarrow{l:=v} () \quad \text{(Assign)}$$

$!$

$$!l \xrightarrow{!l=v} v \quad \text{(Dereference)}$$

random

$$\frac{a \in \{\mathrm{true}, \mathrm{false}\}}{\mathsf{random}\ () \longrightarrow_{1/2} a} \quad \text{(Random)}$$

Figure 8: Rules for OCaml primitives

match $[\cdot]$ with pm	match
try $[\cdot]$ with pm	try
$(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)$	tuple
$e :: [\cdot]$	cons1
$[\cdot] :: v$	cons2
schedule$([\cdot])$	schedule
C_{pm} ::=	minimal program evaluation context
let $pat = [\cdot];;$ *definitions*	let

For instance, $e\ [\cdot]$ and $[\cdot]\ v$ are evaluation contexts, so we evaluate the argument of applications first, and when it becomes a value, we evaluate the function. We denote by $C_m[e]$ the context C_m with the hole $[\cdot]$ replaced by e, and similarly for C_{pm}. The stack contains a minimal program evaluation context C_{pm} in the last element of the list and expression evaluation contexts C_m in the other elements if it is non-empty.

Hence, we evaluate expressions and programs by reducing triples $env, pe, stack$, where pe means program *program* or expression e. The reduction rules $env, pe, stack \xrightarrow{L}_p env', pe', stack'$ are defined in Figures 9 and 10 for expressions and Figure 11 for programs. The label L is defined above in Section 3.2.2. These reductions are probabilistic; the probability p is omitted when it is 1. Most rules are straightforward. In order to evaluate an expression $C_m[e]$, we need to reduce e under the context C_m. To do that, we push the context C_m on the stack with the current environment by rule (Context in), evaluate the expression e until it becomes a value v, and finally pop the context C_m from the stack by rule (Context out), inserting the obtained value v in C_m, yielding $C_m[v]$. In case the expression e raises an exception v, we use rules (Context raise1) and (Context raise2). If the context C_m is not a try context, the result of $C_m[e]$ is also raise v by (Context raise1). If C_m is a try context, we evaluate that try by (Context raise2), followed by (Try2). The rules (Let ctx in), (Let ctx out), and (Let ctx raise) play the same role as (Context in), (Context out), and (Context raise1) respectively, for programs instead of expressions: they allow reducing under the minimal program evaluation context let $pat = [\cdot];;$ *definitions*. There is no rule corresponding to (Context raise2) for programs because there is no try program context.

Example 3 Let us present as an example the reduction of a simple program in an empty environment and an empty stack:

$$\emptyset, \text{let } x = \text{if random () then 0 else 1};;, [\,].$$

We first reduce the expression part of the let, by keeping in the stack the fact that the expression is under the context let $x = [\cdot]$. This expression reduces eventually to a value, and at this point we plug this value back into the context. So we first reduce the previous configuration by (Let ctx in) into:

$$\emptyset, \text{if random () then 0 else 1}, [\emptyset, \text{let } x = [\cdot];;]$$

By (Context in), we prepare to reduce the condition of the if:

$$\emptyset, \mathsf{random}\ (), [\emptyset, \mathsf{if}\ [\cdot]\ \mathsf{then}\ 0\ \mathsf{else}\ 1; \emptyset, \mathsf{let}\ x = [\cdot];;]$$

By (Random), random () reduces to true with probability 1/2 and false with probability 1/2. For the purpose of the example, let us consider the case where random () reduces to true. By (Primitives), the configuration reduces with probability 1/2 into

$$\emptyset, \mathrm{true}, [\emptyset, \mathsf{if}\ [\cdot]\ \mathsf{then}\ 0\ \mathsf{else}\ 1; \emptyset, \mathsf{let}\ x = [\cdot];;]$$

By (Context out), we insert the value of the condition back into the if:

$$\emptyset, \mathsf{if}\ \mathrm{true}\ \mathsf{then}\ 0\ \mathsf{else}\ 1, [\mathsf{let}\ x = [\cdot];;]$$

By (If1), we evaluate the if:

$$\emptyset, 0, [\mathsf{let}\ x = [\cdot];;]$$

By (Let ctx out), we insert the obtained value back into the context let $x = [\cdot];;$

$$\emptyset, \mathsf{let}\ x = 0;;, [\,]$$

By (Variable), we have that 0 matches $x \triangleright \{x \mapsto 0\}$. So, by (Let match1), the configuration reduces into the following last configuration:

$$\{x \mapsto 0\}, \varepsilon, [\,]$$

The expressions addthread(*program*) and schedule(e) are treated specially because they alter parts of the semantic configuration other than $env, pe, stack$. Their treatment is detailed in Section 3.2.5.

3.2.4 Store

As usual, the contents of locations are stored in a *store*, which maps locations to their current values. Figure 12 defines the relation $store \xrightarrow{L} store'$. If a program or an expression reduces by $env, pe, stack \xrightarrow{L}_p env', pe', stack'$, then the store *store* will be updated into *store*′ such that $store \xrightarrow{L} store'$. When L is empty, the store is unchanged by rule (Store empty). When L is $!l = v$, the store is also unchanged, but the reduction succeeds only when the contents of l is v, by rule (Store lookup). When L is $l := v$, the store is updated so that l contains v, by rule (Store assign). When L is ref $v = l$, a new location l is created with contents v, so the contents of l must not be defined in the initial store, by rule (Store alloc).

$$env, x, stack \longrightarrow env, env(x), stack \qquad \text{(Env)}$$

$$\frac{prim\ v_1\ \ldots\ v_n \xrightarrow{L}_p e}{env, prim\ v_1\ \ldots\ v_n, stack \xrightarrow{L}_p env, e, stack} \qquad \text{(Primitives)}$$

$$\frac{\begin{array}{l}e \text{ is not a value} \\ \text{and, when } C_m \text{ is a try context, } e \text{ is not an exceptional value}\end{array}}{env, C_m[e], stack \longrightarrow env, e, (env, C_m) :: stack} \qquad \text{(Context in)}$$

$$env', v, (env, C_m) :: stack \longrightarrow env, C_m[v], stack \qquad \text{(Context out)}$$

$$\frac{C_m \text{ is not a try context}}{env', \mathsf{raise}\ v, (env, C_m) :: stack \longrightarrow env, \mathsf{raise}\ v, stack} \qquad \text{(Context raise1)}$$

$$\frac{C_m \text{ is a try context}}{env', \mathsf{raise}\ v, (env, C_m) :: stack \longrightarrow env, C_m[\mathsf{raise}\ v], stack} \qquad \text{(Context raise2)}$$

$$env, \mathsf{function}\ pm, stack \longrightarrow env, \mathsf{function}[env, pm], stack \qquad \text{(Closure)}$$

$$env, \mathsf{function}[env', pm]\ v_0, stack \longrightarrow env', \mathsf{match}\ v_0\ \mathsf{with}\ pm, stack \qquad \text{(Expr apply)}$$

$$env, v; e, stack \longrightarrow env, e, stack \qquad \text{(Sequence)}$$

$$env, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2, stack \longrightarrow env, e_1, stack \qquad \text{(If1)}$$

$$env, \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2, stack \longrightarrow env, e_2, stack \qquad \text{(If2)}$$

$$\frac{v\ \mathsf{matches}\ pat \triangleright env'}{\begin{array}{l}env, \mathsf{match}\ v\ \mathsf{with}\ pat \to e \mid pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n, stack \longrightarrow \\ env \oplus env', e, stack\end{array}} \qquad \text{(Match1)}$$

$$\frac{\neg(v\ \mathsf{matches}\ pat)}{\begin{array}{l}env, \mathsf{match}\ v\ \mathsf{with}\ pat \to e \mid pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n, stack \longrightarrow \\ env, \mathsf{match}\ v\ \mathsf{with}\ pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n, stack\end{array}} \qquad \text{(Match2)}$$

$$\frac{\neg(v\ \mathsf{matches}\ pat)}{env, \mathsf{match}\ v\ \mathsf{with}\ pat \to e, stack \longrightarrow env, \mathsf{raise\ Match\_failure}, stack} \qquad \text{(Match fail)}$$

$$env, \mathsf{try}\ v\ \mathsf{with}\ pm, stack \longrightarrow env, v, stack \qquad \text{(Try1)}$$

$$env, \mathsf{try\ raise}\ v\ \mathsf{with}\ pm, stack \longrightarrow env, \mathsf{match}\ v\ \mathsf{with}\ pm \mid \_ \to \mathsf{raise}\ v, stack \qquad \text{(Try2)}$$

Figure 9: Rules for expressions

$$\frac{v \text{ matches } pat \triangleright env'}{env, \text{let } pat = v \text{ in } e, stack \longrightarrow env \oplus env', e, stack} \quad \text{(Let1)}$$

$$\frac{\neg(v \text{ matches } pat)}{env, \text{let } pat = v \text{ in } e, stack \longrightarrow env, \text{raise Match\_failure}, stack} \quad \text{(Let2)}$$

$$\frac{letrecenv = \{x_1 \mapsto \text{function } pm_1, \ldots, x_n \mapsto \text{function } pm_n\}}{\begin{array}{l} env, \text{let rec } x_1 = \text{function } pm_1 \text{ and } \ldots \text{ and } x_n = \text{function } pm_n \text{ in } e, stack \\ \quad \longrightarrow env[x_1 \mapsto \text{letrec}[env, letrecenv \text{ in } x_1], \\ \qquad \ldots, \\ \qquad x_n \mapsto \text{letrec}[env, letrecenv \text{ in } x_n]], e, stack \end{array}}$$

(Closure let rec)

$$\frac{letrecenv = \{x_1 \mapsto \text{function } pm_1, \ldots, x_n \mapsto \text{function } pm_n\}}{\begin{array}{l} env, \text{letrec}[env', letrecenv \text{ in } x_i] \; v_0, stack \longrightarrow \\ \quad env'[x_1 \mapsto \text{letrec}[env', letrecenv \text{ in } x_1], \\ \qquad \ldots, \\ \qquad x_n \mapsto \text{letrec}[env', letrecenv \text{ in } x_n]], \\ \quad \text{match } v_0 \text{ with } pm_i, stack \end{array}}$$

(Expr letrec apply)

Figure 10: Rules for expressions (continued)

$$\frac{e \text{ is not a value}}{env, \text{let } pat = e;; definitions, [\,] \longrightarrow env, e, [env, \text{let } pat = [\cdot];; definitions]} \quad \text{(Let ctx in)}$$

$$env', v, [env, \text{let } pat = [\cdot];; definitions] \longrightarrow env, \text{let } pat = v;; definitions, [\,] \quad \text{(Let ctx out)}$$

$$env', \text{raise } v, [env, \text{let } pat = [\cdot];; definitions] \longrightarrow env, \text{raise } v, [\,] \quad \text{(Let ctx raise)}$$

$$\frac{v \text{ matches } pat \triangleright env'}{env, \text{let } pat = v;; definitions, [\,] \longrightarrow env \oplus env', definitions, [\,]} \quad \text{(Let match1)}$$

$$\frac{\neg(v \text{ matches } pat)}{env, \text{let } pat = v;; definitions, [\,] \longrightarrow env, \text{raise Match\_failure}, [\,]} \quad \text{(Let match2)}$$

$$\frac{letrecenv = \{x_1 \mapsto \text{function } pm_1, \ldots, x_n \mapsto \text{function } pm_n\}}{\begin{array}{l} env, \text{let rec } x_1 = \text{function } pm_1 \text{ and } \ldots \text{ and } x_n = \text{function } pm_n;; definitions, [\,] \\ \quad \longrightarrow env[x_1 \mapsto \text{letrec}[env, letrecenv \text{ in } x_1], \\ \qquad \ldots, \\ \qquad x_n \mapsto \text{letrec}[env, letrecenv \text{ in } x_n]], definitions, [\,] \end{array}}$$

(Closure let rec)

Figure 11: Rules for programs

$$store \longrightarrow store \qquad \text{(Store empty)}$$

$$\frac{store(l) = v}{store \xrightarrow{!l=v} store} \qquad \text{(Store lookup)}$$

$$\frac{l \in \mathrm{Dom}(store)}{store \xrightarrow{l:=v} store[l \mapsto v]} \qquad \text{(Store assign)}$$

$$\frac{l \notin \mathrm{Dom}(store)}{store \xrightarrow{\mathsf{ref}\ v=l} store[l \mapsto v]} \qquad \text{(Store alloc)}$$

Figure 12: Store rules

3.2.5 Toplevel Reduction

As mentioned in Section 3.1, and in contrast to [15], we consider several threads running in parallel. Each thread has a configuration $Th_i = \langle env_i, pe_i, stack_i, store_i \rangle$ that contains the current $env_i, pe_i, stack_i$ as explained in Section 3.2.3, as well as the contents of locations local to this thread, in a store $store_i$, as explained in Section 3.2.4. The complete semantic configuration is then

$$\mathcal{C} = [Th_1, \ldots, Th_n], globalstore, tj$$

where tj 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 global locations, and to store the files containing private data from the CryptoVerif process (free variables of roles and tables). In practice, these files may be copied from one machine to another by the user, so they are actually shared between several threads. The values in the global store contain no closure and no reference. (In OCaml, closures and references can be written to a file only by marshalling, but marshalling is ruled out by Assumption A4, since it may bypass the type system.) The global store contains locations in a set S_g, while the local stores contain locations in an infinite set S_l, with $S_g \cap S_l = \emptyset$.

The reduction rules for semantic configurations $\mathcal{C}$ are defined in Figure 13. Actually, this figure defines three relations. The relation $Th \rightarrow_p Th'$, defined by rule (Thread), handles all operations that deal with the current thread only. It updates the store using the same label L as the one used for evaluating the program or the expression, and it checks that this label concerns the local store of the thread. (The location l, if any, must be in S_l.)

Second, the relation $Th, globalstore \rightarrow_p Th', globalstore'$, defined by rules (Globalstore1) and (Globalstore2), handles all operations local to one thread and the global store operations. By rule (Globalstore1), it uses the relation $Th \rightarrow_p Th'$ to handle the operations local to one thread. By rule (Globalstore2), it handles the global store operations. It updates the global store using the

same label L as the one used for evaluating the program or the expression, and it checks that this label concerns the global store. The location l must be in S_g, and the creation of a location in the global store is forbidden. (Otherwise, one would need a way to tell the system whether a new location should be created in the local or in the global store, and to communicate the global locations to the other threads.) We assume that all locations of the global store are initialized at the beginning of the program.

Finally, the relation $\mathcal{C} \rightarrow_p \mathcal{C}'$, defined by the last four rules of Figure 13, gives the semantics of the full language. Rule (Toplevel) runs the current thread tj, using the relation $Th, globalstore \rightarrow_p Th', globalstore'$. Rule (Toplevel add thread) defines the semantics of addthread($program$): it creates a new thread that runs the program $program$, with empty environment, stack, and store. Rules (Toplevel schedule1) and (Toplevel schedule2) define the semantics of schedule: schedule(tj') schedules thread number tj' when this thread exists, and otherwise it raises the exception Invalid_argument.

Splitting the definition of the semantics into three relations allows us to lighten notations in proofs: we can use the reduction on a thread, or on a thread and the global store, without mentioning the other components when they are not affected.

The construct 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 property corresponds to Assumption A6: if several 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.

3.2.6 Modules

OCaml programs typically contain several modules. We adopt a very simple model of modules. A module named μ consists of an OCaml program $program(\mu)$ and its interface $interface(\mu)$ that is the set of OCaml identifiers defined in μ and usable in other modules. When needed to distinguish identifiers coming from different modules, we use identifiers of the form $\mu.x$ for variables defined in module μ. A correct OCaml program is then of the form $program = program(\mu_1);; \ldots ;; program(\mu_n);;$, where, for all $i \leq n$, the free variables of μ_i are defined in the interfaces of μ_j with $j < i$.

Such a program is run by using the previous reduction rules from the initial configuration

$$\mathcal{C}_0(program) = [\langle \emptyset, program, [\,], \emptyset \rangle], globalstore_0, 1$$

where $globalstore_0 = \{l \mapsto initval_l \mid l \in S_g\}$ is the initial value of the global store, and $initval_l$ is the default value for location l: the empty list [ ] for lists,

$$\frac{\begin{array}{l} store \xrightarrow{L} store' \\ L \text{ is empty or } L \text{ is } !l = v,\ l := v, \text{ or } \mathsf{ref}\ v = l \text{ with } l \in S_l \\ env, pe, stack \xrightarrow{L}_p env', pe', stack' \end{array}}{\langle env, pe, stack, store\rangle \longrightarrow_p \langle env', pe', stack', store'\rangle} \quad \text{(Thread)}$$

$$\frac{Th \longrightarrow_p Th'}{Th, globalstore \longrightarrow_p Th', globalstore} \quad \text{(Globalstore1)}$$

$$\frac{\begin{array}{l} globalstore \xrightarrow{L} globalstore' \\ L \text{ is } !l = v \text{ or } l := v \text{ with } l \in S_g \\ env, pe, stack \xrightarrow{L}_p env', pe', stack' \end{array}}{\langle env, pe, stack, store\rangle, globalstore \longrightarrow_p \langle env', pe', stack', store\rangle, globalstore'} \quad \text{(Globalstore2)}$$

$$\frac{Th, globalstore \longrightarrow_p Th', globalstore'}{\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, Th, Th_{tj+1}, \ldots, Th_n], globalstore, tj \longrightarrow_p \\ [Th_1, \ldots, Th_{tj-1}, Th', Th_{tj+1}, \ldots, Th_n], globalstore', tj\end{array}} \quad \text{(Toplevel)}$$

$$\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{addthread}(program), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n, \langle \emptyset, program, [\,], \emptyset\rangle], \\ \quad globalstore, tj\end{array} \quad \text{(Toplevel add thread)}$$

$$\frac{1 \leq tj' \leq n}{\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{schedule}(tj'), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], globalstore, tj'\end{array}} \quad \text{(Toplevel schedule1)}$$

$$\frac{tj' < 1 \text{ or } tj' > n}{\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{schedule}(tj'), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{raise\ Invalid\_argument}, stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj\end{array}} \quad \text{(Toplevel schedule2)}$$

Figure 13: Top level rules

the empty string "" for strings, 0 for integers, false for booleans. (Each location implicitly has a type. Values in the global store cannot contain locations and closures, so we do not define a default value for them.) The program *program* does not contain closures nor locations in S_l, but may contain locations in S_g. (Closures are created by function and letrec; locations in S_l are created by ref.)

Although we ignore types is our syntax, we suppose that our OCaml programs are well-typed, which is checked by the OCaml compiler, and we use the guarantee that well-typed programs do not go wrong: a program stops only when the current thread has been reduced into the empty definition list or an exception raise v (with the empty stack).

3.2.7 Equivalence Modulo Renaming of Locations

The rule (Store alloc) is non-deterministic, since the new location l can be any unused location in S_l. To remove this non-determinism, we consider equivalence classes of OCaml semantic configurations modulo renaming of locations in S_l. We still denote these equivalence classes as OCaml configurations $\mathcal{C}$, and denote an equivalence class by one of its members. On these equivalence classes, the semantics is purely probabilistic. (There is no non-deterministic choice.) If a configuration $\mathcal{C}$ can reduce, then the sum of the probabilities of all possible reductions is 1:

$$\sum_{\{\mathcal{C}' | \mathcal{C} \to_{p(\mathcal{C}')} \mathcal{C}'\}} p(\mathcal{C}') = 1$$

Moreover, for each reduction $\mathcal{C} \to_p \mathcal{C}'$, we have $p > 0$.

We will also use notations similar to Definition 2.6 for the OCaml semantics. We denote by $\mathcal{CT}$ an OCaml trace, $\mathcal{CTS}$ a set of OCaml traces, and we also use the notation $\to^*$ for reductions with several steps.

4 Instrumentation of the OCaml Semantics

In order to prove the correctness of our compiler, we instrument OCaml code in three ways; this section details this instrumentation and proves that it does not alter the semantics of OCaml.

First, we add a new kind of functions and closures tagfunction that behave exactly in the same way as regular functions and closures, but are labeled with additional tags. We use these tagged functions to differentiate functions coming from our generated code and functions coming from the adversary. Hence, we add two new expressions tagfunction^t pm for tagged functions and tagfunction$^{t,\tau}[env, pm]$ for the corresponding closures. We also add tagfunction$^{t,\tau}[env, pm]$ to the values. The tag t indicates the origin of the function or closure; it will be an oracle name or a role name, indicating that the function implements this oracle or role. The tag τ is a fresh tag generated when the function is reduced into a closure: each new closure gets a different tag, so that two closures are the same if and only if they have the same tag. This property will be used in Section 6 to count the number of calls to the same

$$\mathsf{funval}(\mathsf{tagfunction}^{t,\tau}[env, pm]) \qquad \text{(Tagged funval)}$$

$$\frac{\tau \text{ fresh}}{env, \mathsf{tagfunction}^{t}\ pm, stack \longrightarrow env, \mathsf{tagfunction}^{t,\tau}[env, pm], stack} \qquad \text{(Tagged closure)}$$

$$env, \mathsf{tagfunction}^{t,\tau}[env', pm]\ v_0, stack \longrightarrow env', \mathsf{match}\ v_0\ \mathsf{with}\ pm, stack \qquad \text{(Tagged expr apply)}$$

Figure 14: Semantics of tagged functions

closure. The semantic rules for tagged functions are given in Figure 14. They are the same as those for ordinary functions, except for the addition of tags. Much like for locations, we consider traces modulo renaming of tags τ, so that the choice of a fresh tag τ in (Tagged closure) does not lead to non-determinism. The condition that τ is fresh in this rule means that τ is distinct from all tags previously used in the considered trace.

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 expression $\mathsf{event}\ ev(e_1, \ldots, e_k)$ that adds the event $ev(v_1, \ldots, v_k)$ to *events* when $e_1, \ldots, e_k$ evaluate to the values $v_1, \ldots, v_k$ respectively. We consider a new minimal expression evaluation context $\mathsf{event}(ev(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n))$, for evaluating the arguments of events. 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 pairs (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. Hence, the instrumented semantic configuration is

$$\mathcal{CI} = [Th_1, \ldots, Th_n], globalstore, j, \mathcal{MI}, events$$

We adapt the toplevel semantic rules to this configuration as shown in Figure 15. The instrumented semantic rules (New toplevel), (New toplevel schedule1), and (New toplevel schedule2) are straightforwardly adapted from the corresponding rules in the non-instrumented semantics by adding the components $\mathcal{MI}, events$. The rule (Toplevel event) gives the semantics of event: it adds its argument $e(v_1, \ldots, v_n)$ to the list *events* in the configuration and returns $(v_1, \ldots, v_n)$. The rule (New toplevel add thread) gives the instrumented semantics of $\mathsf{addthread}$: the $\mathsf{addthread}$ construct is modified to reject new programs

$$\frac{Th, globalstore \longrightarrow_p Th', globalstore'}{[Th_1, \ldots, Th_{tj-1}, Th, Th_{tj+1}, \ldots, Th_n], globalstore, tj, \mathcal{MI}, events \longrightarrow_p [Th_1, \ldots, Th_{tj-1}, Th', Th_{tj+1}, \ldots, Th_n], globalstore', tj, \mathcal{MI}, events}$$

(New toplevel)

$$\frac{\begin{array}{l} program = program(\mu_{\mathsf{prim}});; program(\mu_1);; \ldots;; program(\mu_l);; program' \\ program' \text{ does not contain } program(\mu_{\mathsf{prim}}) \text{ nor any } program(\mu) \text{ for } \mu \in \mathcal{M}_g \\ \mathcal{M} = \{\mu_1, \ldots, \mu_l\} \subseteq \mathcal{M}_g \\ \forall \mu \in \mathcal{M}, \exists b, (\mu, b) \in \mathcal{MI} \\ \mathcal{MI}' = \{(\mu, \text{false}) \mid \mu \in \mathcal{M} \wedge (\mu, \text{false}) \in \mathcal{MI}\} \\ \text{or} \\ program \text{ does not contain } program(\mu_{\mathsf{prim}}) \text{ nor any } program(\mu) \text{ for } \mu \in \mathcal{M}_g \\ \mathcal{M} = \emptyset, \mathcal{MI}' = \emptyset \end{array}}{\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{addthread}(program), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n, \langle \emptyset, program, [\,], \emptyset\rangle], \\ \quad globalstore, tj, \mathcal{MI} \setminus \mathcal{MI}', events \end{array}}$$

(New toplevel add thread)

$$\frac{1 \leq tj' \leq n}{\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{schedule}(tj'), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj', \mathcal{MI}, events \end{array}}$$

(New toplevel schedule1)

$$\frac{tj' < 1 \text{ or } tj' > n}{\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{schedule}(tj'), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{raise\ Invalid\_argument}, stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \end{array}}$$

(New toplevel schedule2)

$$\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{return}(\mathcal{MI}', v), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, v, stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI} \cup \mathcal{MI}', events \end{array}$$

(Toplevel return)

$$\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{event}\ ev(v_1, \ldots, v_n), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (v_1, \ldots, v_n), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, e(v_1, \ldots, v_n) :: events \end{array}$$

(Toplevel event)

Figure 15: Updated toplevel rules for the instrumented semantics

that contain a module that cannot be called. We let $\mathcal{M}_g$ be the set of generated modules. The programs spawned by addthread can be of two forms. Either they are *attacker programs* that contain neither the module corresponding to the primitives μ_{prim} nor any generated module in $\mathcal{M}_g$, or they are *protocol programs* that first contain the module corresponding to the primitives μ_{prim}, then the necessary generated modules $\mu_1, \ldots, \mu_l$ in $\mathcal{M}_g$, and finally any non-generated program *program*$'$. (We require this order on the modules for simplicity.) The generated modules $\mu_1, \ldots, \mu_l$ must be callable according to the value of $\mathcal{MI}$. The modules $\mu_1, \ldots, \mu_l$ that can be called only once are removed from the callable modules by removing the multiset $\mathcal{MI}'$ from $\mathcal{MI}$.

We also add the expression $\mathsf{return}(\mathcal{MI}', e)$ that adds to the multiset $\mathcal{MI}$ the generated modules present in $\mathcal{MI}'$, and returns the result of e, as defined by rule (Toplevel return). This expression is useful to add modules newly defined at the return from an oracle. We also add the minimal expression evaluation context $\mathsf{return}(\mathcal{MI}, [\cdot])$ to be able to evaluate the second argument of return.

Let us now show 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. This assumption is formalized as follows:

Assumption 4.1 (Only allowed roles) *The instrumented* addthread *rule* (New toplevel add thread) *never fails.*

We first show that, when a program or expression is a value v or an exceptional value $\mathsf{raise}\ v$, the environment does not matter. To prove this property, we define the following equivalence.

Definition 4.2 *We define the equivalence* $\approx_{v\,Th}$ *on threads by*

$$\langle env, pe, stack, store\rangle \approx_{v\,Th} \langle env', pe', stack', store'\rangle$$

if and only if $pe, stack, store = pe', stack', store'$*, and if* pe *is not a value* v *or an exceptional value* $\mathsf{raise}\ v$*, then* $env = env'$*.*

We extend this equivalence to non instrumented configurations $\mathcal{C}$ *and* $\mathcal{C}'$ *by* $\mathcal{C} \approx_v \mathcal{C}'$ *if and only if*

- $\mathcal{C} = [Th_1, \ldots, Th_n], globalstore, tj$,
- $\mathcal{C}' = [Th'_1, \ldots, Th'_n], globalstore, tj$,
- $\forall tj' \leq n, Th_{tj'} \approx_{v\,Th} Th'_{tj'}$.

We show that configurations equivalent by $\approx_v$ reduce in the same way.

Lemma 4.3 *If* $\mathcal{C} \approx_v \mathcal{C}'$ *and* $\mathcal{C} \rightarrow_p \mathcal{C}''$*, then* $\mathcal{C}' \rightarrow_p \mathcal{C}'''$ *and* $\mathcal{C}'' \approx_v \mathcal{C}'''$*.*

Proof No semantic rule uses the environment when the program or expression is a value or an exceptional value.

Indeed, the only semantic rules that apply when the program or expression is a value or an exceptional value are (Context out), (Context raise1), (Context raise2), (Let ctx out), and (Let ctx raise). All these rules replace the current environment with the one stored at the top of the stack. □

By reviewing the changes to the semantics, we can see that the total probability of all reductions is still 1 for the instrumented semantics: If an instrumented semantic configuration $\mathcal{CI}$ can reduce, then

$$\sum_{\{\mathcal{CI}' | \mathcal{CI} \rightarrow_{p(\mathcal{CI}')} \mathcal{CI}'\}} p(\mathcal{CI}') = 1\,.$$

Moreover, for each reduction $\mathcal{CI} \rightarrow_p \mathcal{CI}'$, we have $p > 0$.

Let us now define the function $\mathsf{noinstr}_{\mathcal{CI}}$ that takes a configuration in the instrumented semantics and returns the corresponding configuration in the non-instrumented semantics.

Definition 4.4 *The function* $\mathsf{noinstr}_{Th1}$ *applied to a thread replaces*

1. *every* $\mathsf{return}(\mathcal{MI}, e)$ *with* e,
2. *every* event $ev(e_1, \ldots, e_n)$ *with* $(e_1, \ldots, e_n)$,
3. *and all* tagfunction *functions and closures with regular ones*

in this thread.

The function $\mathsf{noinstr}_{Th2}$ *modifies the stack of the thread by*

- *removing any pair of the form* $(env, \mathsf{return}(\mathcal{MI}, [\cdot]))$,
- *and transforming each pair of the form* $(env, \mathsf{event}(ev(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)))$ *into the pair* $(env, (e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n))$.

Let $\mathsf{noinstr}_{Th} \stackrel{\text{def}}{=} \mathsf{noinstr}_{Th1} \circ \mathsf{noinstr}_{Th2}$.
Finally, let us define

$$\mathsf{noinstr}_{\mathcal{CI}}([Th_1, \ldots, Th_n], globalstore, tj, \mathcal{MI}, events) \stackrel{\text{def}}{=} [\mathsf{noinstr}_{Th}(Th_1), \ldots, \mathsf{noinstr}_{Th}(Th_n)], globalstore, tj$$

We do not need to replace elements of the global store, as they cannot contain closures: event, return, and tagged functions cannot appear in them.

The next proposition shows that, with Assumption 4.1, there is a weak bisimulation between the non-instrumented semantics and the instrumented semantics, that is, the reductions match in the two semantics, but the number of steps may differ. Indeed, the return and event expressions introduce an additional transition in the instrumented semantics. All other constructs reduce in the same number of steps in both semantics. Hence, the instrumentation does not alter the semantics of the language. This result is proved in Appendix B.

Proposition 4.5 *1. If $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ and $\mathcal{C}_1, \ldots, \mathcal{C}_n$ are pairwise distinct configurations such that for all $i \leq n$, we have $\mathcal{C} \to_{p_i} \mathcal{C}_i$ with $\sum_{i \leq n} p_i = 1$, then there exist pairwise distinct instrumented configurations $\mathcal{CI}_1, \ldots, \mathcal{CI}_n$ such that for all $i \leq n$, we have $\mathcal{CI} \to^*_{p_i} \mathcal{CI}_i$ and $\mathcal{C}_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_i)$.*

2. If $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ and $\mathcal{CI}_1, \ldots, \mathcal{CI}_n$ are pairwise distinct instrumented configurations such that for all $i \leq n$, we have $\mathcal{CI} \to_{p_i} \mathcal{CI}_i$ with $\sum_{i \leq n} p_i = 1$, then there exist pairwise distinct configurations $\mathcal{C}_1, \ldots, \mathcal{C}_n$ such that for all $i \leq n$, we have $\mathcal{C} \to^*_{p_i} \mathcal{C}_i$ and $\mathcal{C}_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_i)$.

In the rest of the paper, we use only the instrumented semantics. Furthermore, we denote instrumented configurations by $\mathcal{C}$ to lighten notations.

5 Translation

In this section, we describe how our compiler translates an annotated CryptoVerif process. It 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 that takes a CryptoVerif variable name and returns an OCaml variable name.

The function $\mathbb{G}_{\mathsf{M}}$ transforms a CryptoVerif term M into an OCaml term. It is defined as follows:

$$\mathbb{G}_{\mathsf{M}}(x[\widetilde{i}]) \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{var}}(x) \qquad \text{(Variable)}$$

$$\mathbb{G}_{\mathsf{M}}(f(M_1, \ldots, M_m)) \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{M}}(M_1)), \ldots, (\mathbb{G}_{\mathsf{M}}(M_m)) \qquad \text{(Function call)}$$

Before defining the translation of an oracle, let us first introduce some notations. For each CryptoVerif variable x, we denote by T_x the type of x, and by extension, for each CryptoVerif term M, we denote by T_M the type of M. More precisely, if M is the variable x, then $T_M \stackrel{\text{def}}{=} T_x$, and if M is a function application with a function of type $T_1 \times \cdots \times T_n \to T$, then $T_M \stackrel{\text{def}}{=} T$.

We say that an oracle or role definition occurs *at the beginning of* Q when it occurs in Q just under replication or parallel composition. We define the function reduce$'$ that returns a description of the oracles made available by an oracle definition Q. In more detail, reduce$'(Q)$ is a list $[(Q_1, b_1), \ldots, (Q_l, b_l)]$ such that $Q_1, \ldots, Q_l$ are the oracle definitions at the beginning of Q, from left to right, and the boolean b_l is true when Q_l is under replication, and false otherwise. In this function, the replication indices $\widetilde{i}$ can be partially instantiated into integer values. In contrast to the function reduce, reduce$'(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q)$ does not instantiate the replication index i.

$$\text{reduce}'(0) \stackrel{\text{def}}{=} [\,] \qquad \text{(Nil)}$$

$$\text{reduce}'(Q_1 \mid Q_2) \stackrel{\text{def}}{=} \text{reduce}'(Q_1) \mathbin{@} \text{reduce}'(Q_2) \qquad \text{(Par)}$$

$$\mathrm{reduce}'(\mathsf{foreach}\ i \le n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} [(Q_1, \mathrm{true}), \ldots, (Q_l, \mathrm{true})] \quad \text{when } \mathrm{reduce}'(Q) = [(Q_1, b_1), \ldots, (Q_l, b_l)] \text{ for some } b_1, \ldots, b_l \tag{Repl}$$

$$\mathrm{reduce}'(O[\widetilde{i}](x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]) := P) \stackrel{\text{def}}{=} [(O[\widetilde{i}](x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]) := P, \mathrm{false})] \tag{Oracle}$$

$$\mathrm{reduce}'(\mathsf{role}\,\{Q) \stackrel{\text{def}}{=} [\,] \tag{Role}$$

The function reduce′ takes processes Q that follow return statements that do not end a role. By Assumption 2.9, we are inside a role, so by Property 2.8, the construct role $\{Q'$ cannot appear in Q before a return statement that ends the current oracle. So, the function reduce′ will never be called on role $\{Q'$.

We also define the function $\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}$ that returns a description of the modules that correspond to roles defined at the beginning of an oracle definition Q. The function $\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}$ is similar to the function reduce′ above: it returns pairs containing the module generated for the role and a boolean indicating whether the role is under replication or not. In contrast to reduce′, it returns a set and not a list.

$$\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(0) \stackrel{\text{def}}{=} \emptyset \tag{Nil}$$

$$\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_1 \mid Q_2) \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_1) \cup \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_2) \tag{Par}$$

$$\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(\mathsf{foreach}\ i \le n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} \{(\mu, \mathrm{true}) \mid \exists b, (\mu, b) \in \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q)\} \tag{Repl}$$

$$\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(O[\widetilde{i}](x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]) := P) \stackrel{\text{def}}{=} \emptyset \tag{Oracle}$$

$$\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(\mathsf{role}\,\{Q) \stackrel{\text{def}}{=} \{(\mu_{\mathsf{role}}, \mathrm{false})\} \tag{Role}$$

The function $\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}$ takes processes Q that follow return statements that end the current role. By Assumption 2.9, there cannot be an oracle definition before a role $\{Q'$ in Q. So the function $\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}$ will never be called on oracle definitions.

To translate an oracle, we translate the body of the oracle using the function $\mathbb{G}$ defined in Figure 16. Most cases are straightforward. After defining a variable, we store it in a file if needed, using $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}])$, defined by $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]) \stackrel{\text{def}}{=} (f := \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x))$ if $(x[\widetilde{i}], f) \in \mathsf{files}$ and $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]) \stackrel{\text{def}}{=} ()$ otherwise. A file is modeled by a global store location.

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, as defined in (Return1). Otherwise, we update the set of available roles using the primitive return introduced in Section 4, as defined in (Return2).

In the insert case, we add the inserted element to the considered table Tbl contained in the global store location f. In the get case, we read the table by read_table, keeping only the elements that satisfy the required condition (which is tested by $\mathbb{G}_{\mathsf{test}}$). These elements are stored in the list l. If l is empty, we run P'; otherwise, we choose a random element in l and run P with that element. To choose that element, we use a function random_l such that $\mathsf{random}_l\ l$ returns a random element of the list l, such that the probability of returning

$$\mathbb{G}(x[\widetilde{i}] \overset{R}{\leftarrow} T; P) \overset{\text{def}}{=} \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{random}}(T)\ ()\ \mathsf{in}\ \mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]); \mathbb{G}(P) \qquad \text{(New)}$$

$$\mathbb{G}(x[\widetilde{i}] \leftarrow M; P) \overset{\text{def}}{=} \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{in}\ \mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]); \mathbb{G}(P) \qquad \text{(Let)}$$

$$\mathbb{G}(\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P') \overset{\text{def}}{=} \mathsf{if}\ \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{then}\ \mathbb{G}(P)\ \mathsf{else}\ \mathbb{G}(P') \qquad \text{(If)}$$

$$\frac{[(Q_1, b_1), \ldots, (Q_l, b_l)] \overset{\text{def}}{=} \text{reduce}'(Q)}{\mathbb{G}(\mathsf{return}(N_1, \ldots, N_k); Q) \overset{\text{def}}{=} (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_l, b_l), \mathbb{G}_{\mathsf{M}}(N_1), \ldots, \mathbb{G}_{\mathsf{M}}(N_k))} \qquad \text{(Return1)}$$

$$\mathbb{G}(\mathsf{return}(N_1, \ldots, N_k)\}; Q) \overset{\text{def}}{=} (\mathsf{return}(\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q), (\mathbb{G}_{\mathsf{M}}(N_1), \ldots, \mathbb{G}_{\mathsf{M}}(N_k)))) \qquad \text{(Return2)}$$

$$\mathbb{G}(\mathsf{end}) \overset{\text{def}}{=} (\mathsf{raise\ Match\_failure}) \qquad \text{(End)}$$

$$\mathbb{G}(\mathsf{event}\ ev(M_1, \ldots, M_k); P) \overset{\text{def}}{=} \mathsf{event}\ ev(\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_k)); \mathbb{G}(P) \qquad \text{(Event)}$$

$$\frac{(Tbl, f) \in \mathsf{tables}}{\begin{array}{l}\mathbb{G}(\mathsf{insert}\ Tbl(M_1, \ldots, M_k); P) \overset{\text{def}}{=} \\ \quad (f := (\mathbb{G}_{\mathsf{ser}}(T_{M_1})\ \mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{ser}}(T_{M_k})\ \mathbb{G}_{\mathsf{M}}(M_k)) :: (!f); \mathbb{G}(P))\end{array}} \qquad \text{(Insert)}$$

$$\begin{array}{l}\mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M) \overset{\text{def}}{=} \\ \quad (\mathsf{function}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) \rightarrow \\ \qquad \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x_1) = \mathbb{G}_{\mathsf{deser}}(T_{x_1})\ \mathbb{G}_{\mathsf{var}}(x_1)\ \mathsf{in} \ldots \\ \qquad \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x_k) = \mathbb{G}_{\mathsf{deser}}(T_{x_k})\ \mathbb{G}_{\mathsf{var}}(x_k)\ \mathsf{in} \\ \qquad \mathsf{if}\ (\mathbb{G}_{\mathsf{M}}(M))\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) \\ \qquad\qquad\qquad \mathsf{else\ raise\ Match\_failure} \\ \quad \mid \_ \rightarrow \mathsf{raise\ Bad\_file})\end{array} \qquad \text{(Test)}$$

$$\mathbb{G}_{\mathsf{fold}} \overset{\text{def}}{=} f \rightarrow \mathsf{function}\ a \rightarrow \mathsf{function}\ [\,] \rightarrow a \mid x :: l \rightarrow f\ (\mathit{fold}\ f\ a\ l)\ x \qquad \text{(Fold)}$$

$$\begin{array}{l}\text{read\_table}(f, c) \overset{\text{def}}{=} \mathsf{let\ rec}\ \mathit{fold} = \mathsf{function}\ \mathbb{G}_{\mathsf{fold}}\ \mathsf{in} \\ \quad \mathit{fold}\ (\mathsf{function}\ a \rightarrow \mathsf{function}\ x \rightarrow \\ \qquad (\mathsf{try}\ (c\ x) :: a\ \mathsf{with} \\ \qquad\quad \mathsf{Match\_failure} \rightarrow a))\ [\,]\ !f\end{array} \qquad \text{(Read table)}$$

$$\frac{(Tbl, f) \in \mathsf{tables}}{\begin{array}{l}\mathbb{G}(\mathsf{get}\ Tbl(x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P') \overset{\text{def}}{=} \\ \quad \mathsf{let}\ l = \text{read\_table}(f, \mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M))\ \mathsf{in} \\ \quad \mathsf{if}\ l = [\,]\ \mathsf{then}\ \mathbb{G}(P') \\ \qquad\quad \mathsf{else\ let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = \mathsf{random}_l\ l\ \mathsf{in} \\ \qquad\quad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P))\end{array}} \qquad \text{(Get)}$$

Figure 16: Translation function $\mathbb{G}$ of an oracle body in OCaml

$\mathbb{G}_{\mathsf{O}}(Q, \text{false}) \stackrel{\text{def}}{=} \mathsf{let}\ token = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}^{O}\ pm_{\text{false}}(Q)$
where $pm_{\text{false}}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\text{def}}{=}$
$(\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) \rightarrow$
$\mathsf{if}\ (!token)\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \ldots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k))$
$\mathsf{then}\ (token := \text{false}; \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P))$
$\mathsf{else\ raise\ Bad\_Call}$

(Oracle1)

$\mathbb{G}_{\mathsf{O}}(Q, \text{true}) \stackrel{\text{def}}{=} \mathsf{tagfunction}^{O}\ pm_{\text{true}}(Q)$
where $pm_{\text{true}}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\text{def}}{=}$
$(\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) \rightarrow$
$\mathsf{if}\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \ldots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k))$
$\mathsf{then}\ (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P))$
$\mathsf{else\ raise\ Bad\_Call}$

(Oracle2)

Figure 17: Translation of an oracle

the j-th element of l is among$(\{1, \ldots, |l|\}, j)$. We suppose that this function is programmed from the OCaml primitive random, and is present in the module for cryptographic primitives μ_{prim}.

An oracle $O(x_1, \ldots, x_n) := P$ is transformed into a closure by the function $\mathbb{G}_{\mathsf{O}}$ as shown in Figure 17. When the oracle O is not under replication (second argument of $\mathbb{G}_{\mathsf{O}}$ false, in (Oracle1)), 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, the exception Bad_Call will be raised. The translation of an oracle always checks that the arguments are correct values for their CryptoVerif types, and stores them in files if necessary by calling $\mathbb{G}_{\mathsf{file}}$.

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 $Q(\mathsf{role})$, so its interface is $\mathit{interface}(\mu_{\mathsf{role}}) \stackrel{\text{def}}{=} \{\mu_{\mathsf{role}}.\mathsf{init}\}$ and its program is

$\mathit{program}(\mu_{\mathsf{role}}) \stackrel{\text{def}}{=} \mathsf{let}\ \mu_{\mathsf{role}}.\mathsf{init} = \mathsf{let}\ token = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}}\ pm_{\mathsf{role}}$
where $pm_{\mathsf{role}} \stackrel{\text{def}}{=} () \rightarrow$
$\mathsf{if}\ (!token)\ \mathsf{then}$
$(token := \text{false};$
$\mathbb{G}_{\mathsf{read}}(x_1[\,])\ \mathsf{in}\ \ldots\ \mathsf{in}\ \mathbb{G}_{\mathsf{read}}(x_m[\,])\ \mathsf{in}$
$(\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_k, b_k)))$
$\mathsf{else\ raise\ Bad\_Call}$

where $[(Q_1, b_1), \ldots, (Q_k, b_k)] = \text{reduce}'(Q(\mathsf{role}))$ and $x_1[\,], \ldots, x_m[\,]$ are the free variables of $Q(\mathsf{role})$, which are the variables we need to retrieve from the files. The function $\mathbb{G}_{\mathsf{read}}(x[\,])$, which reads the contents of the file associated to $x[\,]$, is defined by $\mathbb{G}_{\mathsf{read}}(x[\,]) \stackrel{\text{def}}{=} \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{deser}}(T_x)\ !(f)\ \text{if}\ (x[\,], f) \in \mathsf{files}$.

Example 4 Let us explain the translation of the role alice described in Examples 1 and 2. This role contains only the following oracle

$$\mathrm{OA}() := m \stackrel{R}{\leftarrow} nonce; s \stackrel{R}{\leftarrow} seed; \mathsf{return}(m, \mathrm{sign}(m, sk, s))$$

This role is translated into the module μ_{alice}. Its program $program(\mu_{\mathsf{alice}})$ is:

```
let μalice.init = let token = ref true in tagfunction^alice () →
    if (!token) then
        (token := false;
         G_read(sk[]) in
         let token = ref true in tagfunction^OA () →
             if (!token) then
                 (token := false;
                  let G_var(m) = G_random(nonce) () in
                  let G_var(s) = G_random(seed) () in
                  (G_var(m), G_f(sign) (G_var(m), G_var(sk), G_var(s))))
             else raise Bad_Call)
    else raise Bad_Call
```

This program defines the function $\mu_{\mathsf{alice}}.\mathsf{init}$, which expects () as argument and returns the function that implements oracle OA. This function itself expects () as argument and returns the OCaml representation of the message $m, \mathrm{sign}(m, sk, s)$ returned by OA.

The function $\mu_{\mathsf{alice}}.\mathsf{init}$ can be called only once, which is guaranteed using a boolean reference *token*. If *token* is already false, then this function has already been called, so we raise the exception Bad_Call. Otherwise, we set *token* to false, we read the value of *sk* from the file *skfile* and we continue by the translation of the CryptoVerif oracle OA. The oracle OA can also be called only once (but the program $program(\mu_{\mathsf{alice}})$ can be launched N_1 times in different threads). So we define a new boolean reference *token*, to guarantee this property, and define the function that implements OA. When this function is called for the first time, it sets this second *token* to false, creates a new nonce $\mathbb{G}_{\mathsf{var}}(m)$ and a new seed $\mathbb{G}_{\mathsf{var}}(s)$, and finally returns the nonce $\mathbb{G}_{\mathsf{var}}(m)$ with the signature. When this function is called again, it raises the exceptional value Bad_Call.

To call the translation of oracle OA, one can execute:

$$\mu_{\mathsf{alice}}.\mathsf{init}\ ()\ ()$$

This code initializes the role and calls the closure returned by $\mu_{\mathsf{alice}}.\mathsf{init}\ ()$, which corresponds to the translation of OA.

The generated modules $\mathcal{M}_g$ (μ_{role} for each role in the CryptoVerif process) 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. These programs interact with an adversary that we model as an OCaml program $program_0$. We consider that the programs of the protocol are launched by the adversary $program_0$ using the addthread construct. The generated modules depend only on the module containing the cryptographic primitives μ_{prim}, so when the program of a thread uses the primitives or the generated modules, we can order the programs of the modules in the argument of addthread in the order $program(\mu_{\mathsf{prim}});; program(\mu_{\mathsf{role}_1});; \ldots ;; program(\mu_{\mathsf{role}_k});; program'$ where $program'$ contains no generated module, as required by the instrumented semantics of addthread (New toplevel add thread). We assume that $program_0$ uses the generated modules only inside addthread, and that $program_0$ 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.) We assume that only the generated modules use events, tagged functions, and return. The adversary must not use events, which serve for specifying security properties of the protocol, nor return, which serves for updating the set of callable generated modules. He uses regular functions rather than tagged functions. Moreover, as mentioned in Assumption A3, we suppose that only the generated modules access files that contain private CryptoVerif data (free variables of roles and tables). So we let $S_{priv} \stackrel{\text{def}}{=} \{f \mid (x[\,], f) \in \mathsf{files} \text{ or } (Tbl, f) \in \mathsf{tables}\} \subseteq S_g$ be the set of global locations reserved for private CryptoVerif data, and we have the following assumption:

Assumption 5.1 *The locations in S_{priv} occur only in the programs of generated modules; they do not occur elsewhere in $program_0$.*

The program $program_0$ is run in the initial (instrumented) OCaml configuration $\mathcal{C}_0(Q_0, program_0)$ defined as follows:

$$\mathcal{C}_0(Q_0, program_0) \stackrel{\text{def}}{=} [\langle \emptyset, program_0, [\,], \emptyset \rangle], globalstore_0, 1, \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_0), [\,]$$

where $\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_0)$ is the set of modules available at the beginning of the execution and $globalstore_0 \stackrel{\text{def}}{=} \{l \mapsto initval_l \mid l \in S_g\}$ is the initial value of global store, as defined in Section 3.2.6. Tables are represented by lists, and their initial value $initval_l$ is the empty list $[\,]$, representing that the tables are initially empty. Files that contain free variables of roles are represented by strings, and their initial value $initval_l$ is the empty string "". For other elements, the initial value $initval_l$ is the default value for the type of location l.

6 Proof of Security

This section presents the proof of correctness of our compiler. We give ourselves a CryptoVerif process Q_0 that corresponds to a cryptographic protocol. By using our compiler, we first generate modules $\mathcal{M}_g$ that correspond to the

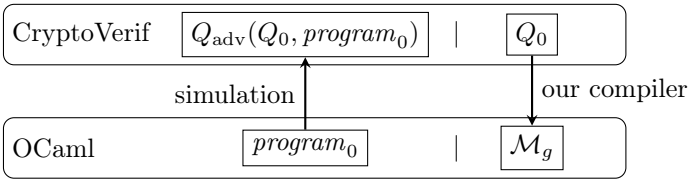

Figure 18: Overview of our proof

roles present inside Q_0, as explained in the previous section. We consider an adversary interacting with the protocol implementation, modeled as an OCaml program $program_0$ that uses the generated modules in $\mathcal{M}_g$. Informally, when CryptoVerif shows that Q_0 satisfies a certain security property, it shows that for any CryptoVerif adversary Q, the probability that $Q \mid Q_0$ breaks the security property is bounded by a certain bound, which CryptoVerif computes. Our goal is to show that the same probability bound also applies to the generated implementation, that is, the probability that $program_0$ breaks the security property is bounded by the same bound. As illustrated in Figure 18, to prove this property, we build from the OCaml adversary $program_0$ a CryptoVerif adversary $Q_{\mathrm{adv}}(Q_0, program_0)$ that simulates $program_0$. Basically, we run the OCaml program $program_0$ inside a CryptoVerif primitive *simulate*; when $program_0$ would call the translation of an oracle in $\mathcal{M}_g$, $Q_{\mathrm{adv}}(Q_0, program_0)$ calls the corresponding oracle in Q_0. We prove that $Q_{\mathrm{adv}}(Q_0, program_0) \mid Q_0$ and $program_0$ using $\mathcal{M}_g$ behave similarly, hence they have the same probability of breaking the security property. To achieve this goal, we need to prove, firstly, that the translations of the oracles behave in the same way as the CryptoVerif oracles, and secondly, that our simulation is sound.

In Section 6.1, we state our assumptions on the cryptographic primitives, and show that the primitives behave correctly independently of the rest of the program. In Section 6.2, we prove that the OCaml translation of a CryptoVerif oracle behaves like the oracle. In Section 6.3, we define the CryptoVerif adversary that simulates the OCaml adversary $program_0$. Finally, in Section 6.4, we prove that the CryptoVerif adversary interacting with Q_0 behaves like the OCaml adversary interacting with the generated implementation. This result shows the desired correctness of our compiler.

6.1 Correctness of Cryptographic Primitives

Let us first formalize the assumptions we make about the implementation of cryptographic primitives. Let $program_{\mathsf{prim}} \stackrel{\mathrm{def}}{=} program(\mu_{\mathsf{prim}})$ be the program of the module that defines the primitives and $interface_{\mathsf{prim}} \stackrel{\mathrm{def}}{=} interface(\mu_{\mathsf{prim}})$ be its interface. The interface $interface_{\mathsf{prim}}$ consists of the function random_l, the functions $\mathbb{G}_{\mathsf{f}}(f)$ for each CryptoVerif function f, and the functions $\mathbb{G}_{\mathsf{random}}(T)$, $\mathbb{G}_{\mathsf{ser}}(T)$, $\mathbb{G}_{\mathsf{deser}}(T)$, and $\mathbb{G}_{\mathsf{pred}}(T)$ for each CryptoVerif type T for which these

functions are used in the translation, as described in Section 2.3. (The functions $\mathbb{G}_{\mathsf{ser}}(T)$ and $\mathbb{G}_{\mathsf{deser}}(T)$ are either both present or both absent in $interface_{\mathsf{prim}}$.) We rely on the following assumptions.

Assumption 6.1 *There are no* schedule, addthread, return, *nor* event *operations and no global store locations in* $program_{\mathsf{prim}}$.

An OCaml semantic configuration in which the current thread does not use addthread, return, event, schedule operations, nor global store locations reduces by using the rule (Thread) of Figure 13, so we can reduce it by considering as configuration only a thread Th. We denote by $\mathcal{TT}$ the traces over threads.

Let $Th_0^s \stackrel{\text{def}}{=} \langle \emptyset, program_{\mathsf{prim}};;, [\,], \emptyset \rangle$ be a thread configuration that evaluates only the implementation of the cryptographic primitives module.

Assumption 6.2 *There exists a unique complete thread trace* $\mathcal{TT}$ *beginning at the configuration* Th_0^s *and there exists* env_{prim} *such that the last configuration of the trace* $\mathcal{TT}$ *is:*

$$Th = \langle env_{\mathsf{prim}}, \varepsilon, [\,], \emptyset \rangle$$

This assumption means that there are no uncatched exceptions, no access to the store, and no random operations in the initialization of the module μ_{prim}, so that the environment env_{prim} is always the same. Typically, the initialization just defines functions, so this assumption is not restrictive. Random choices and a limited access to the store explained below are allowed during calls to primitives. By definition of a module, we have $interface_{\mathsf{prim}} \subseteq \mathrm{Dom}(env_{\mathsf{prim}})$.

Assumption 6.3 *For each CryptoVerif type* T*, OCaml values of the corresponding type* $\mathbb{G}_T(T)$ *does not contain closures nor store or global store locations.*

In particular, bitstrings received or returned by cryptographic primitives are typically represented by strings. Strings are values in our semantics, and do not contain locations. So one cannot alter the contents of a string after creation: string values are not mutable. In OCaml, the type `string` is mutable, so either one should use an immutable abstract type instead of `string` to represent bitstrings, or one needs to assume that the network code does not modify the contents of strings that go through our generated closures, as mentioned in Assumption A5.

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 $\mathbb{G}_{\mathsf{pred}}(T)$. Furthermore, the OCaml value true and the CryptoVerif value true are such that $\mathbb{G}_{\mathsf{val}bool}(\mathrm{true}) = \mathrm{true}$, and the same goes for false. We extend this function to events by $\mathbb{G}_{\mathsf{ev}}(e(a_1, \ldots, a_j)) = e(\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_j}(a_j))$ 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: the implementation of the cryptographic

primitives in $interface_{\mathsf{prim}}$ correctly emulates the corresponding behavior of CryptoVerif.

Assumption 6.4 (Correct primitives) *For each CryptoVerif function f of type $T_1 \times \cdots \times T_n \to T$, for each CryptoVerif values $a_1, \ldots, a_n$ of types $T_1, \ldots, T_n$, there exist env and store such that*

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{f}}(f))\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_n}(a_n)), [\,], \emptyset\rangle \to^* \langle env, \mathbb{G}_{\mathsf{val}T}(f(a_1, \ldots, a_n)), [\,], store\rangle\,.$$

For each CryptoVerif type T such that the function $\mathbb{G}_{\mathsf{random}}(T)$ is in $interface_{\mathsf{prim}}$, for each CryptoVerif value $a \in T$, there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{random}}(T))\ (), [\,], \emptyset\rangle \to^*_{1/|T|} \langle env, \mathbb{G}_{\mathsf{val}T}(a), [\,], store\rangle\,.$$

For each CryptoVerif type T such that the function $\mathbb{G}_{\mathsf{pred}}(T)$ is in $interface_{\mathsf{prim}}$, for each value v of the OCaml type $\mathbb{G}_T(T)$, there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{pred}}(T))\ v, [\,], \emptyset\rangle \to^* \langle env, v', [\,], store\rangle$$

where $v' = \text{true}$ when $\mathbb{G}^{-1}_{\mathsf{val}T}(v)$ exists, and $v' = \text{false}$ otherwise.

For each CryptoVerif type T such that the functions $\mathbb{G}_{\mathsf{ser}}(T)$ and $\mathbb{G}_{\mathsf{deser}}(T)$ are in $interface_{\mathsf{prim}}$, for each CryptoVerif value $a \in T$, there exists an OCaml string value $\mathsf{ser}(T, a)$, such that there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{ser}}(T))\ \mathbb{G}_{\mathsf{val}T}(a), [\,], \emptyset\rangle \to^* \langle env, \mathsf{ser}(T, a), [\,], store\rangle$$

and there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{deser}}(T))\ \mathsf{ser}(T, a), [\,], \emptyset\rangle \to^* \langle env, \mathbb{G}_{\mathsf{val}T}(a), [\,], store\rangle\,.$$

If v is a non-empty list, then for each $a \in v$, there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathsf{random}_l)\ v, [\,], \emptyset\rangle \to^*_{\sum_{j \in S} \text{among}(\{1,\ldots,|v|\},j)} \langle env, a, [\,], store\rangle$$

where $S \stackrel{\text{def}}{=} \{1 \leq j \leq |v| \mid \mathsf{nth}(v, j) = a\}$.

In contrast to the conference version of this paper [9], in this paper, we allow the cryptographic primitives to use the store for their internal computations (which often happens in practice). However, the primitives are not allowed to communicate across calls or to communicate data to the adversary or to the rest of the code using the store. They can create new locations and use these locations internally; these locations become unreachable as soon as the call to the primitive ends. This assumption corresponds to the intuition that the cryptographic primitives are pure functions: their usage of the store should not have any visible side effect. This assumption is modeled above by considering that the primitives are initially called in an empty store. Since their return value does not contain locations, the store at the end of the call will be unreachable.

The last statement of Assumption 6.4 guarantees that random_l is programmed correctly: $\mathsf{random}_l\ v$ returns a random element of the list v, such that the probability of returning the j-th element of v is among$(\{1,\ldots,|v|\},j)$. In case the same element occurs several times in v, the probability of that element is then the sum of the probabilities of all its occurrences.

In general, when primitives make probabilistic choices, they might return the same result in several traces with a different environment and store. To simplify notations, Assumption 6.4 states that this does not happen, so that we have the same environment and store in all final configurations that yield the same result. Our proof could easily be extended to the general case if desired.

The next proposition shows that the primitives always return correct results, when they are called inside an OCaml program, so possibly with a non-empty store and a non-empty stack. We prove it in Appendix C. It is a consequence of Assumption 6.4.

Proposition 6.5 (Correct behavior of the primitives) *Let us consider a thread* $Th \stackrel{\text{def}}{=} \langle env, env_{\mathsf{prim}}(s)\ v, stack, store\rangle$.

- *If* $s = \mathbb{G}_{\mathsf{f}}(f)$, *$f$ is a CryptoVerif function of type* $T_1 \times \cdots \times T_n \to T$, *and* $v = (\mathbb{G}_{\mathsf{val}T_1}(a_1),\ldots,\mathbb{G}_{\mathsf{val}T_n}(a_n))$ *for some CryptoVerif values* $a_1,\ldots,a_n$ *of types* $T_1,\ldots,T_n$, *then there exist* env' *and* $store'$ *such that*
$$Th \to^* \langle env', \mathbb{G}_{\mathsf{val}T}(f(a_1,\ldots,a_n)), stack, store'\rangle\,.$$
- *If* $s = \mathbb{G}_{\mathsf{random}}(T)$ *and* $v = ()$, *then for each CryptoVerif value* $a \in T$, *there exist* env' *and* $store'$ *such that*
$$Th \to^*_{1/|T|} \langle env', \mathbb{G}_{\mathsf{val}T}(a), stack, store'\rangle\,.$$
- *If* $s = \mathbb{G}_{\mathsf{pred}}(T)$, *then there exist* env' *and* $store'$ *such that*
$$Th \to^* \langle env', v', stack, store'\rangle$$
where $v' = \text{true}$ *when* $\mathbb{G}^{-1}_{\mathsf{val}T}(v)$ *exists, and* $v' = \text{false}$ *otherwise.*
- *If* $s = \mathbb{G}_{\mathsf{ser}}(T)$ *and* $v = \mathbb{G}_{\mathsf{val}T}(a)$, *then there exist* env' *and* $store'$ *such that*
$$Th \to^* \langle env', \mathsf{ser}(T,a), stack, store'\rangle\,.$$
- *If* $s = \mathbb{G}_{\mathsf{deser}}(T)$ *and* $v = \mathsf{ser}(T,a)$, *then there exist* env' *and* $store'$ *such that*
$$Th \to^* \langle env', \mathbb{G}_{\mathsf{val}T}(a), stack, store'\rangle\,.$$
- *If* $s = \mathsf{random}_l$ *and* v *is a non-empty list, then for each* $a \in v$, *there exist* env' *and* $store'$ *such that*
$$Th \to^*_{\sum_{j\in S} \text{among}(\{1,\ldots,|v|\},j)} \langle env', a, stack, store'\rangle$$
where $S \stackrel{\text{def}}{=} \{1 \le j \le |v| \mid \mathsf{nth}(v,j) = a\}$.

In all cases, we have $store' \supseteq store$.

6.2 Correctness of the Translation of Oracle Bodies

In this section, we show the correctness of the translation of oracle bodies in our compiler: we show a correspondence between the semantics of the oracle body in CryptoVerif and the semantics of its translation into OCaml.

Let $\mathsf{fv}(M)$, $\mathsf{fv}(P)$, $\mathsf{fv}(Q)$ be the free variables of the CryptoVerif term M and processes P and Q, respectively, defined as follows:

$\mathsf{fv}(x[\tilde{i}]) \stackrel{\text{def}}{=} \{x[\tilde{i}]\}$
$\mathsf{fv}(f(M_1, \ldots, M_k)) \stackrel{\text{def}}{=} \bigcup_{i=1}^{k} \mathsf{fv}(M_i)$
$\mathsf{fv}(0) \stackrel{\text{def}}{=} \emptyset$
$\mathsf{fv}(Q \mid Q') \stackrel{\text{def}}{=} \mathsf{fv}(Q) \cup \mathsf{fv}(Q')$
$\mathsf{fv}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} \mathsf{fv}(Q)$
$\mathsf{fv}(O[\tilde{i}](x_1[\tilde{i}] : T_1, \ldots, x_k[\tilde{i}] : T_k) := P) \stackrel{\text{def}}{=} \mathsf{fv}(P) \setminus \{x_1[\tilde{i}], \ldots, x_k[\tilde{i}]\}$
$\mathsf{fv}(\mathsf{return}(M_1, \ldots, M_k); Q) \stackrel{\text{def}}{=} \bigcup_{i=1}^{k} \mathsf{fv}(M_i) \cup \mathsf{fv}(Q)$
$\mathsf{fv}(\mathsf{end}) \stackrel{\text{def}}{=} \emptyset$
$\mathsf{fv}(x[\tilde{i}] \stackrel{R}{\leftarrow} T; P) \stackrel{\text{def}}{=} \mathsf{fv}(P) \setminus \{x[\tilde{i}]\}$
$\mathsf{fv}(x[\tilde{i}] \leftarrow M; P) \stackrel{\text{def}}{=} \mathsf{fv}(P) \setminus \{x[\tilde{i}]\}$
$\mathsf{fv}(\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=} \mathsf{fv}(M) \cup \mathsf{fv}(P) \cup \mathsf{fv}(P')$
$\mathsf{fv}(\mathsf{insert}\ Tbl(M_1, \ldots, M_k); P) \stackrel{\text{def}}{=} \bigcup_{i=1}^{k} \mathsf{fv}(M_i) \cup \mathsf{fv}(P)$
$\mathsf{fv}(\mathsf{get}\ Tbl(x_1[\tilde{i}], \ldots, x_k[\tilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=}$
$\quad (\mathsf{fv}(M) \cup \mathsf{fv}(P) \setminus \{x_1[\tilde{i}], \ldots, x_k[\tilde{i}]\}) \cup \mathsf{fv}(P')$
$\mathsf{fv}(\mathsf{event}\ e(M_1, \ldots, M_k); P) \stackrel{\text{def}}{=} \bigcup_{i=1}^{k} \mathsf{fv}(M_i) \cup \mathsf{fv}(P)$
$\mathsf{fv}(\mathsf{let}\ (x_1[\tilde{i}] : T_1, \ldots, x_{k'}[\tilde{i}] : T_{k'}) = O[M_1, \ldots, M_l](M'_1, \ldots, M'_k)\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=}$
$\quad \bigcup_{i=1}^{l} \mathsf{fv}(M_i) \cup \bigcup_{i=1}^{k} \mathsf{fv}(M'_i) \cup (\mathsf{fv}(P) \setminus \{x_1[\tilde{i}], \ldots, x_{k'}[\tilde{i}]\}) \cup \mathsf{fv}(P')$
$\mathsf{fv}(\mathsf{let}\ x[\tilde{i}] : T = \mathsf{loop}\ O[M_1, \ldots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=}$
$\quad \bigcup_{i=1}^{l} \mathsf{fv}(M_i) \cup \mathsf{fv}(M') \cup (\mathsf{fv}(P) \setminus \{x[\tilde{i}]\}) \cup \mathsf{fv}(P')$

The only unusual point in this definition is that variables are arrays $x[\tilde{i}]$. We extend this definition to terms and processes in which the replication indices $\tilde{i}$ have been instantiated to bitstrings: for example, $\mathsf{fv}(x[\tilde{a}]) = \{x[\tilde{a}]\}$. We extend this definition to sets of processes by $\mathsf{fv}(\mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}} \mathsf{fv}(Q)$ and to stacks by $\mathsf{fv}(\mathcal{R}) = \bigcup_{((x_1[\tilde{a}], \ldots, x_k[\tilde{a}]), P_1, P_2) \in \mathcal{R}} \mathsf{fv}(P_1) \setminus \{x_1[\tilde{a}], \ldots, x_k[\tilde{a}]\} \cup \mathsf{fv}(P_2)$.

Next, we define the OCaml value corresponding to a CryptoVerif table, and we use this definition to define the OCaml environment and global store corresponding to a CryptoVerif environment and to CryptoVerif tables.

Definition 6.6 (CryptoVerif table to OCaml list) *Let us consider a table Tbl of type $T_1 \times \cdots \times T_l$. The serialized OCaml value that corresponds to an element of this table is*

$$\mathbb{G}_{\mathsf{tblel}}(Tbl, (b_1, \ldots, b_l)) \stackrel{\text{def}}{=} (\mathsf{ser}(T_1, \mathbb{G}_{\mathsf{val}T_1}(b_1)), \ldots, \mathsf{ser}(T_l, \mathbb{G}_{\mathsf{val}T_l}(b_l)))\,.$$

Let $t = [a_1; \ldots; a_k]$ be the contents of the table Tbl: each a_i is an element of the

table. Let us denote

$$\mathbb{G}_{\mathsf{tbl}}(Tbl, t) \stackrel{\text{def}}{=} [\mathbb{G}_{\mathsf{tblel}}(a_1); \ldots; \mathbb{G}_{\mathsf{tblel}}(a_k)]$$

the OCaml list corresponding to t.

Definition 6.7 (Minimal environment and global store)

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

$$\begin{aligned} globalstore(E, \mathcal{T}) \stackrel{\text{def}}{=} & \{f \mapsto \mathbb{G}_{\mathsf{tbl}}(Tbl, \mathcal{T}(Tbl)) \mid (Tbl, f) \in \mathsf{tables}\} \\ & \cup \{f \mapsto \mathsf{ser}(T_x, a) \mid (x[\,], f) \in \mathsf{files}, E(x[\,]) = a\} \\ & \cup \{f \mapsto \texttt{""} \mid (x[\,], f) \in \mathsf{files}, x \textit{ not defined in } E\} \end{aligned} \quad \text{(Globalstore)}$$

We define $env(E, M)$ and $env(E, Q)$ in the same way.

The *globalstore* function defined above returns the global store in which the contents of the files and the tables is correct with respect to the CryptoVerif configuration elements E and $\mathcal{T}$. The *env* function returns the environment corresponding to E for the free variables in P (or M, or Q).

First, we show a correspondence between a CryptoVerif term and its OCaml translation.

Lemma 6.8 (Term reduction) *Let M be a CryptoVerif term of type T. If*

$$Th = \langle env, \mathbb{G}_{\mathsf{M}}(M), stack, store\rangle \textit{ with } env \supseteq env_{\mathsf{prim}} \cup env(E, M)\,,$$

and $E, M \Downarrow a$, then $Th \rightarrow^* Th'$ where

$$Th' \stackrel{\text{def}}{=} \langle env', \mathbb{G}_{\mathsf{val}T}(a), stack, store'\rangle$$

for some env' and $store'$ such that $store' \supseteq store$.

Proof We prove this result by induction on the syntax of terms.

- Case $M = x[\widetilde{a'}]$: Since $E, M \Downarrow a$ is derived by (Var), we have $E(x[\widetilde{a'}]) = a$. Since $env(E, M) \subseteq env$, we have $env(\mathbb{G}_{\mathsf{var}}(x)) = \mathbb{G}_{\mathsf{val}T}(a)$, so

$$Th = \langle env, \mathbb{G}_{\mathsf{var}}(x), stack, store\rangle \rightarrow Th' = \langle env, \mathbb{G}_{\mathsf{val}T}(a), stack, store\rangle$$

- Case $M = f(M_1, \ldots, M_k)$, where f is of type $T_1 \times \cdots \times T_k \rightarrow T$. Since $E, M \Downarrow a$ is derived by (Fun), we have $E, M_i \Downarrow a_i$ for all $i \leq k$, for some $a_1, \ldots, a_k$ such that $f(a_1, \ldots, a_k) = a$.

$$\begin{aligned} Th &= \langle env, \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_k)), stack, store\rangle \\ &\rightarrow \langle env, (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_k)), stack', store\rangle \\ &\quad \text{where } stack' \stackrel{\text{def}}{=} (env, \mathbb{G}_{\mathsf{f}}(f)\ [\cdot]) :: stack \end{aligned}$$

$\to Th_1 \stackrel{\text{def}}{=} \langle env, \mathbb{G}_{\mathsf{M}}(M_k), stack'', store\rangle$

where $stack'' \stackrel{\text{def}}{=} (env, (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{k-1}), [\cdot])) :: stack'$

$\to^* Th_2 \stackrel{\text{def}}{=} \langle env', \mathbb{G}_{\mathsf{val}T_k}(a_k), stack'', store'\rangle$

by induction hypothesis applied to M_k

$\to \langle env, (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{k-1}), \mathbb{G}_{\mathsf{val}T_k}(a_k)), stack', store'\rangle$

$\to^* Th_3 \stackrel{\text{def}}{=} \langle env, (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), stack', store''\rangle$

by an easy induction

$\to \langle env, \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), stack, store''\rangle$

$\to \langle env, \mathbb{G}_{\mathsf{f}}(f), stack''', store''\rangle$

where $stack''' \stackrel{\text{def}}{=} (env, [\cdot]\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k))) :: stack$

$\to \langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{f}}(f)), stack''', store''\rangle$ since $env_{\mathsf{prim}} \subseteq env$

$\to \langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{f}}(f))\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), stack, store''\rangle$

$\to^* Th' \stackrel{\text{def}}{=} \langle env'', \mathbb{G}_{\mathsf{val}T}(a), stack, store'''\rangle$ by Proposition 6.5

By Proposition 6.5 and induction hypothesis, we have $store''' \supseteq store'' \supseteq store' \supseteq store$. □

Let us now introduce some notations.

Definition 6.9 (Helper functions) *For an OCaml configuration*

$$\mathcal{C} = [Th_1, \ldots, Th_n], globalstore, tj', \mathcal{MI}, events$$

with $Th_{tj} = \langle env_{tj}, pe_{tj}, stack_{tj}, store_{tj}\rangle$ for all $tj \leq n$, let us define the following functions:

- $\mathcal{C}_{pe}(\mathcal{C}, tj) \stackrel{\text{def}}{=} pe_{tj}$, $\quad \mathcal{C}_{pe}(\mathcal{C}) \stackrel{\text{def}}{=} \mathcal{C}_{pe}(\mathcal{C}, tj')$,
- $\mathcal{C}_{Th}(\mathcal{C}, tj) \stackrel{\text{def}}{=} Th_{tj}$, $\quad \mathcal{C}_{Th}(\mathcal{C}) \stackrel{\text{def}}{=} \mathcal{C}_{Th}(\mathcal{C}, tj')$,
- $\mathcal{C}_{t}(\mathcal{C}) \stackrel{\text{def}}{=} tj'$,
- $\mathcal{C}_{tmax}(\mathcal{C}) \stackrel{\text{def}}{=} n$,
- $\mathcal{C}_{globalstore}(\mathcal{C}) \stackrel{\text{def}}{=} globalstore$,
- $\mathcal{C}_{events}(\mathcal{C}) \stackrel{\text{def}}{=} events$,

We also define these functions on traces, where they return the elements in the last configuration of the trace.

We also define

$$\mathcal{C}[\mathsf{Th} \mapsto Th', \mathsf{globalstore} \mapsto globalstore', \mathsf{MI} \mapsto \mathcal{MI}', \mathsf{events} \mapsto events'] \stackrel{\text{def}}{=}$$
$$[Th_1, \ldots, Th_{tj'-1}, Th', Th_{tj'+1}, \ldots, Th_n], globalstore', tj', \mathcal{MI}', events'\,.$$

In this notation, one can omit globalstore, MI, *or* events. *When omitted, we keep the corresponding element of the configuration $\mathcal{C}$.*

Next, we prove that the CryptoVerif oracle bodies P are correctly translated into OCaml as $\mathbb{G}(P)$. We extend the translation $\mathbb{G}(P)$ to processes in which some replication indices have been instantiated into their values, using the formulas of Section 5 where replication indices i may be replaced with their value a. It is easy to see that $\mathbb{G}(P\{a/i\}) = \mathbb{G}(P)$.

Lemma 6.10 (Inner reduction) *Let $\mathfrak{C}$ be a CryptoVerif configuration. Suppose that the program part P of $\mathfrak{C}$ is not in a return, end, call, or loop form. Suppose that we have n possible 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

$$\begin{aligned} \mathcal{C}_{Th}(\mathcal{C}) &= \langle env, \mathbb{G}(P), stack, store\rangle \text{ with } env \supseteq env_{\mathsf{prim}} \cup env(E, P)\,, \\ \mathcal{C}_{globalstore}(\mathcal{C}) &\supseteq globalstore(E, \mathcal{T})\,, \\ \mathcal{C}_{events}(\mathcal{C}) &= \mathbb{G}_{\mathsf{ev}}(\mathcal{E})\,. \end{aligned}$$

Then there exist n disjoint sets of OCaml traces $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ all starting at $\mathcal{C}$ such that none of these traces is a prefix of another of these traces, $\Pr[\mathcal{CTS}_i] = p_i$ for all $i \leq n$, and if $\mathcal{C}'$ is the last configuration of a trace in $\mathcal{CTS}_i$, then we have $\mathcal{C}' = \mathcal{C}[\mathsf{Th} \mapsto Th', \mathsf{globalstore} \mapsto globalstore', \mathsf{events} \mapsto events']$ where

$$\begin{aligned} Th' &= \langle env', \mathbb{G}(P_i), stack, store'\rangle \\ &\quad \text{with } env' \supseteq env_{\mathsf{prim}} \cup env(E_i, P_i) \text{ and } store' \supseteq store\,, \\ globalstore' &\supseteq globalstore(E_i, \mathcal{T}_i)\,, \\ globalstore'(l) &= \mathcal{C}_{globalstore}(\mathcal{C})(l) \text{ for all } l \notin S_{priv}\,, \\ events' &= \mathbb{G}_{\mathsf{ev}}(\mathcal{E}_i)\,. \end{aligned}$$

The proof of this lemma can be found in Appendix D. This lemma is proved by cases on the process P. We use Lemma 6.8 when we need to evaluate a term. The cases end and return will be handled when we prove the invariant for the whole system. The oracle bodies that we translate into OCaml do not contain calls nor loops. This lemma shows that the following invariants are preserved during the evaluation of oracle bodies: the OCaml environment and global store contain the minimal environment and global store corresponding to the CryptoVerif configuration; the public part of the global store does not change; the OCaml and CryptoVerif events match. Locations may be added in the store, but the contents of existing locations does not change.

6.3 Simulation of the OCaml Adversary

In this section, we show how to simulate in CryptoVerif any OCaml program $program_0$ that corresponds to an adversary interacting with the protocol implementation generated from the CryptoVerif process Q_0. Basically, we run the OCaml program $program_0$ inside the CryptoVerif primitive *simulate* (which is

possible since these primitives can represent any deterministic Turing machine). When $program_0$ needs to call an oracle of Q_0, the primitive returns and the call is made by CryptoVerif. When $program_0$ needs to generate a random number, this generation is performed by CryptoVerif.

We assume that the OCaml program $program_0$ runs in bounded time, so makes a bounded number of oracle calls. By Assumption 2.11, when an oracle O (resp. role role) is under replication, this replication has bound N_O (resp. N_{role}). When oracle O is under replication, we let N_O be the maximum number of calls to the same closure $\mathsf{tagfunction}^{O,\tau}[env, pm]$ corresponding to oracle O. When a role role is under replication, we let N_{role} be the maximum number of executions of $\mathsf{addthread}(program)$ for some $program$ that contains μ_{role}. These replication bounds are chosen such that the OCaml program $program_0$ never exhausts the number of oracle calls allowed by the CryptoVerif process. We let $N_{\mathsf{rand+calls}}$ be the maximum number of oracle calls and random number generations that the OCaml program $program_0$ can make plus one. We let N_{steps} be the maximum number of reduction steps of the program $program_0$ in the semantics of OCaml. Formally, we use the following definition:

Definition 6.11 *The number of calls to the closure with tag O, τ in a trace $\mathcal{CT}$, denoted $N_{\mathsf{calls}}(O, \tau, \mathcal{CT})$, is the number of configurations $\mathcal{C}$ such that $\mathcal{C}_{pe}(\mathcal{C}) = \mathsf{tagfunction}^{O,\tau}[env, pm]$ v in $\mathcal{CT}$ excluding its last configuration.*

The number of executions of role role in a trace $\mathcal{CT}$, denoted $N_{\mathsf{exec}}(\mathsf{role}, \mathcal{CT})$, is the number of configurations $\mathcal{C}$ such that $\mathcal{C}_{pe}(\mathcal{C}) = \mathsf{addthread}(program)$ where $program$ contains $program(\mu_{\mathsf{role}})$ in $\mathcal{CT}$ excluding its last configuration.

The number of random number generations in a trace $\mathcal{CT}$, denoted $N_{\mathsf{rand}}(\mathcal{CT})$, is the number of transitions derived by rule (Random) *in $\mathcal{CT}$.*

We define

$$
\begin{aligned}
N_O &\stackrel{\text{def}}{=} \max_{\mathcal{CT},\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT}) \\
N_{\mathsf{role}} &\stackrel{\text{def}}{=} \max_{\mathcal{CT}} N_{\mathsf{exec}}(\mathsf{role}, \mathcal{CT}) \\
N_{\mathsf{rand+calls}} &\stackrel{\text{def}}{=} \max_{\mathcal{CT}} \left(N_{\mathsf{rand}}(\mathcal{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT}) \right) + 1 \\
N_{\mathsf{steps}} &\stackrel{\text{def}}{=} \max_{\mathcal{CT}} |\mathcal{CT}|
\end{aligned}
$$

where $\mathcal{CT}$ ranges over traces that begin with the configuration $\mathcal{C}_0(Q_0, program_0)$.

While N_O is an optimal bound, N_{role} is not optimal. Consider for instance a process of the form

$$\mathsf{foreach}\ i \leq N_O\ \mathsf{do}\ O() := \ldots\}\ \mathsf{foreach}\ j \leq N_{\mathsf{role}}\ \mathsf{do}\ \mathsf{role}\ \{\ldots$$

By distributing the instantiations of role on every available index i, the optimal bound of the replication j is the maximum during all executions of $program_0$ of the number of instantiations of role divided by the number of calls to O made

```
1  Q_adv(Q_0, program_0) = Q_start(Q_0, program_0) | Q_c(Q_0, program_0)
2  Q_start(Q_0, program_0) = O_start() :=
3        s_0 : T_CS ← s_0(Q_0, program_0);
4        let r : T_CS = loop O_loop(s_0) in end else end
5  Q_c(Q_0, program_0) = foreach i' ≤ N_rand+calls do
6    O_loop[i'](s : T_CS) :=
7        let (s' : T_CS, o : T_o, i : bitstring, args : bitstring) = simulate(s) in
8        if o = o_S then
9           return(s', false)
10       else if o = o_1 then
11          let (a_{1,1} : T_{1,1}, ..., a_{1,m_1} : T_{1,m_1}) = args in
12          let (i_{1,1} : [1, N_{1,1}], ..., i_{1,n_1} : [1, N_{1,n_1}]) = i in
13          let (r_{1,1} : T'_{1,1}, ..., r_{1,m'_1} : T'_{1,m'_1}) = O_1[i_{1,1}, ..., i_{1,n_1}](a_{1,1}, ..., a_{1,m_1}) in
14             return(simulate'_{O_1}(s', (r_{1,1}, ..., r_{1,m'_1})), true)
15          else return(simulate''_{O_1}(s'), true)
16       else if o = o_2 then
17          ⋮
18       else if o = o_R then
19          b_R ←R bool;
20          return(simulate_R(s', b_R), true)
```

Figure 19: The program $Q_{\mathrm{adv}}(Q_0, \mathit{program}_0)$

before these instantiations of role. To get this optimal bound, we would need to associate each new instantiation of role to the index i with the least number of associated instantiations of role. Since a role is often under at most one replication, we decided not to complicate the proof with details needed to get the optimal bound.

From the OCaml program $\mathit{program}_0$, we define a CryptoVerif adversary $Q_{\mathrm{adv}}(Q_0, \mathit{program}_0)$ given in Figure 19 and explained below. We will prove that this process, when executed in parallel with Q_0, has the same behavior as the OCaml program $\mathit{program}_0$. The initial CryptoVerif configuration is then $\mathfrak{C}_0(Q_0, \mathit{program}_0) = \mathfrak{C}_{\mathrm{i}}(Q_0 \mid Q_{\mathrm{adv}}(Q_0, \mathit{program}_0))$.

In Figure 19, we use a let construct with pattern matching, which can be defined as follows. We define the function $\mathit{tuple}_{T_1,\ldots,T_j} : T_1 \times \cdots \times T_j \to \mathit{bitstring}$ that creates a tuple with j elements (for instance by concatenating the j bitstrings with information on their length, so that they can be unambiguously recovered), and the associated projections $\pi_{k,T_1,\ldots,T_j} : \mathit{bitstring} \to T_k$ with $k \leq j$ (which may return any value when their argument is not a tuple with j elements). The construct let $(x_1 : T_1, \ldots, x_j : T_j) = M$ in P is an abbreviation

for:

$$x \leftarrow M; x_1 \leftarrow \pi_{1,T_1,\ldots,T_j}(x); \ldots; x_j \leftarrow \pi_{j,T_1,\ldots,T_j}(x);\\ \mathsf{if}\ x = \mathit{tuple}_{T_1,\ldots,T_j}(x_1,\ldots,x_j)\ \mathsf{then}\ P$$

where x is a fresh variable. The CryptoVerif term $(M_1, \ldots, M_j)$ is an abbreviation for $\mathit{tuple}_{T_1,\ldots,T_j}(M_1, \ldots, M_j)$, where $T_1, \ldots, T_j$ are the types of $M_1, \ldots, M_j$, respectively.

Let $O_1, \ldots, O_n$ be the oracle names in Q_0. We define n constants $o_1, \ldots, o_n$ which are used to designate the oracles $O_1, \ldots, O_n$ respectively, o_R which corresponds to a random choice, and o_S which corresponds to the end of the OCaml program. We define the CryptoVerif type $T_o \stackrel{\text{def}}{=} \{o_R, o_S, o_1, \ldots, o_n\}$, which contains all these bitstring constants.

The adversary is mainly encoded by the function *simulate*. This function takes as argument the bitstring representation $s = \mathsf{repr}(\mathcal{CS})$ of a simulator configuration $\mathcal{CS}$. The configuration $\mathcal{CS}$ consists of a non-instrumented OCaml configuration $\mathcal{C}$ (with some extensions to the syntax described later) and sets $\mathcal{RI}$ and $\mathcal{I}$ that finitely represent the callable oracles $\mathcal{Q}$ of the CryptoVerif configuration:

$$\mathcal{CS} = \underbrace{([Th_1, \ldots, Th_n], \mathit{globalstore}, i)}_{\mathcal{C}}, \mathcal{RI}, \mathcal{I}\,.$$

The function repr is injective. We denote its inverse by repr^{-1}. We also define a CryptoVerif type $T_{\mathcal{CS}}$ that consists of all bitstrings in the image of repr, that is, all bitstrings that correspond to simulator configurations $\mathcal{CS}$. We also use the notations of Definition 6.9 for simulator configurations.

When we call an oracle or instantiate a role under replication, we must choose an unused replication index for this replication, and call the oracle or instantiate the role with that replication index. In this simulation, we will always choose the smallest replication index that has not been used yet, so that the used indices form an interval $[1, a-1]$ and the unused indices are in $[a, N]$ where N is the bound of the considered replication. The sets $\mathcal{RI}$ and $\mathcal{I}$ represent the sets of callable roles and oracles, by storing the smallest index a that is not used yet.

More precisely, the set $\mathcal{RI}$ represents the set of callable roles with their replication indices. Elements of $\mathcal{RI}$ are either:

- of the form $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$, which means the role role is under replication, the roles $\mathsf{role}[1, \widetilde{a'}]$ to $\mathsf{role}[a-1, \widetilde{a'}]$ have been used, and the roles $\mathsf{role}[a, \widetilde{a'}]$ to $\mathsf{role}[N_{\mathsf{role}}, \widetilde{a'}]$ are usable,
- or of the form $\mathsf{role}[\widetilde{a}]$, which means that role is not under replication and the role role is callable with the replication indices $\widetilde{a}$.

The set $\mathcal{RI}$ never contains simultaneously $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$ and $\mathsf{role}[a'', \widetilde{a'}]$ for the same role and $\widetilde{a'}$, and it never contains simultaneously $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$ and $\mathsf{role}\big[[a'', +\infty[, \widetilde{a'}\big]$ with $a \neq a''$ for the same role and $\widetilde{a'}$.

The set $\mathcal{I}$ represents the set of callable oracles with their replication indices. Elements of $\mathcal{I}$ are either:

- of the form $O\big[[a, +\infty[, \widetilde{a'}\big]$, which means that the oracle O is under replication and the oracles $O[1, \widetilde{a'}]$ to $O[a-1, \widetilde{a'}]$ have been used, and the oracles $O[a, \widetilde{a'}]$ to $O[N_O, \widetilde{a'}]$ are usable,
- or of the form $O[\widetilde{a}]$ which means that O is an oracle not under replication that can be called with the replication indices $\widetilde{a}$.

The set $\mathcal{I}$ never contains simultaneously $O\big[[a, +\infty[, \widetilde{a'}\big]$ and $O[a'', \widetilde{a'}]$ for the same O and $\widetilde{a'}$, and it never contains simultaneously $O\big[[a, +\infty[, \widetilde{a'}\big]$ and $O\big[[a'', +\infty[, \widetilde{a'}\big]$ with $a \neq a''$ for the same O and $\widetilde{a'}$.

Next, we define functions that manipulate these sets of oracles and roles. We define the subtraction operation $\mathcal{I} - O[\widetilde{a}]$ on sets of oracles.

- If $O\big[[a, +\infty[, \widetilde{a'}\big]$ is in $\mathcal{I}$, then

$$\mathcal{I} - (O[a, \widetilde{a'}]) \stackrel{\text{def}}{=} \mathcal{I} \setminus \{O\big[[a, +\infty[, \widetilde{a'}\big]\} \cup \{O\big[[a+1, +\infty[, \widetilde{a'}\big]\}\,.$$

- If $O[\widetilde{a}]$ is in $\mathcal{I}$, then

$$\mathcal{I} - (O[\widetilde{a}]) \stackrel{\text{def}}{=} \mathcal{I} \setminus \{O[\widetilde{a}]\}\,.$$

We define similarly the subtraction on sets of roles $\mathcal{RI} - \mathsf{role}[\widetilde{a}]$. We also generalize this operator to sets:

$$\mathcal{RI} - \{\mathsf{role}_1[\widetilde{a_1}], \ldots, \mathsf{role}_k[\widetilde{a_k}]\} \stackrel{\text{def}}{=} (\ldots(\mathcal{RI} - \mathsf{role}_1[\widetilde{a_1}]) - \ldots) - \mathsf{role}_k[\widetilde{a_k}]\,.$$

We let $\mathsf{smallest}(\mathcal{RI}, \mathsf{role})$ be the smallest indices present for the role role in $\mathcal{RI}$: when $\widetilde{a} = \mathsf{smallest}(\mathcal{RI}, \mathsf{role})$, we have $\mathsf{role}[\widetilde{a}] \in \mathcal{RI}$ or there exist a' and $\widetilde{a'}$ such that $\widetilde{a} = a', \widetilde{a'}$ and $\mathsf{role}\big[[a', +\infty[, \widetilde{a'}\big] \in \mathcal{RI}$.

Let us define the function $\mathsf{oracles'}$, which is similar to the function reduce$'$, but just returns the oracle name and its replication indices $\widetilde{i}$ (which can be partly instantiated to values), instead of returning the entire oracle definition:

$$\mathsf{oracles'}(0) \stackrel{\text{def}}{=} [\,] \tag{Nil}$$

$$\mathsf{oracles'}(Q_1 \mid Q_2) \stackrel{\text{def}}{=} \mathsf{oracles'}(Q_1) \mathbin{@} \mathsf{oracles'}(Q_2) \tag{Par}$$

$$\mathsf{oracles'}(\mathsf{foreach}\ i' \leq n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} \big[O_1[\_, \widetilde{i}], \ldots, O_l[\_, \widetilde{i}]\big] \text{ when } \mathsf{oracles'}(Q) = \big[O_1[i', \widetilde{i}], \ldots, O_l[i', \widetilde{i}]\big] \tag{Repl}$$

$$\mathsf{oracles'}(\mathsf{role}\,\{Q) \stackrel{\text{def}}{=} [\,] \tag{Role}$$

$$\mathsf{oracles'}(O[\widetilde{i}](x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]) := P) \stackrel{\text{def}}{=} \big[O[\widetilde{i}]\big] \tag{Oracle}$$

This function returns elements of the form $O[\widetilde{i}]$ for oracles that are not directly under replication and $O[\_, \widetilde{i}]$ for oracles directly under replication. Similarly to reduce$'$, this function returns an empty list when encountering a role definition.

Let us consider a process $Q' = \mathsf{foreach}\ i' \leq n\ \mathsf{do}\ Q$. By Assumption 2.10, there is no replication in Q, and so all oracles in Q are under the same replications and have exactly the same replication indices $i', \widetilde{i}$, where the indices $\widetilde{i}$ are the replication indices of replications above Q'. So, by rule (Repl), $\mathsf{oracles}'(Q')$ produces the list of callable oracles in Q where we replace the replication index i' with $\_$.

By Property 2.4, an oracle with a certain name O always takes arguments of the same types and always returns values of the same types. So we can say that the oracle O_i takes m_i arguments of types $T_{i,1}, \ldots, T_{i,m_i}$, and returns m'_i bitstrings of types $T'_{i,1}, \ldots, T'_{i,m'_i}$. We can also define $\mathsf{returnoracles}(O[\widetilde{i}]) \stackrel{\text{def}}{=} \mathsf{oracles}'(Q)$ where Q is an oracle definition located after a return statement in a body of the oracle $O[\widetilde{i}]$ in Q_0. This definition is correct because, by Property 2.4, the structure of the processes Q after any return statement of a given oracle O is always the same, so the list $\mathsf{oracles}'(Q)$ will be the same for each of these Q. The function $\mathsf{returnoracles}$ can take an oracle with its replication indices partly instantiated to values: $\mathsf{returnoracles}(O[\widetilde{a}]) \stackrel{\text{def}}{=} \mathsf{returnoracles}(O[\widetilde{i}])\{\widetilde{a}/\widetilde{i}\}$.

Let us denote by $Q(\mathsf{role})$ the subprocess of Q_0 that corresponds to the role role. For a subprocess Q of Q_0 that is under replication indices $\widetilde{i}$ in Q_0, we denote $Q[\widetilde{a}]$ the process Q where we substituted elements of $\widetilde{i}$ by the respective elements of $\widetilde{a}$.

Definition 6.12 (First oracle) *The* first oracles *of a role* role *are the oracles that can be called when we are at the beginning of the subprocess corresponding to the role, that is,* $\mathsf{oracles}'(Q(\mathsf{role}))$.

We define $\mathsf{add}(\mathcal{I}, \mathcal{RI})$ as the addition of the first oracles present in $\mathcal{RI}$ to $\mathcal{I}$:

$$\mathsf{add}(\mathcal{I}, \mathcal{RI}) \stackrel{\text{def}}{=} \mathcal{I} \cup \{O[\widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[\widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])\} \cup$$
$$\{O\big[[1, +\infty[, \widetilde{a}\big] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[\_, \widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])\}$$

The syntax of the language of the simulator is almost the same as the language we described in Section 3, with the addition of tagged functions introduced in Section 4. We add the functional values $\mathsf{call}(O[\widetilde{a}])$ and $\mathsf{call}(O[\_, \widetilde{a}])$ that replace our generated closures for the oracle O. The value $\mathsf{call}(O[\widetilde{a}])$ is used when O is not directly under replication; $\mathsf{call}(O[\_, \widetilde{a}])$ is used when O is directly under replication.

We present the semantics followed by our simulator in Figure 20. When we encounter a configuration containing a successful call to an oracle (by call) or a random operation, we cannot reduce. These operations are executed, but not inside the simulator: we stop the simulator in its current state, and in CryptoVerif, we call the requested oracle with the requested arguments, or generate a random bit. Otherwise, when the simulator configuration reduces into another configuration in the OCaml semantics, by rule (Simulator toplevel), we also reduce in the same way. By rules (FailedCall1) and (FailedCall2), we raise the exception $\mathsf{Bad\_Call}$ when the call to the oracle is invalid, as our generated code does in this case. Notice that, in the OCaml implementation, the

$O[\widetilde{a}] \notin \mathcal{I}$
or O does not have k arguments
or O has k arguments of type $T_1, \ldots, T_k$ and $\exists i,\ \nexists a_i, v_i = \mathbb{G}_{\mathsf{val}T_i}(a_i)$

$$env, \mathsf{call}(O[\widetilde{a}])\ (v_1, \ldots, v_k), stack \longrightarrow env, \mathsf{raise\ Bad\_Call}, stack$$

(FailedCall1)

$\forall a', O\big[[a', +\infty[, \widetilde{a}\big] \notin \mathcal{I}$
or O does not have k arguments
or O has k arguments of type $T_1, \ldots, T_k$ and $\exists i,\ \nexists a_i, v_i = \mathbb{G}_{\mathsf{val}T_i}(a_i)$

$$env, \mathsf{call}(O[\_, \widetilde{a}])\ (v_1, \ldots, v_k), stack \longrightarrow env, \mathsf{raise\ Bad\_Call}, stack$$

(FailedCall2)

$\mathcal{C} \to \mathcal{C}'$ using the rules of Figures 7–14, (FailedCall1), and (FailedCall2)
but not (Random) and (Toplevel add thread)

$$\mathcal{C}, \mathcal{RI}, \mathcal{I} \to \mathcal{C}', \mathcal{RI}, \mathcal{I}$$

(Simulator toplevel)

$program^a = program_{\mathsf{prim}};; program(\mu_{\mathsf{role}_1});; \ldots;; program(\mu_{\mathsf{role}_l});; program'$
$program'$ does not contain $program(\mu_{\mathsf{prim}})$ nor any $program(\mu)$ for $\mu \in \mathcal{M}_g$
$\{\mu_{\mathsf{role}_1}, \ldots, \mu_{\mathsf{role}_l}\} \subseteq \mathcal{M}_g$
$\widetilde{a_1} = \mathsf{smallest}(\mathcal{RI}, \mathsf{role}_1), \ldots, \widetilde{a_l} = \mathsf{smallest}(\mathcal{RI}, \mathsf{role}_l)$
$\mathcal{RI}'' = \{\mathsf{role}_1[\widetilde{a_1}], \ldots, \mathsf{role}_l[\widetilde{a_l}]\} \qquad \mathcal{RI}' = \mathcal{RI} - \mathcal{RI}'' \qquad \mathcal{I}' = \mathsf{add}(\mathcal{I}, \mathcal{RI}'')$
$program^b = program_{\mathsf{prim}};; program'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots;; program'(\mathsf{role}_l[\widetilde{a_l}]);;$
$program'$
or
$program^a$ does not contain $program(\mu_{\mathsf{prim}})$ nor any $program(\mu)$ for $\mu \in \mathcal{M}_g$
$\mathcal{RI}'' = \emptyset \qquad \mathcal{RI}' = \mathcal{RI} \qquad \mathcal{I}' = \mathcal{I} \qquad program^b = program^a$

$$[Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{addthread}(program^a), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], globalstore, tj, \mathcal{RI}, \mathcal{I} \longrightarrow$$
$$[Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n, \langle \emptyset, program^b, [\,], \emptyset\rangle], globalstore, tj, \mathcal{RI}', \mathcal{I}'$$

(Simulator add thread)

Figure 20: Semantics followed by the simulator

adversary can test whether an oracle call succeeds or not, by catching the exception Bad_Call. In CryptoVerif, failed calls can happen only when the called oracle is not available, and in this case, the reduction blocks. This different behavior does not give additional power to the OCaml adversary, because the adversary can test before performing the call whether it will succeed or not. The rules (FailedCall1) and (FailedCall2) implement this test. By rule (Simulator add thread), we modify the behavior of the addthread construct to transform references to our generated modules $program(\mu_{\mathsf{role}})$ into references to the corresponding role $program'(\mathsf{role}[\widetilde{a}])$ where $\widetilde{a}$ are the replication indices we chose for this particular reference and

$$program'(\mathsf{role}[\widetilde{a}]) \stackrel{\text{def}}{=} \mathsf{let}\ \mu_{\mathsf{role}}.\mathsf{init} = \mathsf{let}\ token = \mathsf{ref}\ \mathrm{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}}\ pm'_{\mathsf{role}[\widetilde{a}]}$$
where $pm'_{\mathsf{role}[\widetilde{a}]} \stackrel{\text{def}}{=} () \rightarrow$
$\quad \mathsf{if}\ (!token)\ \mathsf{then}\ (token := \mathrm{false}; (\mathsf{call}(O_1[\widetilde{a_1}]), \ldots, \mathsf{call}(O_k[\widetilde{a_k}])))$
$\qquad\qquad \mathsf{else\ raise\ Bad\_Call}$

where $\mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}]) = [O_1[\widetilde{a_1}], \ldots, O_k[\widetilde{a_k}]]$, and the $\widetilde{a_j}$ are either $\widetilde{a}$ or $\_, \widetilde{a}$. In particular, the initialization function defined in $program'(\mathsf{role}[\widetilde{a}])$ returns oracles represented by call values instead of closures.

The CryptoVerif function $simulate : T_{\mathcal{CS}} \rightarrow bitstring$ follows the simulator semantics defined in Figure 20: we define $simulate(\mathsf{repr}(\mathcal{CS})) \stackrel{\text{def}}{=} simreturn(\mathcal{CS}')$ where $\mathcal{CS}'$ is the configuration such that either $\mathcal{CS}$ reduces into $\mathcal{CS}'$ in at most N_{steps} reductions and $\mathcal{CS}'$ does not reduce, or $\mathcal{CS}$ reduces into $\mathcal{CS}'$ in exactly N_{steps} reductions, by the semantics of Figure 20, and $simreturn(\mathcal{CS}')$ is defined below. (We need to bound the number of reductions to make sure that *simulate* is always defined. The proof of the simulation between OCaml and CryptoVerif, presented in the next section, shows that the simulator configuration always blocks after at most N_{steps} reductions, so that we are always in the first case.)

- If $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{call}(O[\widetilde{a}])\ (v_1, \ldots, v_l)$, let $T_1, \ldots, T_l$ be the type of the arguments of the oracle O and let o be the constant associated to O. We define

$$simreturn(\mathcal{CS}') \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}'), o, \widetilde{a}, (\mathbb{G}^{-1}_{\mathsf{val}T_1}(v_1), \ldots, \mathbb{G}^{-1}_{\mathsf{val}T_l}(v_l)))\,.$$

- If $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{call}(O[\_, \widetilde{a}])\ (v_1, \ldots, v_l)$, let $T_1, \ldots, T_l$ be the type of the arguments of the oracle O, let o be the constant associated to O, and let a' be the value such that $O\big[[a', +\infty[, \widetilde{a}\big]$ is in the set $\mathcal{I}$ where $\mathcal{CS}' = \mathcal{C}, \mathcal{RI}, \mathcal{I}$. We define

$$simreturn(\mathcal{CS}') \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}'), o, (a', \widetilde{a}), (\mathbb{G}^{-1}_{\mathsf{val}T_1}(v_1), \ldots, \mathbb{G}^{-1}_{\mathsf{val}T_l}(v_l)))\,.$$

- If $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{random}\ ()$, we define

$$simreturn(\mathcal{CS}') \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}'), o_R, (), ())\,.$$

- Otherwise, we define

$$simreturn(\mathcal{CS}') \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}'), o_S, (), ()) .$$

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

When *simulate* returns $(\mathsf{repr}(\mathcal{CS}'), o, \widetilde{a}, (a_1, \ldots, a_l))$, the CryptoVerif process $Q_c(Q_0, program_0)$ performs the corresponding oracle call $O[\widetilde{a}](a_1, \ldots, a_l)$ (lines 10–17 of Figure 19). Similarly, when *simulate* returns $(\mathsf{repr}(\mathcal{CS}'), o_R, ()$, $())$, the process $Q_c(Q_0, program_0)$ performs a random choice (lines 18–20), and when *simulate* returns $(\mathsf{repr}(\mathcal{CS}'), o_S, (), ())$, the process $Q_c(Q_0, program_0)$ terminates (lines 8–9; the corresponding OCaml program also terminates).

The functions $simulate'_O$ and $simulate''_O$ replace, in the simulator configuration, the call expression with the result returned by the oracle, and raise the Match_failure exception, respectively. The function $simulate'_O$ handles the situation in which an oracle returns a result by return; the function $simulate''_O$ handles the situation in which the oracle terminates with end. Formally, these functions are defined as follows.

Definition 6.13 (Simulation of oracle return) *Let us consider a simulator configuration $\mathcal{CS} = \mathcal{C}, \mathcal{RI}, \mathcal{I}$, with*

$$\mathcal{C}_{pe}(\mathcal{CS}) = \mathsf{call}(O[\widetilde{a}])\ (v_1, \ldots, v_l) \text{ or } \mathsf{call}(O[\_, \widetilde{a'}])\ (v_1, \ldots, v_l) .$$

When $\mathcal{C}_{pe}(\mathcal{CS})$ is of the second form, we denote by $\widetilde{a}$ the indices $a'', \widetilde{a'}$ where a'' is such that $O\big[[a'', +\infty[, \widetilde{a'}\big] \in \mathcal{I}$. Let $\mathcal{I}' \stackrel{\text{def}}{=} \mathcal{I} - (O[\widetilde{a}])$.

We define the CryptoVerif function $simulate'_O : T_{\mathcal{CS}} \times bitstring \to T_{\mathcal{CS}}$ as follows.

If the returns in oracle O end the current role, then by Property 2.7, there is only one return *statement in O; let Q be the oracle definition following this statement, and let*

$$\begin{aligned}\mathcal{RI}' \stackrel{\text{def}}{=} &\{\mathsf{role}[\widetilde{a}] \mid (\mu_{\mathsf{role}}, \text{false}) \in \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q)\} \\ &\cup \{\mathsf{role}\big[[1, +\infty[, \widetilde{a}\big] \mid (\mu_{\mathsf{role}}, \text{true}) \in \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q)\} .\end{aligned}$$

Let $T_1, \ldots, T_n$ be the types of the return value of O. We define:

$$simulate'_O(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I}), (r_1, \ldots, r_n)) \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{C}', \mathcal{RI} \cup \mathcal{RI}', \mathcal{I}')$$

where $\mathcal{C}'$ is the configuration $\mathcal{C}$ in which the current expression is replaced with the translated result: $(\mathbb{G}_{\mathsf{val}T_1}(r_1), \ldots, \mathbb{G}_{\mathsf{val}T_n}(r_n))$.

If the returns in oracle O do not end the current role, then let us define $\mathcal{O} \stackrel{\text{def}}{=} \mathsf{returnoracles}(O[\widetilde{a}])$. Let $\mathcal{I}''$ be the set $\mathcal{I}'$ to which we added the oracles present in $\mathcal{O}$:

$$\mathcal{I}'' \stackrel{\text{def}}{=} \mathcal{I}' \cup \{O'\big[[1, +\infty[, \widetilde{a}\big] \mid O'[\_, \widetilde{a}] \in \mathcal{O}\} \cup \{O'[\widetilde{a}] \mid O'[\widetilde{a}] \in \mathcal{O}\} .$$

We define:

$$simulate'_O(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I}), (r_1, \ldots, r_n)) \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{C}', \mathcal{RI}, \mathcal{I}'')$$

where $\mathcal{C}'$ is the configuration $\mathcal{C}$ in which the current expression is replaced with the translated result: $(\mathsf{call}(O_1[\widetilde{a_1}]), \ldots, \mathsf{call}(O_l[\widetilde{a_l}]), \mathbb{G}_{\mathsf{val}T_1}(r_1), \ldots, \mathbb{G}_{\mathsf{val}T_n}(r_n))$, *with $\mathcal{O} = \{O_1[\widetilde{a_1}], \ldots, O_l[\widetilde{a_l}]\}$ and the $\widetilde{a_j}$ are either $\widetilde{a}$ or $\_, \widetilde{a}$.*

In all other cases (that is, $\mathcal{CS}$ is not of the form mentioned above or the bitstring a is not a tuple of n bitstrings of types $T_1, \ldots, T_n$), $simulate'_O(\mathsf{repr}(\mathcal{CS}), a)$ can take any value, since these cases are in fact not used.

Finally, we define the CryptoVerif function $simulate''_O : T_{\mathcal{CS}} \to T_{\mathcal{CS}}$ by:

$$simulate''_O(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I})) \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{C}'', \mathcal{RI}, \mathcal{I}')$$

where $\mathcal{C}''$ is the configuration $\mathcal{C}$ in which the current expression is replaced with raise Match_failure. *In all other cases (that is, $\mathcal{CS}$ is not of the form mentioned above), $simulate''_O(\mathsf{repr}(\mathcal{CS}))$ can take any value, since these cases are in fact not used.*

When the returns in oracle O end the current role, the function $simulate'_O$ does not return the oracles following the current oracle, but adds the corresponding roles to the role set $\mathcal{RI}$. The programs that contain these roles can then be launched by addthread.

Definition 6.14 (Random simulation) *We define the CryptoVerif function $simulate_R : T_{\mathcal{CS}} \times bool \to T_{\mathcal{CS}}$ by*

$$simulate_R(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I}), b) \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{C}'(b), \mathcal{RI}, \mathcal{I})$$

where $\mathcal{C}'(b)$ is the configuration $\mathcal{C}$ in which the current expression is replaced with the OCaml boolean value $\mathbb{G}_{\mathsf{val}bool}(b)$.

Let us finally define the initial state of the simulator. Let $\mathcal{RI}_0$ be the set of initially callable roles of Q_0 with their replication indices: $\mathcal{RI}_0 \stackrel{\text{def}}{=} \{\mathsf{role}[\,] \mid (\mu_{\mathsf{role}}, \text{false}) \in \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_0)\} \cup \{\mathsf{role}\big[[1, +\infty[\big] \mid (\mu_{\mathsf{role}}, \text{true}) \in \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_0)\}$. We define:

$$s_0(Q_0, program_0) \stackrel{\text{def}}{=} \mathsf{repr}((\langle\!|\langle\emptyset, program_0, [\,], \emptyset\rangle|\!\rangle, globalstore_0, 1), \mathcal{RI}_0, \emptyset)$$

6.4 Correspondence between the CryptoVerif and OCaml Systems

In this section, we prove our main security theorem by relating the CryptoVerif and OCaml systems.

In CryptoVerif, the 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 c of the form "for all a, if $e_1(a)$ has been executed, then $e_2(a)$ has also been executed", we define D_c by $D_c(\mathcal{E}) = \text{true}$ 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. We can represent the adversary for Q_0 by any CryptoVerif process Q_{adv} that does not contain events nor variables that occur in Q_0. Then we bound the probability $\Pr[\mathfrak{C}_{\text{i}}(Q_0 \mid Q_{\text{adv}}) : D_c]$, that is, the probability that the adversary Q_{adv} breaks the correspondence in Q_0, for any adversary Q_{adv} for Q_0. We can also define secrecy using events and distinguishers [6].

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 goal is to prove that, for all protocols Q_0, OCaml adversaries $\mathit{program}_0$, and distinguishers D, we have $\Pr[\mathfrak{C}_0(Q_0, \mathit{program}_0) : D] = \Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D]$. From this property, it is easy to see that, if CryptoVerif bounds the probability $\Pr[\mathfrak{C}_{\text{i}}(Q_0 \mid Q_{\text{adv}}) : D]$ for any adversary Q_{adv} for Q_0, then the same bound also holds for the probability $\Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D]$ corresponding to the generated implementation. Indeed, $\Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D] = \Pr[\mathfrak{C}_0(Q_0, \mathit{program}_0) : D] = \Pr[\mathfrak{C}_{\text{i}}(Q_0 \mid Q_{\text{adv}}(Q_0, \mathit{program}_0)) : D]$ and $Q_{\text{adv}}(Q_0, \mathit{program}_0)$ is an adversary for Q_0.

To that order, we first introduce an intermediate semantics for CryptoVerif that decomposes the evaluation of the function *simulate* into several small steps. We easily relate this semantics to the semantics of CryptoVerif. Next, in Section 6.4.2, we relate the intermediate semantics to the OCaml semantics. For this purpose, we introduce a relation between intermediate semantic configurations and OCaml traces, that, in particular, ensures that the events are the same on both sides and we prove that this relation is preserved by reduction. Finally, in Section 6.4.3, we use these results to prove our main theorem.

6.4.1 Intermediate Semantics

We introduce extended CryptoVerif configurations $\mathfrak{C}^{\mathsf{cs}}$, which are configurations of the form $\mathfrak{C}$ or $\mathfrak{C}, \mathit{steps}, \mathcal{CS}$, where $\mathcal{CS}$ is a simulator configuration and *steps* is the maximum number of reductions of $\mathcal{CS}$ that can still be performed. (We use the field *steps* to guarantee termination.) The configurations $\mathfrak{C}, \mathit{steps}, \mathcal{CS}$ serve to represent the state of the system during the evaluation of the function *simulate*. We define a reduction relation $\rightsquigarrow$ on the extended configurations $\mathfrak{C}^{\mathsf{cs}}$.

Definition 6.15 *Let us define the reduction relation $\rightsquigarrow$ such that:*

$$\frac{E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_p \mathfrak{C}' \quad P \text{ is not of the form } x[a] \leftarrow \mathit{simulate}(s[a]); P' \text{ for any } x, a, P'}{E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightsquigarrow_p \mathfrak{C}'} \quad \text{(CryptoVerif)}$$

$$\frac{E(s[a]) = \mathsf{repr}(\mathcal{CS})}{\begin{array}{l}E, x[a] \leftarrow \mathit{simulate}(s[a]); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightsquigarrow \\ E, x[a] \leftarrow \mathit{simulate}(s[a]); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, N_{\mathsf{steps}}, \mathcal{CS}\end{array}} \quad \text{(Enter Simulator)}$$

$$\frac{\mathcal{CS} \rightarrow \mathcal{CS}' \qquad \mathit{steps} > 0}{E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{steps}, \mathcal{CS} \rightsquigarrow E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{steps} - 1, \mathcal{CS}'} \quad \text{(Simulator)}$$

$$\frac{\mathcal{CS} \text{ does not reduce or } \mathit{steps} = 0}{\begin{array}{l}E, x[a] \leftarrow \mathit{simulate}(s[a]); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{steps}, \mathcal{CS} \rightsquigarrow \\ \quad E[x[a] \mapsto \mathit{simreturn}(\mathcal{CS})], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Leave Simulator)}$$

When encountering a configuration $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ such that P is of the form $x[a] \leftarrow \mathit{simulate}(s[a]); P'$ and $E(s[a]) = \mathsf{repr}(\mathcal{CS})$, we reduce $\mathfrak{C}$ into an extended configuration $\mathfrak{C}, N_{\mathsf{steps}}, \mathcal{CS}$ by (Enter Simulator). We reduce $\mathcal{CS}$ by (Simulator) until it blocks or the number of allowed reductions N_{steps} is exhausted, and then we resume the CryptoVerif reductions by (Leave Simulator).

In the next lemma and proposition, we relate traces using $\rightsquigarrow$ to traces using $\rightarrow$, to prove that all events have the same probability in these two semantics. These results are proved in Appendix E.

Lemma 6.16 *Let $\mathfrak{C}$ be a CryptoVerif configuration.*

- *If $\mathfrak{C} \rightarrow_p \mathfrak{C}'$, then there is a trace $\mathfrak{C} \rightsquigarrow_p^* \mathfrak{C}'$ and all intermediate configurations in this trace (if any) are of the form $\mathfrak{C}, \mathit{steps}, \mathcal{CS}$.*
- *If $\mathfrak{C}$ does not reduce by $\rightarrow$, then it does not reduce by $\rightsquigarrow$ either.*

We denote by $\Pr[\mathfrak{C}^{\mathsf{cs}}(\rightsquigarrow) : D]$ the probability of the set of complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathfrak{C}^{\mathsf{cs}}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = \text{true}$. The next proposition shows that all events have the same probability in the intermediate semantics as in the CryptoVerif semantics.

Proposition 6.17 $\Pr[\mathfrak{C}(\rightsquigarrow) : D] = \Pr[\mathfrak{C} : D]$.

6.4.2 Relation between the Intermediate Semantics and the OCaml Semantics

Let us first give some preliminary definitions.

Definition 6.18 (Accessible trace, configuration) *An* accessible *trace $\mathcal{CT}$ (for the adversary $\mathit{program}_0$) is a trace beginning with the initial configuration $\mathcal{C}_0(Q_0, \mathit{program}_0)$ defined in Section 5.*

A configuration $\mathcal{C}$ is accessible *if there exists an accessible trace such that its last configuration is $\mathcal{C}$.*

Definition 6.19 (Concretization of $\mathcal{I}$ and $\mathcal{RI}$) *Let us define the sets of oracles $\mathcal{O}^\infty(\mathcal{I})$ and $\mathcal{O}^\infty(\mathcal{RI})$ represented by $\mathcal{I}$ and $\mathcal{RI}$ respectively:*

$$\begin{aligned}
\mathcal{O}^\infty(\mathcal{I}) &= \{O[b, \widetilde{a'}] \mid O\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{I}, a \le b\} \cup \{O[\widetilde{a}] \mid O[\widetilde{a}] \in \mathcal{I}\} \\
\mathcal{O}^\infty(\mathcal{RI}) &= \{O[b, \widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[_, \widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}]), 1 \le b\} \\
&\quad \cup \{O[\widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[\widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])\} \\
&\quad \cup \{O[b, \widetilde{a'}] \mid \mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{RI}, \\
&\qquad\qquad O[b, \widetilde{a'}] \in \mathsf{oracles}'(Q(\mathsf{role})[b, \widetilde{a'}]), a \le b\}
\end{aligned}$$

The definition of $\mathcal{O}^\infty(\mathcal{I})$ and $\mathcal{O}^\infty(\mathcal{RI})$ ignores the replication bounds and allows the indices of oracles to go to infinity. Using unbounded indices is helpful in Properties 13 and 14 of Definition 6.27 below. By Assumption 2.10, when O is a first oracle of a role role under replication, O cannot be under replication in $Q(\mathsf{role})$. So the last component of $\mathcal{O}^\infty(\mathcal{RI})$ cannot contain oracles under replication.

Let us describe how the sets $\mathcal{I}$ and $\mathcal{RI}$ represent the contents of the set of callable processes $\mathcal{Q}$.

Definition 6.20 (Relation between $\mathcal{I}$, $\mathcal{RI}$ and $\mathcal{Q}$) *Let us define the sets of oracles $\mathcal{O}(\mathcal{I})$ and $\mathcal{O}(\mathcal{RI})$ represented by $\mathcal{I}$ and $\mathcal{RI}$ respectively:*

$$\begin{aligned}
\mathcal{O}(\mathcal{I}) &= \{O[b, \widetilde{a'}] \mid O\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{I}, a \le b \le N_O\} \cup \{O[\widetilde{a}] \mid O[\widetilde{a}] \in \mathcal{I}\} \\
\mathcal{O}(\mathcal{RI}) &= \{O[b, \widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, \\
&\qquad\qquad O[_, \widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}]), 1 \le b \le N_O\} \\
&\quad \cup \{O[\widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[\widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])\} \\
&\quad \cup \{O[b, \widetilde{a'}] \mid \mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{RI}, \\
&\qquad\qquad O[b, \widetilde{a'}] \in \mathsf{oracles}'(Q(\mathsf{role})[b, \widetilde{a'}]), a \le b \le N_{\mathsf{role}}\}
\end{aligned}$$

We write $\mathcal{Q} \leftrightarrow \mathcal{RI}, \mathcal{I}$ when the following two properties hold:

- *$\mathcal{Q}$ consists of exactly one element $O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P$ for each $O[\widetilde{a}]$ present in the set $\mathcal{O}(\mathcal{I}) \cup \mathcal{O}(\mathcal{RI})$. We denote by $\mathcal{Q}(O[\widetilde{a}])$ this element of $\mathcal{Q}$.*
- *If $O\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{I}$, then there exist a process Q and an index i such that i does not occur in $\mathsf{fv}(Q)$ and for all $b \in \{a, \ldots, N_O\}$, we have $\mathcal{Q}(O[b, \widetilde{a'}]) = Q\{b/i\}$.*

In contrast to the sets we defined in Definition 6.19, the indices of oracles in $\mathcal{Q}$ are bounded by the replication bounds. So we redefine sets of oracles $\mathcal{O}(\mathcal{RI})$ and $\mathcal{O}(\mathcal{I})$ that correspond to $\mathcal{RI}$ and $\mathcal{I}$, but with indices bounded by N_O and N_{role} as appropriate. The sets $\mathcal{O}(\mathcal{RI})$ and $\mathcal{O}(\mathcal{I})$ are included in $\mathcal{O}^\infty(\mathcal{RI})$ and $\mathcal{O}^\infty(\mathcal{I})$, respectively. The set of processes $\mathcal{Q}$ corresponds to $\mathcal{RI}, \mathcal{I}$ when it contains exactly one definition for each oracle in $\mathcal{O}(\mathcal{I}) \cup \mathcal{O}(\mathcal{RI})$. Furthermore, in case an oracle is under replication, the corresponding elements of $\mathcal{Q}$ all have the same form; they differ only by the value of the replication index. We enforce this property in the last item of Definition 6.20.

Next, we define several sets of oracles and roles, which allow us to determine which oracles and roles are in which state (callable immediately, available later) in a simulator configuration.

Definition 6.21 (Oracle sets) *Let* $\mathcal{O}_{\mathsf{call}}(Th)$ *be the set of oracles* $O[\widetilde{a}]$ *not under replication that occur in* call *constructs in the thread* Th*, without entering tagged functions and closures.*

Let $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$ *be the set of oracles* $O[a, \widetilde{a}]$ *such that* O *is under replication,* $a > N_O$*, and* $\mathsf{call}(O[\_, \widetilde{a}])$ *occurs in the thread* Th*, without entering tagged functions and closures.*

Let $\mathcal{R}_{\mathsf{init\text{-}closure}}(Th)$ *be the set of roles* $\mathsf{role}[\widetilde{a}]$ *such that there exists* env *such that a closure* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *is present in the thread* Th*, and such that* $env(token)$ *is bound in its store to* true*.*

Let $\mathcal{R}_{\mathsf{init\text{-}function}}(Th)$ *be the set of roles* $\mathsf{role}[\widetilde{a}]$ *such that the initialization function* $program'(\mathsf{role}[\widetilde{a}])$ *is present in the thread* Th*.*

Let $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$*,* $\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS})$*, and* $\mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS})$ *be the unions of the corresponding sets for all threads of the configuration.*

We have already defined the set $\mathsf{returnoracles}(O[\widetilde{a}])$*, which is the set of oracles returned by* $O[\widetilde{a}]$*. Let us define* $\mathsf{returnoracles}'$ *such that:*

$$\begin{aligned}\mathsf{returnoracles}'(O[\widetilde{a}]) = {} & \{O'[\widetilde{a'}] \mid O'[\widetilde{a'}] \in \mathsf{returnoracles}(O[\widetilde{a}])\} \\ & \cup \{O'[b, \widetilde{a'}] \mid O'[\_, \widetilde{a'}] \in \mathsf{returnoracles}(O[\widetilde{a}]), 1 \leq b \leq N_{O'}\}\end{aligned}$$

Let $\mathcal{CS} = \mathcal{C}, \mathcal{RI}, \mathcal{I}$*. We denote the callable set of oracles:*

$$\mathsf{callable}(\mathcal{CS}) = \mathcal{O}^{\infty}(\mathcal{I}) \cup \mathcal{O}^{\infty}(\mathcal{RI}) \cup \mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS}) \cup \mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS}))$$

Let $\mathsf{willbeavailable}(\mathcal{CS})$ *be the set of oracles that can eventually become available. This set is the least fixpoint of the function* f *defined by* $f(C) = C \cup \mathsf{returnoracles}'(C)$ *that contains the set* $\mathsf{returnoracles}'(\mathsf{callable}(\mathcal{CS}))$*.*

The definition of $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$ may be surprising, as it considers $O[a, \widetilde{a}]$ with a greater than the replication bound N_O. We have made this choice to guarantee that $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$ is always included in $\mathcal{O}^{\infty}(\mathcal{I})$: the indices up to N_O may have been consumed by calls already made to the oracle, while the indices greater than N_O always remain, because we make at most N_O calls to this oracle by definition of N_O. This property is exploited in Property 14 of Definition 6.27 below.

The sets $\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS})$ and $\mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS})$ are sets of roles with their replication indices, which can be seen as a role set $\mathcal{RI}$. The set $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS}) \cup \mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS}))$ is the set of the first oracles of roles present in $\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS})$ and $\mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS})$.

The following definitions present functions used to relate OCaml and simulator threads.

Definition 6.22 (Replace initialization) *The function* replaceinitpm *replaces in its argument the pattern matchings corresponding to role initialization of the simulator by the OCaml module initialization: to be more precise,*

$\mathsf{replaceinitpm}(Th)$ *replaces each occurrence of* $\mathsf{tagfunction}^{\mathsf{role}}\ pm'_{\mathsf{role}[\widetilde{a}]}$ *in* Th *with* $\mathsf{tagfunction}^{\mathsf{role}}\ pm_{\mu_{\mathsf{role}}}$ *and each occurrence of* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *in* Th *with* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm_{\mu_{\mathsf{role}}}]$.

This function transforms every occurrence of the tagged closures corresponding to role initialization in the simulator, which are added by the addthread construct, into the corresponding tagged closures in OCaml.

Definition 6.23 (Correct closure) *Suppose that* $\mathcal{Q} \leftrightarrow \mathcal{RI}, \mathcal{I}$ *for some* $\mathcal{RI}$, l_{tok} *is a function that maps each oracle* $O[\widetilde{a}]$ *to the location of its token, and* τ_{O} *is a function maps each oracle* $O[\_, \widetilde{a}]$ *to the tag* τ *of the corresponding closure. We define the set of closures that correspond to an oracle:*

- *for an oracle* $O[\widetilde{a}] \in \mathcal{I}$*:*

$$\begin{aligned}&\mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \\ &\quad \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{false}}(\mathcal{Q}(O[\widetilde{a}]))] \mid \\ &\quad\ env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}(O[\widetilde{a}])), env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\}\end{aligned}$$

- *for an oracle* $O[\widetilde{a}] \notin \mathcal{I}$*:*

$$\begin{aligned}&\mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \\ &\quad \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{false}}(Q)] \mid \textit{for any } Q, env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\}\end{aligned}$$

- *for an oracle* $O\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$ *with* $a' \leq N_O$*,*

$$\begin{aligned}&\mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \\ &\quad \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(\mathcal{Q}(O[a', \widetilde{a''}]))] \mid \\ &\quad\ \tau = \tau_{\mathsf{O}}(O[\_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}(O[a', \widetilde{a''}]))\}\end{aligned}$$

- *for an oracle* $O\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$ *with* $a' > N_O$*,*

$$\begin{aligned}&\mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \\ &\quad \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(Q)] \mid \tau = \tau_{\mathsf{O}}(O[\_, \widetilde{a''}]), \textit{for any } Q, env\}\end{aligned}$$

- *for an oracle* $O\big[[a', +\infty[, \widetilde{a''}\big] \notin \mathcal{I}$*:*

$$\mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \emptyset$$

The function correctclosure serves to map calls $\mathsf{call}(R)$ in the simulator configuration into their corresponding closures in the OCaml configuration: $\mathsf{call}(R)$ will be mapped below to an element of $\mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

In the case $O[\widetilde{a}] \in \mathcal{I}$, we map $\mathsf{call}(O[\widetilde{a}])$ into the closure that translates the process $\mathcal{Q}(O[\widetilde{a}])$.

The case $O[\widetilde{a}] \notin \mathcal{I}$ may be used when the oracle $O[\widetilde{a}]$ has been called but the thread still contains a call to this oracle. If the oracle is called again, the call will fail. The process $\mathcal{Q}(O[\widetilde{a}])$ is removed from $\mathcal{Q}$ after execution, so we do not know which process to translate to obtain the correct closure for $O[\widetilde{a}]$, that is why the correct closures for a call to an already called oracle can contain the translation of any process Q. This translation will fail and raise the exception Bad_Call regardless of the translated process Q.

Oracles under replication cannot disappear from $\mathcal{I}$ after having been added to it: when one calls the oracle $O[\_, \widetilde{a''}]$, we just increment the counter a' of the element $O\big[[a', +\infty[, \widetilde{a''}\big]$ present in $\mathcal{I}$. We need to distinguish whether the adversary has exhausted all the N_O calls available for this oracle or not. If there remains available calls, the process $\mathcal{Q}(O[a', \widetilde{a''}])$ is defined, and we require that $\mathsf{call}(O[\_, \widetilde{a''}])$ is mapped into a closure that translates this process. Otherwise, if all the calls are exhausted, $a' > N_O$, and $\mathcal{Q}(O[a', \widetilde{a''}])$ is not defined, but we know that the adversary will not call the oracle again, so $\mathsf{call}(O[\_, \widetilde{a''}])$ can be mapped to closures that translate any process.

The case $O\big[[a', +\infty[, \widetilde{a''}\big] \notin \mathcal{I}$ never happens: it would mean that the oracle $O[\_, \widetilde{a''}]$ can be called but there is no reference to it in the set $\mathcal{I}$.

Definition 6.24 (Replace call)

$\mathsf{replacecalls}(\langle env, pe, stack, store\rangle, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) =$
$\{\langle env', \sigma(pe), \sigma(stack), \sigma(store)\rangle \mid$ *if* pe *is a value* v *or an exceptional value* raise v*, then* env' *is any environment, else* $env' = \sigma(env)$*, where* σ *is a function that replaces, for each* R*, each occurrence of* $\mathsf{call}(R)$ *with an element of* $\mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\}$

The function replacecalls replaces in its argument each call $\mathsf{call}(O[\widetilde{a}])$ with a closure that corresponds to the oracle $O[\widetilde{a}]$, computed by correctclosure. It allows any environment when the current program or expression is a value or an exceptional value, because in these cases, the environment is not used.

Definition 6.25 (Token part of the store)

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

The function gettokens returns the part of the store corresponding to the tokens of the closures of oracles not under replication.

Definition 6.26 (Processes) *We use the following notations:*

P_{loop} *is the process from line 7 to line 20 in Figure 19.*

$Q_{loop} \stackrel{\text{def}}{=} O_{\text{loop}}[i'](s : T_{\mathcal{CS}}) := P_{loop}$.

$P_{return\text{-}loop}(\alpha) \stackrel{\text{def}}{=}$ if $b_{\alpha,r}[\,]$ then

let $r[\,] : T_{\mathcal{CS}} =$ loop $O_{loop}[\alpha+1](r'_{\alpha,r}[\,])$ in end else end

else $r[\,] \leftarrow r'_{\alpha,r}[\,]$; end .

$\mathcal{R}_{loop}(\alpha) \stackrel{\text{def}}{=} [((r'_{\alpha,r}[\,], b_{\alpha,r}[\,]), P_{return\text{-}loop}(\alpha), \mathsf{end}), (x[\,], \mathsf{return}(x[\,]), \mathsf{end})]$.

Definition 6.27 (Relation between extended CryptoVerif configurations and OCaml traces) *Let $\mathfrak{C}^{\mathsf{cs}}$ be an extended CryptoVerif configuration and $\mathcal{CT}$ be an accessible OCaml trace. We say that $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ when there exists an injective function τ_{O} that maps oracles $O[\_, \widetilde{a}]$ such that $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$ for some a' to tags τ, such that the following properties are all true:*

1. $\mathfrak{C}^{\mathsf{cs}} = E, P_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E}, steps, \mathcal{CS}$.
   $\mathcal{CS} = ([Th_1, \ldots, Th_n], globalstore, tj), \mathcal{RI}, \mathcal{I}$.
   For all $i \leq n$, $Th_i = \langle env_i, pe_i, stack_i, store_i\rangle$.
2. $\mathcal{C}$ *is the last configuration of* $\mathcal{CT}$.
   $\mathcal{C} = [Th'_1, \ldots, Th'_n], globalstore', tj, \mathcal{MI}, events$.
   For all $i \leq n$, $Th'_i = \langle env'_i, pe'_i, stack'_i, store'_i\rangle$.
3. $\mathcal{Q} = \{Q_{loop}\{a/i'\} \mid \alpha < a \leq N_{\mathsf{rand+calls}}\} \cup \mathcal{Q}_0$ *and* $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}, \mathcal{I}$.
4. $\mathsf{fv}(P_{loop}\{\alpha/i'\}) \cup \mathsf{fv}(\mathcal{Q}) \cup \mathsf{fv}(\mathcal{R}_{loop}(\alpha)) \subseteq \mathrm{Dom}(E)$.
5. *For* $i \leq n$, *all store locations in* S_l *present in* Th_i *are in* $\mathrm{Dom}(store_i)$.
6. *For each thread* $i \leq n$, *one of the following two cases occurs:*
   (a) $Th'_i = \mathsf{replaceinitpm}(Th_i)$.
   $Th_i = \langle \emptyset, program_{\mathsf{prim}};; program'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots;; program'(\mathsf{role}_l[\widetilde{a_l}]);; program', [\,], \emptyset\rangle$.
   There is no closure, no tagged function $\mathsf{tagfunction}^t$ *pm, no* event, *and no* return *in program', except in program*(μ_{role}) *in arguments of* addthread.
   (b) *The following properties hold:*
      i. *There exist* $store''_i$ *and an injective function* l_{tok} *that associates to each* $O[\widetilde{a}]$ *in* $\mathcal{O}_{\mathsf{call}}(Th_i)$ *a store location that does not occur in* $\mathcal{CS}$ *such that*

$$\langle env'_i, pe'_i, stack'_i, store''_i\rangle \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th_i), \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}}),$$
$$store''_i \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th_i), l_{\mathsf{tok}}) \subseteq store'_i .$$

ii. *There exists an injective function* $l_{\mathsf{init\text{-}tok}}$ *that associates to each role* $\mathsf{role}[\widetilde{a}]$ *such that a closure* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *occurs in the thread* Th_i *for some* env *and* τ*, a store location such that for all closures* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *present in* Th_i*, we have* $l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}]) = env(token)$.
The locations $l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}])$ *and* $l_{\mathsf{tok}}(O[\widetilde{a'}])$ *are distinct for every role* $\mathsf{role}[\widetilde{a}]$ *and oracle* $O[\widetilde{a'}]$.
The locations $l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}])$ *occur only in* $\mathrm{Dom}(store_i)$ *and in* $env(token)$ *where* env *is the environment of a tagged closure* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *in* Th_i.

iii. *For each tagged closure* $\mathsf{tagfunction}^{t,\tau}[env, pm]$ *present in* Th_i*, the tag* t *is a role* role*,* $env_{\mathsf{prim}} \subseteq env$*, and there exist indices* $\widetilde{a}$ *such that* $pm = pm'_{\mathsf{role}[\widetilde{a}]}$.

iv. *There is no tagged function* $\mathsf{tagfunction}^t$ pm*, no* event*, and no* return *in* Th_i *except in* $program(\mu_{\mathsf{role}})$ *in arguments of* addthread.

7. *For all locations* $l \in S_{priv}$*,* l *does not occur in* $Th_1, \ldots, Th_n$ *except in* $program(\mu_{\mathsf{role}})$ *in arguments of* addthread.

8. $\forall l \in S_{priv}, globalstore(l) = initval_l$.

9. $globalstore(E, \mathcal{T}) \subseteq globalstore'$.

10. $\forall l \notin S_{priv}, globalstore(l) = globalstore'(l)$.

11. $\mathcal{MI} = \{(\mu_{\mathsf{role}}, \mathrm{false}) \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}\} \cup \{(\mu_{\mathsf{role}}, \mathrm{true}) \mid \mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{RI}\}$.

12. $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$.

13. *The sets* $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$*,* $\mathcal{O}^\infty(\mathcal{RI})$*, and* $\mathsf{willbeavailable}(\mathcal{CS})$ *are pairwise disjoint.*

14. *The* $4n$ *sets of oracles* $\mathcal{O}_{\mathsf{call}}(Th_i)$*,* $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th_i)$*,* $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th_i))$*, and* $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th_i))$ *for* $i \leq n$ *are pairwise disjoint, and are all included in* $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$.

15. $|\mathcal{CT}| + steps \geq N_{\mathsf{steps}}$.

16. $\alpha \leq N_{\mathsf{rand}}(\mathcal{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT}) + 1$.

17. *If* $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$*, then* $a' \leq N_{\mathsf{calls}}(O, \tau_{\mathsf{O}}(O[\_, \widetilde{a}]), \mathcal{CT}) + 1$.

18. *If* $\mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{RI}$*, then* $a' \leq N_{\mathsf{exec}}(\mathsf{role}, \mathcal{CT}) + 1$.

The relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is our main tool to relate the CryptoVerif and OCaml systems. This relation holds only when the CryptoVerif adversary is evaluating the function *simulate* (line 7 of Figure 19), as shown by the form of the extended CryptoVerif configuration $\mathfrak{C}^{\mathsf{cs}}$ in Item 1. (The value α is the current value of

the index i', that is, the number of iterations in the loop.) Items 1 and 2 also ensure that there is the same number of threads in the simulator configuration $\mathcal{CS}$ and in the OCaml configuration $\mathcal{C}$.

Item 3 is an invariant on the CryptoVerif side: it relates the available oracles in $\mathcal{Q}$ to elements of the simulator configuration. This item ensures basically that when the simulator calls an oracle present in $\mathcal{I}$, it is also present in $\mathcal{Q}$, and the oracle call in the CryptoVerif adversary (line 13 of Figure 19) can proceed. Item 4 is an invariant of the CryptoVerif semantics: the environment contains bindings for every free variable present in the current configuration. Item 5 is an invariant of the simulator: each store location that occurs in a thread is present in the domain of the store. (When a location is created, it is immediately added to the store.)

Item 6 relates the threads of the simulator and of the OCaml semantics. A thread can be in one of the following two states. If it satisfies Item 6(a), the thread is a protocol thread that was not scheduled yet. The simulator and OCaml threads correspond by transforming the program $\mathit{program}'(\mathsf{role}[\widetilde{a}])$ present in the simulator into the program of the module corresponding to the role, $\mathit{program}(\mu_{\mathsf{role}})$. Otherwise, the thread satisfies Item 6(b). In this case, Item 6(b)i relates the contents of the simulator thread and the OCaml thread by replacing $\mathit{program}'(\mathsf{role}[\widetilde{a}])$ with $\mathit{program}(\mu_{\mathsf{role}})$ as above, and by replacing calls to oracles using call with a corresponding tagged closure. 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(b)ii ensures that all instances of a closure of a given role initialization $\mathsf{role}[\widetilde{a}]$ share the same store location for their tokens. This ensures that a role initialization closure is not called twice. Item 6(b)ii also ensures that all locations used for the tokens of role initialization are not accessible elsewhere. Item 6(b)iii ensures that every tagged closure present in the simulator is a correct closure for the initialization of a role. Item 6(b)iv is an invariant of the simulator that ensures that the adversary does not have access to our OCaml instrumentation features.

Items 7 to 10 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 10). The private part (files and tables) is empty in the simulator, since this part is handled by CryptoVerif itself (Item 8) and cannot be accessed by the adversary (Item 7). We require that the private part of the OCaml global store corresponds to the CryptoVerif configuration (Item 9).

Item 11 relates the OCaml multiset of callable modules $\mathcal{MI}$ and the simulator set of callable roles $\mathcal{RI}$. Item 12 relates the OCaml and CryptoVerif events.

Items 13 and 14 present restrictions on sets of oracles. To understand how all these oracle sets interact, let us present the flow of an oracle not under replication $O[\widetilde{a}]$ in these sets.

1. Initially, if the oracle occurs at the beginning of the process, it is in $\mathcal{O}^{\infty}(\mathcal{RI})$; otherwise, it is in $\mathsf{willbeavailable}(\mathcal{CS})$.

2. For an oracle occurring at the beginning of a role, when the role containing it is instantiated using addthread, the oracle moves from $\mathcal{O}^\infty(\mathcal{RI})$ to $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th))$. It is also added into $\mathcal{O}^\infty(\mathcal{I})$.

3. When the initialization function of the role is reduced into a closure, the oracle moves from $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th))$ to $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th))$.

4. When the initialization function of the role is called, the oracle moves from $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th))$ to $\mathcal{O}_{\mathsf{call}}(Th)$.

5. When the oracle itself is called, it is removed from $\mathcal{O}^\infty(\mathcal{I})$, and when the call to the oracle disappears from the thread, it is removed from $\mathcal{O}_{\mathsf{call}}(Th)$. The oracles made available after the call are removed from willbeavailable($\mathcal{CS}$) and added either to $\mathcal{O}^\infty(\mathcal{RI})$ if they start a role or to $\mathcal{O}^\infty(\mathcal{I})$ and $\mathcal{O}_{\mathsf{call}}(Th)$ if they do not start a role.

The case of an oracle under replication is fairly similar, using $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$ instead of $\mathcal{O}_{\mathsf{call}}(Th)$. Items 13 and 14 ensure that an oracle cannot be simultaneously in two different sets. These properties allow us to prove that the injections of Items 6(b)i and 6(b)ii are kept. (We use $\mathcal{O}^\infty$ rather than $\mathcal{O}$ to make sure that we cannot have simultaneously $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$ with $a' > N_O$ and $O[b, \widetilde{a}] \in$ willbeavailable($\mathcal{CS}$) for all b. Indeed, $\mathcal{O}(\{O\big[[a', +\infty[, \widetilde{a}\big]\})$ is empty when $a' > N_O$, so this situation would not be prevented by Item 13 if it used $\mathcal{O}$. It is prevented using $\mathcal{O}^\infty$.)

Items 15 to 18 ensure that we never reach the limits on the number of simulator steps N_{steps} (Item 15), the number of calls to the oracles (Item 16 for the oracle O_{loop} and Item 17 for the other oracles), and the number of calls to roles (Item 18), by making sure that the number of calls on the CryptoVerif side is at most the number of calls on the OCaml side. The number of calls made to oracle $O[\_, \widetilde{a}]$ in CryptoVerif, $a' - 1$ such that $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$, may be less than the number of calls to that oracle in the OCaml trace, $N_{\mathsf{calls}}(O, \tau_{\mathsf{O}}(O[\_, \widetilde{a}]), \mathcal{CT})$, because failed calls are not counted on the CryptoVerif side.

The next two lemmas show that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved during execution. Lemma 6.28 shows that it holds at the beginning, as soon as the simulator reaches line 7 of Figure 19.

Lemma 6.28 *There exists a trace $\mathfrak{C}_0(Q_0, program_0) \rightsquigarrow^* \mathfrak{C}^{\mathsf{cs}}$ where $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}_0$ and $\mathcal{CT}_0 = \mathcal{C}_0(Q_0, program_0)$.*

Lemma 6.29 shows that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved. More precisely, the relation does not hold at all steps (in particular because it holds only when the CryptoVerif adversary is executing *simulate*), but if it holds at some point, we can continue execution so that either it holds again at a later point, or execution ends with matching events.

Lemma 6.29 *Let $\mathfrak{C}^{\mathsf{cs}}$ such that there exists an accessible trace $\mathcal{CT}$ satisfying $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.*

- *Either there exist n configurations $\mathfrak{C}_1^{\mathsf{cs}}, \ldots, \mathfrak{C}_n^{\mathsf{cs}}$ and n traces $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow_{p_1}^+ \mathfrak{C}_1^{\mathsf{cs}}$, $\ldots, \mathfrak{C}^{\mathsf{cs}} \rightsquigarrow_{p_n}^+ \mathfrak{C}_n^{\mathsf{cs}}$ such that none of these traces is a prefix of another, $\sum_{i \leq n} p_i = 1$, and for each accessible trace $\mathcal{CT}$ such that $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$, there exist n pairwise disjoint trace sets $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ such that all traces in these sets are extensions of $\mathcal{CT}$, none of these traces is a prefix of another, $\Pr[\mathcal{CTS}_i] = p_i \cdot \Pr[\mathcal{CT}]$, and for each trace $\mathcal{CT}' \in \mathcal{CTS}_i$, we have $\mathfrak{C}_i^{\mathsf{cs}} \equiv \mathcal{CT}'$.*

- *Or for each accessible trace $\mathcal{CT}$ such that $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$, the last configuration $\mathcal{C}$ of $\mathcal{CT}$ cannot reduce, $\mathfrak{C}^{\mathsf{cs}} \rightarrow^+ \mathfrak{C}_1^{\mathsf{cs}}$, the configuration $\mathfrak{C}_1^{\mathsf{cs}}$ cannot reduce, and the event list $\mathcal{E}$ of $\mathfrak{C}_1^{\mathsf{cs}}$ and the event list events of $\mathcal{C}$ satisfy events $= \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$.*

We prove these lemmas in Appendix F. Let us present a proof sketch of Lemma 6.29.

Proof sketch Let us take an extended CryptoVerif configuration $\mathfrak{C}^{\mathsf{cs}}$ and an OCaml trace $\mathcal{CT}$ such that $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. Let $\mathcal{C}$ be the last configuration of $\mathcal{CT}$. Let $\mathcal{CS}$ be the configuration of the simulator in $\mathfrak{C}^{\mathsf{cs}}$ and Th be the current thread of $\mathcal{CS}$.

Case 1: the current thread of $\mathcal{CS}$ verifies Item 6(a), we run the initialization of the module. The programs of the current threads of $\mathcal{CS}$ and $\mathcal{C}$ are the same except that $\mathit{program}'(\mathsf{role}[\widetilde{a}])$ present in $\mathcal{CS}$ are transformed into $\mathit{program}(\mu_{\mathsf{role}})$. We show that after having reduced the initialization of the primitives and the initialization of the roles in both sides, the current threads verify Item 6(b). The oracles in $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th))$ that correspond to the roles implemented in this initialization are moved to $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th))$. We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

Case 2: the current thread of $\mathcal{CS}$ verifies Item 6(b). We distinguish cases on the form of the simulator configuration $\mathcal{CS}$.

Let us first look at the cases in which the configuration $\mathcal{CS}$ does not reduce. We use the rule (Leave Simulator), thus finishing the evaluation of the function *simulate*.

- If the current expression of $\mathcal{CS}$ is $\mathsf{call}(O_j[\widetilde{a}])\ v$, then the result of *simulate* is such that $o = o_j$, so the CryptoVerif adversary of Figure 19 calls the oracle O_j at line 13 in the branch $o = o_j$, ends one iteration of O_{loop}, and starts the next iteration until it reaches line 7. We use Lemma 6.10 and we exploit the definition of $\mathit{simulate}'_{O_j}$ and $\mathit{simulate}''_{O_j}$ to prove that the OCaml configuration reduces similarly, by calling the OCaml function generated for oracle O_j. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}^\infty(\mathcal{I})$, and from $\mathcal{O}_{\mathsf{call}}(Th)$ if all occurrences of $\mathsf{call}(O_j[\widetilde{a}])$ have disappeared. The newly available oracles, added to sets $\mathcal{O}^\infty(\mathcal{RI})$ or $\mathcal{O}_{\mathsf{call}}(Th)$ and $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$, are removed from the set $\mathsf{willbeavailable}(\mathcal{CS})$. We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

- If the current expression of $\mathcal{CS}$ is $\mathsf{random}\ ()$, then the result of *simulate* is such that $o = o_R$, so the CryptoVerif adversary of Figure 19 samples a random boolean at line 19, ends one iteration of O_{loop}, and starts the next

iteration until it reaches line 7. The current expression of $\mathcal{CS}$ is replaced with true with probability 1/2 and false with probability 1/2. The OCaml configuration reduces similarly: it samples a random boolean by using the rule (Random), and the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

- Otherwise, the configuration $\mathcal{CS}$ cannot reduce, and the corresponding configuration $\mathcal{C}$ cannot reduce either. The result of *simulate* is such that $o = o_S$, so the CryptoVerif adversary of Figure 19 ends the current iteration of O_{loop} at line 9, and ends the loop at line 4, so it also stops. The events in the final CryptoVerif and OCaml configurations match, so the second case of the lemma holds.

If the current expression of $\mathcal{CS}$ is $\mathsf{addthread}(\mathit{program})$, a new thread is created on both sides. If *program* is a protocol program, then this new thread satisfies Item 6(a) by definition of addthread in OCaml and in the simulator and by definition of replaceinitpm. The roles added in this new thread Th are removed from $\mathcal{RI}$ and the corresponding oracles are added to $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}function}}(Th))$ and to $\mathcal{I}$. Otherwise, the new thread satisfies Item 6(b). We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

If the current expression of $\mathcal{CS}$ is $\mathsf{call}(O_j[\widetilde{a}])\ v$ and $\mathcal{CS}$ reduces by (FailedCall1) or (FailedCall2), then the simulator raises Bad_Call, and the corresponding tagged function in OCaml also raises Bad_Call (because the tokens in OCaml correspond to $\mathcal{I}$ in the simulator by Item 6(b)i). We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

If the current expression of $\mathcal{CS}$ is $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]\ ()$, then we execute the initialization function of role role. We remove this role from $\mathcal{R}_{\mathsf{init\text{-}closure}}(Th)$, and add the corresponding oracles to $\mathcal{O}_{\mathsf{call}}(Th)$ and $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$. We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

The other cases are straightforward since the simulator mimics the OCaml semantics. They all preserve the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. □

From Lemmas 6.28 and 6.29, we can prove the following proposition, by extending the traces using Lemma 6.29 until we get complete traces.

Proposition 6.30 *Let $\mathfrak{CT}_1, \ldots, \mathfrak{CT}_n$ be complete CryptoVerif traces starting at $\mathfrak{C}_0(Q_0, pe_0)$.*

Then there exist disjoint sets of complete OCaml traces $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ 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 the event list $\mathcal{E}$ of $\mathfrak{C}$ and the event list events of $\mathcal{C}$ satisfy events $= \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$.

We prove this proposition in more detail in Appendix G. As an immediate consequence of this proposition, we obtain:

Proposition 6.31 $\Pr[\mathfrak{C}_0(Q_0, \mathit{program}_0)(\rightsquigarrow) : D] = \Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D]$.

6.4.3 Security Result

By combining Propositions 6.17 and 6.31, we obtain the following theorem:

Theorem 6.32 (Security result)

$$\Pr[\mathfrak{C}_0(Q_0, \mathit{program}_0) : D] = \Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D] \,.$$

In other words, the adversary $\mathit{program}_0$ against our generated OCaml modules has the same probability of breaking the security property as the adversary $Q_{\mathrm{adv}}(Q_0, \mathit{program}_0)$ against the CryptoVerif process.

CryptoVerif bounds the probability that an adversary Q breaks the security property D, that is, it finds a probability p that depends on the adversary such that, for all CryptoVerif adversaries Q for Q_0,

$$\Pr[\mathfrak{C}_{\mathrm{i}}(Q_0 \mid Q) : D] \leq p \,.$$

The adversaries $Q_{\mathrm{adv}}(Q_0, \mathit{program}_0)$ are CryptoVerif adversaries for Q_0, so for all OCaml programs *program* that obey our assumptions,

$$\Pr[\mathcal{C}_0(Q_0, \mathit{program}) : D] = \Pr[\mathfrak{C}_0(Q_0, \mathit{program}) : D] \leq p$$

Hence, all considered OCaml adversaries *program* can break the security property D with probability at most p.

The probability bound p returned by CryptoVerif is a function that depends on many parameters, expressed on the CryptoVerif protocol specification. Let us relate these parameters to the OCaml implementation. These parameters are as follows:

- The maximum number of times the various oracles and roles have been called, N_O and N_{role}. As shown by our proof and by Definition 6.11, N_O can be set to the maximum number of calls to the same closure representing oracle O in any trace of the OCaml program, and N_{role} can be set to the maximum number of instantiations of the role role in any trace of this program.

- The size of the CryptoVerif types T. The corresponding OCaml type $\mathbb{G}_T(T)$ is fixed by the annotations of the CryptoVerif specification. The size of T can be set to the size of $\mathbb{G}_T(T)$. Similarly, the size of the CryptoVerif values a (used when their type T has unbounded size) can be set to the size of the corresponding OCaml value $\mathbb{G}_{\mathsf{val}T}(a)$.

- The execution time of the cryptographic primitives and of various CryptoVerif constructs. This time can be set to the execution time of the corresponding OCaml implementation.

- The execution time of the adversary. Our proof shows that the function *simulate* executes at most as many reduction steps as the OCaml adversary. However, the CryptoVerif adversary shown in Figure 19 also includes

additional steps and conversions between the OCaml semantic configuration and its CryptoVerif bitstring representation. By using the contents of the OCaml memory as bitstring representation of the semantic configuration in CryptoVerif, we can obtain an efficient implementation of the CryptoVerif adversary that does not take significantly more time than the OCaml adversary.

From the probability bound given by CryptoVerif, we can then obtain a bound on the probability of breaking the security properties in the generated OCaml implementation of the protocol.

As detailed in [8], 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 6.32, our generated implementation of this protocol satisfies the same properties, provided assumptions A1 to A6 hold.

7 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] M. Aizatulin, A. D. Gordon, and J. Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In *CCS'11*, pages 331–340, New York, 2011. ACM.

[2] M. Aizatulin, A. D. Gordon, and J. Jürjens. Computational verification of C protocol implementations by symbolic execution. In *CCS'12*, pages 712–723, New York, 2012. ACM.

[3] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM TOPLAS*, 33(2), 2011.

[4] K. Bhargavan, C. Fournet, A. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM TOPLAS*, 31(1), 2008.

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

[6] B. Blanchet. Automatically verified mechanized proof of one-encryption key exchange. In *CSF'12*, pages 325–339, Los Alamitos, 2012. IEEE.

[7] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *CRYPTO'06*, volume 4117 of *LNCS*, pages 537–554. Springer, 2006.

[8] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations and application to SSH. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):4–31, Mar. 2013.

[9] D. Cadé and B. Blanchet. Proved generation of implementations from computationally-secure protocol specifications. In D. Basin and J. Mitchell, editors, *2nd Conference on Principles of Security and Trust (POST 2013)*, volume 7796 of *LNCS*, pages 63–82, Rome, Italy, Mar. 2013. Springer.

[10] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *CSF'09*, pages 172–185, Los Alamitos, 2009. IEEE.

[11] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *CSF'11*, pages 3–17, Los Alamitos, 2011. IEEE.

[12] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *CCS'11*, pages 341–350, New York, 2011. ACM.

[13] `http://msr-inria.inria.fr/projects/sec/fs2cv/`.

[14] G. Milicia. χ-spaces: Programming security protocols. In *NWPT'02*, 2002.

[15] S. Owens. A sound semantics for OCaml light. In S. Drossopoulou, editor, *ESOP'08*, volume 4960 of *LNCS*, pages 1–15, Heidelberg, 2008. Springer.

[16] A. Pironti and R. Sisto. Provably correct Java implementations of spi calculus security protocols specifications. *Computers and Security*, 29(3):302–314, 2010.

[17] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bharagavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP'11*, pages 266–278, New York, 2011. ACM.

Appendices

Note to the reviewers: these appendices provide detailed proofs of all results of the paper. Since the paper is very long, we are open to publishing these proofs as a technical report instead of including them in the journal paper. They are provided here so that the reviewers can check our results.

A Proof of Lemma 2.5

Proof (of Lemma 2.5) As usual, a multiset S is defined as a function from elements to integers: $S(x)$ is the number of occurrences of x in the multiset S. Multiset union is defined as addition: $(S \uplus S')(x) = S(x) + S'(x)$. The maximum $\max(S, S')$ is the multiset such that $\max(S, S')(x) = \max(S(x), S'(x))$. The inclusion $S \subseteq S'$ is true when $\mathrm{Dom}(S) \subseteq \mathrm{Dom}(S')$ and $\forall x \in \mathrm{Dom}(S), S(x) \leq S'(x)$.

We define the multiset of available oracles inductively as follows:

$$\begin{aligned}
&\mathit{Oracles}(0) = \emptyset\\
&\mathit{Oracles}(Q_1 \mid Q_2) = \mathit{Oracles}(Q_1) \uplus \mathit{Oracles}(Q_2)\\
&\mathit{Oracles}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) = \biguplus_{a \in [1,n]} \mathit{Oracles}(Q\{a/i\})\\
&\mathit{Oracles}(O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P) = \{O[\widetilde{a}]\} \uplus \mathit{Oracles}(P)\\
&\mathit{Oracles}(\mathsf{return}(M_1, \ldots, M_k); Q) = \mathit{Oracles}(Q)\\
&\mathit{Oracles}(\mathsf{end}) = \emptyset\\
&\mathit{Oracles}(x[\widetilde{a}] \stackrel{R}{\leftarrow} T; P) = \mathit{Oracles}(P)\\
&\mathit{Oracles}(x[\widetilde{a}] \leftarrow M; P) = \mathit{Oracles}(P)\\
&\mathit{Oracles}(\mathsf{insert}\ \mathit{Tbl}(M_1, \ldots, M_l); P) = \mathit{Oracles}(P)\\
&\mathit{Oracles}(\mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{a}], \ldots, x_l[\widetilde{a}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P') =\\
&\qquad \max(\mathit{Oracles}(P), \mathit{Oracles}(P'))\\
&\mathit{Oracles}(\mathsf{event}\ e(M_1, \ldots, M_l); P) = \mathit{Oracles}(P)\\
&\mathit{Oracles}(\mathsf{let}\ (x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) = O[\widetilde{M}](\widetilde{M'})\ \mathsf{in}\ P\ \mathsf{else}\ P') =\\
&\qquad \max(\mathit{Oracles}(P), \mathit{Oracles}(P'))\\
&\mathit{Oracles}(\mathsf{let}\ x[\widetilde{i}] : T = \mathsf{loop}\ O[\widetilde{M}](M')\ \mathsf{in}\ P\ \mathsf{else}\ P') =\\
&\qquad \max(\mathit{Oracles}(P), \mathit{Oracles}(P'))\\
&\mathit{Oracles}(E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}) = \mathit{Oracles}(P) \uplus \biguplus_{Q \in \mathcal{Q}} \mathit{Oracles}(Q) \uplus\\
&\qquad\qquad \biguplus_{((x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), P', P'') \in \mathcal{R}} \max(\mathit{Oracles}(P'), \mathit{Oracles}(P''))
\end{aligned}$$

We show that, for all configurations $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ reachable from the initial configuration $\mathfrak{C}_{\mathrm{i}}(Q_0)$, $\mathit{Oracles}(\mathfrak{C})$ contains no duplicates.

Let us first show this property for the initial configuration. We show by an easy induction on Q that $\biguplus_{Q' \in \mathrm{reduce}(Q)} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q)$. Therefore, by definition of $\mathfrak{C}_\mathrm{i}$, we have $\mathit{Oracles}(\mathfrak{C}_\mathrm{i}(Q_0)) = \biguplus_{Q' \in \mathrm{reduce}(Q_0)} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q_0)$. Next, we show by induction on Q_0 that $\mathit{Oracles}(Q_0)$ contains no duplicates.

- In the case $Q \mid Q'$, the oracles defined in Q and Q' are not in different branches of if or get, so by Property 2.3, they have different names. Hence, $\mathit{Oracles}(Q)$ and $\mathit{Oracles}(Q')$ do not both contain $O[\widetilde{a}]$ for the same O. We conclude that $\mathit{Oracles}(Q) \uplus \mathit{Oracles}(Q')$ contains no duplicates using the induction hypothesis.

- In the case $\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q$, by Property 2.3, the replication index i occurs as index in all definitions of oracles in Q, and in the same position. So the multisets $\mathit{Oracles}(Q\{a/i\})$ are disjoint for different choices of a. We conclude that $\biguplus_{a \in [1,n]} \mathit{Oracles}(Q\{a/i\})$ contains no duplicates using the induction hypothesis.

- In the case $O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P$, the definition of O is not in a branch of if or get different from P, so by Property 2.3, there is no definition of O in P. Hence $O[\widetilde{a}] \notin \mathit{Oracles}(P)$. We conclude that $\{O[\widetilde{a}]\} \uplus \mathit{Oracles}(P)$ contains no duplicates using the induction hypothesis.

- In all other cases, the result follows immediately from the induction hypothesis.

Furthermore, $\mathit{Oracles}(\mathfrak{C})$ decreases by reduction: if $\mathfrak{C} \to_p \mathfrak{C}'$, then we have $\mathit{Oracles}(\mathfrak{C}') \subseteq \mathit{Oracles}(\mathfrak{C})$. Indeed, the rules (New), (Let), (Insert), (Event), (Loop1) leave $\mathit{Oracles}(\mathfrak{C})$ unchanged. In the case of (Loop1), we use

$$\begin{aligned}&\max(\max(\max(\mathit{Oracles}(P), \mathit{Oracles}(P')), \mathit{Oracles}(P)), \mathit{Oracles}(P')) = \\ &\quad \max(\mathit{Oracles}(P), \mathit{Oracles}(P'))\,.\end{aligned}$$

The rules (If1), (If2), (Get1), (Get2), (Loop2), (Return), (End) decrease the multiset $\mathit{Oracles}(\mathfrak{C})$ by replacing $\max(\mathit{Oracles}(P), \mathit{Oracles}(P'))$ with either $\mathit{Oracles}(P)$ or $\mathit{Oracles}(P')$. In the case of (Return), we also use $\biguplus_{Q' \in \mathrm{reduce}(Q'')} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q'')$. The rule (Call) removes the called oracle $O[\widetilde{a'}]$ from $\mathit{Oracles}(\mathfrak{C})$.

Therefore, for all configurations $\mathfrak{C}$ reachable from the initial configuration $\mathfrak{C}_\mathrm{i}(Q_0)$, $\mathit{Oracles}(\mathfrak{C})$ contains no duplicates.

By definition of $\mathit{Oracles}$, when $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$, we have $\{O[\widetilde{a}] \mid O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P \in \mathcal{Q}\} \subseteq \mathit{Oracles}(\mathfrak{C})$. (Both sides of the inclusion are multisets.) Therefore, $\mathcal{Q}$ contains at most one element $O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P$ for each $O[\widetilde{a}]$. □

B Proof of Proposition 4.5

Proof (of Proposition 4.5) Let $n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI})$ be the number of occurrences of event or return in the instrumented configuration $\mathcal{CI}$. Let us first prove the following property:

2'. If $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ and $\mathcal{CI}_1, \ldots, \mathcal{CI}_n$ are pairwise distinct instrumented configurations such that for all $i \leq n$, $\mathcal{CI} \rightarrow_{p_i} \mathcal{CI}_i$ with $\sum_{i \leq n} p_i = 1$, then one of the following two properties holds:

P1. $n = 1$, $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_1)$, and $n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI}_1) < n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI})$.

P2. there exist pairwise distinct configurations $\mathcal{C}_1, \ldots, \mathcal{C}_n$ such that for all $i \leq n$, we have $\mathcal{C} \rightarrow_{p_i} \mathcal{C}_i$ and $\mathcal{C}_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_i)$.

We prove this property by case analysis on the possible reductions of $\mathcal{CI}$.

Let us first suppose that $\mathcal{CI}$ reduces by (Globalstore1) and (Toplevel) from a reduction $Th \longrightarrow_p Th'$ of the current thread, and let us distinguish cases depending on the latter reduction:

- The reduction comes from rules (Context in) or (Let ctx in): we have $Th = \langle env, C[e], stack, store\rangle \longrightarrow Th' = \langle env, e, (env, C) :: stack, store\rangle$ where e is not a value and C is a minimal expression or program evaluation context. Let us distinguish cases on the form of C.
  - If $C = \mathsf{return}(\mathcal{MI}, [\cdot])$, then we have $\mathsf{noinstr}_{Th}(Th) = \mathsf{noinstr}_{Th}(\langle env, e, stack, store\rangle) = \mathsf{noinstr}_{Th}(Th')$ by Definition 4.4, so by expanding this property to the complete configuration, and noting that the reduction removes one return, Property P1 holds.
  - If $C = \mathsf{event}\ ev(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)$, then we have by Definition 4.4,

    $$\begin{aligned}&\mathsf{noinstr}_{Th}(Th) = \\ &\quad \mathsf{noinstr}_{Th}(\langle env, (e_1, \ldots, e_{i-1}, e, v_{i+1}, \ldots, v_n), stack, store\rangle)\,, \\ &\mathsf{noinstr}_{Th}(Th') = \mathsf{noinstr}_{Th}(\\ &\quad \langle env, e, (env, (e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)) :: stack, store\rangle)\,,\end{aligned}$$

    and we have $\mathsf{noinstr}_{Th}(Th) \longrightarrow \mathsf{noinstr}_{Th}(Th')$, so Property P2 holds.
  - If C is neither a return nor an event context, then the reduction $Th \longrightarrow_p Th'$ implies $\mathsf{noinstr}_{Th}(Th) \longrightarrow_p \mathsf{noinstr}_{Th}(Th')$, so Property P2 holds.
- The reduction comes from rules (Context out) or (Let ctx out): we have $Th = \langle env, v, (env', C) :: stack, store\rangle \longrightarrow Th' = \langle env', C[v], stack, store\rangle$ where C is a minimal expression or program evaluation context. Let us distinguish cases on the form of C.

– If $C = \mathsf{return}(\mathcal{MI}, [\cdot])$, then we have by Definitions 4.4 and 4.2,

$$\begin{aligned}\mathsf{noinstr}_{Th}(Th) &= \mathsf{noinstr}_{Th}(\langle env, v, stack, store\rangle)\\ &\approx_{v\,Th} \mathsf{noinstr}_{Th}(Th')\,,\end{aligned}$$

so by expanding this property to the complete configuration, and noting that the reduction removes one return, Property P1 holds.

– If $C = \mathsf{event}\ ev(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)$, we have by Definition 4.4,

$$\begin{aligned}&\mathsf{noinstr}_{Th}(Th) = \mathsf{noinstr}_{Th}(\\ &\quad\langle env, v, (env', (e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)) :: stack, store\rangle)\,,\\ &\mathsf{noinstr}_{Th}(Th') =\\ &\quad\mathsf{noinstr}_{Th}(\langle env', (e_1, \ldots, e_{i-1}, v, v_{i+1}, \ldots, v_n), stack, store\rangle)\,,\end{aligned}$$

so Property P2 holds.

– If C is neither a return nor an event context, then Property P2 holds.

- The cases of (Context raise2) and (Let ctx raise) are similar to the previous case: Property P1 holds when the context is $\mathsf{return}(\mathcal{MI}, [\cdot])$; Property P2 holds otherwise.

- Property P2 holds in the other cases.

If $\mathcal{CI} \to \mathcal{CI}_1$ by (Toplevel return), then the program of the current thread is $\mathsf{return}(\mathcal{MI}, v)$ in $\mathcal{CI}$ and the only differences between $\mathcal{CI}$ and $\mathcal{CI}_1$ are that the program of the current thread is v and the set of callable modules is changed in $\mathcal{CI}_1$. Therefore, $\mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}) = \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_1)$, and the reduction removes one return, so Property P1 holds.

If $\mathcal{CI} \to \mathcal{CI}_1$ by (Toplevel event), then the program of the current thread is $\mathsf{event}\ ev(v_1, \ldots, v_n)$ in $\mathcal{CI}$ and the only differences between $\mathcal{CI}$ and $\mathcal{CI}_1$ are that the program of the current thread is $(v_1, \ldots, v_n)$ and the list of executed events is updated in $\mathcal{CI}_1$. Therefore, $\mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}) = \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_1)$, and the reduction removes one event, so Property P1 holds.

Property P2 holds for addthread, schedule, and global store related reductions. In the case of addthread, we use Assumption 4.1.

We have proved that Property 2' holds. Property 2 also holds, since it is a special case of Property 2'.

Let us now prove:

3. If $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ and $\mathcal{CI}$ cannot reduce, then $\mathcal{C}$ cannot reduce.

We need to prove that $\mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ cannot reduce. We can then use Lemma 4.3 to conclude. We distinguish cases depending on the program or expression in the current thread of $\mathcal{CI}$. If this program or expression was of the form $C[e]$ for some program or expression minimal evaluation context C, or $\mathsf{return}(\mathcal{MI}, v)$, or $\mathsf{event}\ ev(v_1, \ldots, v_n)$, the configuration $\mathcal{CI}$ could reduce. In all other cases, $\mathsf{noinstr}_{\mathcal{CI}}$ does not change the form of the possible reductions (since it transforms

tagged functions into functions that behave exactly in the same way). Property 3 is true.

Let us now prove Property 1 by induction on $n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI})$. Let us suppose that $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ and $\mathcal{C}_1, \ldots, \mathcal{C}_n$ are pairwise distinct configurations such that for all $i \leq n$, we have $\mathcal{C} \rightarrow_{p_i} \mathcal{C}_i$ with $\sum_{i \leq n} p_i = 1$.

The configuration $\mathcal{CI}$ must reduce, otherwise, by Property 3, the configuration $\mathcal{C}$ would also not reduce. Let $\mathcal{CI} \rightarrow_{p'_i} \mathcal{CI}_i$ for $i \leq n'$ be all the reductions possible from $\mathcal{CI}$. By Property 2', we are either in case P1 or in case P2.

In case P1, $\mathcal{CI}$ reduces into only one configuration $\mathcal{CI}_1$ such that $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_1)$, and $n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI}_1) < n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI})$. By induction hypothesis, there exist pairwise distinct instrumented configurations $\mathcal{CI}'_1, \ldots, \mathcal{CI}'_n$ such that for all $i \leq n$, we have $\mathcal{CI}_1 \rightarrow^*_{p_i} \mathcal{CI}'_i$ and $\mathcal{C}_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}'_i)$. As there is only one reduction from $\mathcal{CI}$ to $\mathcal{CI}_1$ with probability 1, we can conclude that Property 1 holds in this case.

In case P2, there exist pairwise distinct configurations $\mathcal{C}'_1, \ldots, \mathcal{C}'_{n'}$ such that for all $i \leq n'$, we have $\mathcal{C} \rightarrow_{p'_i} \mathcal{C}'_i$ and $\mathcal{C}'_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_i)$. Since $\sum_{i \leq n} p_i = 1$ and $\sum_{i \leq n'} p'_i = 1$, the reductions $\mathcal{C} \rightarrow_{p_i} \mathcal{C}_i$ $(i \leq n)$ are all possible reductions of $\mathcal{C}$ and the reductions $\mathcal{C} \rightarrow_{p'_i} \mathcal{C}'_i$ $(i \leq n')$ are also all possible reductions of $\mathcal{C}$, so they are the same reductions. Therefore, $n = n'$ and there exists a bijection α from $\{1, \ldots, n\}$ to $\{1, \ldots, n\}$ such that $p_i = p'_{\alpha(i)}$, $\mathcal{C}_i = \mathcal{C}'_{\alpha(i)} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_{\alpha(i)})$, and $\mathcal{CI} \rightarrow_{p_i} \mathcal{CI}_{\alpha(i)}$. By renumbering the configurations $\mathcal{CI}_i$ $(i \leq n)$, we can conclude that Property 1 holds in this case. □

C Proof of Proposition 6.5

Definition C.1 *Let* $Th \stackrel{\text{def}}{=} \langle env, pe, stack, store \rangle$ *and* $Th' \stackrel{\text{def}}{=} \langle env', pe', stack', store' \rangle$ *be two threads, such that the domains of store and store′ are disjoint.*

We define $plug(Th, Th') \stackrel{\text{def}}{=} \langle env, pe, stack \mathbin{@} stack', store \cup store' \rangle$.

Definition C.2 *A well-formed thread* $Th = \langle env, pe, stack, store \rangle$ *is a thread such that:*

1. *all store locations* $l \in S_l$ *that occur in the thread Th are bound in the store:* $l \in \mathrm{Dom}(store)$,
2. *pe and stack do not contain global store locations, nor* return, event, schedule, *or* addthread *operations.*

Lemma C.3 *If Th is a well-formed thread and* $Th \rightarrow_p Th'$, *then for all* Th'' *such that the domains of the stores of* Th'' *and of Th are disjoint, after renaming the fresh locations introduced in* $Th \rightarrow_p Th'$ *so that they do not occur in* Th'', *we have* $plug(Th, Th'') \rightarrow_p plug(Th', Th'')$ *and the domains of the stores of* Th'' *and of* Th' *are disjoint.*

Proof By reviewing the reduction rules, we have Property (P1): if $env, pe, stack \xrightarrow{L}_p env', pe', stack'$, then for every stack $stack''$, we have $env, pe, stack \mathbin{@} stack'' \xrightarrow{L}_p env', pe', stack' \mathbin{@} stack''$.

Let $Th \stackrel{\text{def}}{=} \langle env, pe, stack, store\rangle$ and $Th' \stackrel{\text{def}}{=} \langle env', pe', stack', store'\rangle$. Let us prove that, if $Th \rightarrow_p Th'$, then for every $Th'' \stackrel{\text{def}}{=} \langle env'', pe'', stack'', store''\rangle$ such that $\mathrm{Dom}(store) \cap \mathrm{Dom}(store'') = \emptyset$, we have the reduction $plug(Th, Th'') = \langle env, pe, stack \mathbin{@} stack'', store \cup store''\rangle \rightarrow_p plug(Th', Th'') = \langle env', pe', stack' \mathbin{@} stack'', store' \cup store''\rangle$ with $\mathrm{Dom}(store') \cap \mathrm{Dom}(store'') = \emptyset$. We distinguish cases on the label L present in rule (Thread).

- if L is empty, then by (Store empty), $store = store'$. We conclude by Property (P1) and rule (Thread).
- if L is $!l = v$, by (Store lookup), the location l is in the domain of the store $store$, and $store(l) = v$, and $store = store'$. We also have $(store \cup store'')(l) = v$, so $store \cup store'' \xrightarrow{!l=v} store' \cup store''$. We conclude by Property (P1) and rule (Thread).
- if L is $l := v$, then by (Store assign), the location l is in the domain of the store $store$, and $store' = store[l \mapsto v]$. The domain of the store $store \cup store''$ also contains l, so $store \cup store'' \xrightarrow{l:=v} store' \cup store''$. We conclude by Property (P1) and rule (Thread).
- if L is $\mathsf{ref}\ v = l$, then by (Store alloc), $l \notin \mathrm{Dom}(store)$. By reviewing the uses of L of the form $\mathsf{ref}\ v = l$ in the reduction rules, we can deduce that $pe = \mathsf{ref}\ v$ and $pe' = l$. By Property 1 of Definition C.2, the location l does not occur in Th. Let us take a location $l' \in S_l$ that is not in $\mathrm{Dom}(store) \cup \mathrm{Dom}(store'')$; we rename l into l'. As $l' \notin \mathrm{Dom}(store)$, the thread Th' becomes $\langle env', l', stack', store[l' \mapsto v]\rangle$, which is in the same equivalence class as Th', so we still designate this thread by Th'. Let $L' \stackrel{\text{def}}{=} (\mathsf{ref}\ v = l')$. By Property (P1), $env, pe, stack \mathbin{@} stack'' \xrightarrow{L'} env', l', stack' \mathbin{@} stack''$ and, since $l' \notin \mathrm{Dom}(store) \cup \mathrm{Dom}(store'')$, we have $store \cup store'' \xrightarrow{L'} store[l' \mapsto v] \cup store''$. We conclude that $plug(Th, Th'') = \langle env, pe, stack \mathbin{@} stack'', store \cup store''\rangle \rightarrow plug(Th', Th'') = \langle env', l', stack' \mathbin{@} stack'', store[l' \mapsto v] \cup store''\rangle$ by rule (Thread). □

Lemma C.4 *Let Th be a well-formed thread. If $Th \rightarrow_p Th'$, then Th' is also well-formed.*

Proof Let us prove that both properties of Definition C.2 are preserved.

- The only rule that may add new locations in the thread is (Store alloc), which creates a new location l and also adds it in the domain of the store. So Property 1 is preserved for Th'.
- By looking at the reduction rules, we can see that no rule can create global store locations or return, event, schedule, or addthread operations. So Property 2 is preserved for Th'.

Therefore, the thread Th' is also well-formed. □

Lemma C.5 *If v is a value of the type of the argument of the primitive s, then the thread $\langle \emptyset, env_{\mathsf{prim}}(s)\ v, [\,], \emptyset\rangle$ is well-formed.*

Proof The thread $Th_0^s = \langle \emptyset, program_{\mathsf{prim}};;, [\,], \emptyset\rangle$ is well-formed, since it contains no locations in S_l and by Assumption 6.1, it contains no schedule, addthread, return, or event operations and no global store locations.

By Assumption 6.2, $Th_0^s \to^* Th = \langle env_{\mathsf{prim}}, \varepsilon, [\,], \emptyset\rangle$, so by Lemma C.4, Th is also well-formed. Therefore, the thread $\langle \emptyset, env_{\mathsf{prim}}(s)\ v, [\,], \emptyset\rangle$ is well-formed, since by Assumption 6.3, v contains no locations and no return, event, schedule, or addthread operations since it contains no closure, and env_{prim} contains no locations and no return, event, schedule, or addthread operations since Th is well-formed. □

Proof (of Proposition 6.5) By Assumption 6.4, we have reductions of the form

$$Th_1 \stackrel{\text{def}}{=} \langle \emptyset, env_{\mathsf{prim}}(s)\ v, [\,], \emptyset\rangle \to_p^* Th_1' \stackrel{\text{def}}{=} \langle env', v', [\,], store_1'\rangle\,.$$

Let $Th_2 \stackrel{\text{def}}{=} \langle env, (), stack, store\rangle$. By Lemma C.5, Th_1 is well-formed, so by Lemmas C.3 and C.4,

$$Th = plug(Th_1, Th_2) \to_p^* plug(Th_1', Th_2) = \langle env', v', stack, store_1' \cup store\rangle\,.$$

Letting $store' \stackrel{\text{def}}{=} store_1' \cup store$, we obtain exactly the desired reductions $Th \to_p^* \langle env', v', stack, store'\rangle$, and $store' \supseteq store$. □

D Proof of Lemma 6.10

Let us first prove lemmas useful to prove Lemma 6.10.

Lemma D.1 (Write file) *Let $\mathcal{C}$ be an OCaml configuration. If $\mathcal{C}_{Th}(\mathcal{C}) = \langle env, \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]), stack, store\rangle$, $env(\mathbb{G}_{\mathsf{var}}(x)) = \mathbb{G}_{\mathsf{val}T_x}(a)$, $env \supseteq env_{\mathsf{prim}}$, and $\mathcal{C}_{globalstore}(\mathcal{C}) \supseteq globalstore(E, \mathcal{T})$, then we have $\mathcal{C} \longrightarrow^* \mathcal{C}'$ where*

- $\mathcal{C}' = \mathcal{C}[\mathsf{Th} \mapsto \langle env, (), stack, store'\rangle, \mathsf{globalstore} \mapsto globalstore']$,
- $store' \supseteq store$,
- $globalstore' \supseteq globalstore(E[x[\widetilde{a}] \mapsto a], \mathcal{T})$,
- $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$ *for all* $l \notin S_{priv}$.

Proof If $(x[\widetilde{a}], f) \in \mathsf{files}$ for some f, then we have $x[\widetilde{a}] = x[\,]$ and $\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]) = (f := \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x))$, so

$$\begin{aligned}
\mathcal{C} &\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x)), stack', store\rangle] \\
&\quad \text{where } stack' \stackrel{\text{def}}{=} (env, f := [\cdot]) :: stack \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{ser}}(T_x))\ \mathbb{G}_{\mathsf{val}T_x}(a), stack', store\rangle] \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env', \mathsf{ser}(T_x, a), stack', store'\rangle] \qquad \text{by Proposition 6.5} \\
&\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, f := \mathsf{ser}(T_x, a), stack, store'\rangle] \\
&\to \mathcal{C}' = \mathcal{C}[\mathsf{Th} \mapsto \langle env, (), stack, store'\rangle, \mathsf{globalstore} \mapsto globalstore']
\end{aligned}$$

where

$$\begin{aligned} \mathit{globalstore}' &\stackrel{\text{def}}{=} \mathcal{C}_{\mathit{globalstore}}(\mathcal{C})[f \mapsto \mathsf{ser}(T_x, a)] \\ &\supseteq \mathit{globalstore}(E, \mathcal{T})[f \mapsto \mathsf{ser}(T_x, a)] \\ &\supseteq \mathit{globalstore}(E[x[\,] \mapsto a], \mathcal{T})\,. \end{aligned}$$

The modified location f is in S_{priv}, so for all $l \notin S_{priv}$, we have $\mathit{globalstore}'(l) = \mathcal{C}_{\mathit{globalstore}}(\mathcal{C})(l)$. We have $\mathit{store}' \supseteq \mathit{store}$ by Proposition 6.5.

Otherwise, we have $\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]) = ()$ and $\mathcal{C}' = \mathcal{C}$, so

$$\begin{aligned} \mathit{globalstore}' &= \mathcal{C}_{\mathit{globalstore}}(\mathcal{C}) \\ &\supseteq \mathit{globalstore}(E, \mathcal{T}) = \mathit{globalstore}(E[x[\widetilde{a}] \mapsto a], \mathcal{T})\,, \end{aligned}$$

and for all $l \notin S_{priv}$, we have $\mathit{globalstore}'(l) = \mathcal{C}_{\mathit{globalstore}}(\mathcal{C})(l)$. □

Definition D.2 (Deserialized OCaml values for tables) *Let Tbl be a table of type $T_1 \times \cdots \times T_l$. The OCaml value that corresponds to an element $(b_1, \ldots, b_l)$ of this table is*

$$\mathbb{G}_{\mathsf{val}T_1,\ldots,T_l}(b_1, \ldots, b_l) \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_l}(a_l))\,.$$

Let $t = [a_1; \ldots; a_k]$ be a list of elements of table Tbl. The corresponding OCaml list is

$$\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t) \stackrel{\text{def}}{=} [\mathbb{G}_{\mathsf{val}T_1,\ldots,T_l}(a_1); \ldots; \mathbb{G}_{\mathsf{val}T_1,\ldots,T_l}(a_k)]\,.$$

Definition D.3 (Function filter) *Let E be a CryptoVerif environment, M a CryptoVerif boolean term, $(x_1, \ldots, x_k)$ a tuple of variables, and t a list of tuples of CryptoVerif values of type $T_{x_1} \times \cdots \times T_{x_k}$. We let $\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t)$ be the list of tuples $(a_1, \ldots, a_k)$ in t such that the term M is true when the variables $x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]$ are bound to $a_1, \ldots, a_k$ in the environment E, respectively:*

$$\begin{aligned} &\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t) \stackrel{\text{def}}{=} \\ &\quad [(a_1, \ldots, a_k) \in t \mid E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k], M \Downarrow \text{true}] \end{aligned}$$

Let us recall that our fold function on lists $\mathbb{G}_{\mathsf{fold}}$ is defined in Figure 16 as follows:

$$\mathbb{G}_{\mathsf{fold}} \stackrel{\text{def}}{=} f \to \mathsf{function}\ a \to \mathsf{function}\ [\,] \to a \mid x :: l \to f\ (\mathit{fold}\ f\ a\ l)\ x$$

Lemma D.4 *Suppose that*

$$\begin{aligned} \mathit{Th} &= \langle \mathit{env}, \mathit{fold}\ c\ [\,]\ (\mathbb{G}_{\mathsf{tbl}}(\mathit{Tbl}, t)), \mathit{stack}, \mathit{store}\rangle \\ c &= \mathsf{function}[\mathit{env}', \\ &\qquad a \to \mathsf{function}\ x \to \mathsf{try}\ (c'\ x) :: a\ \mathsf{with}\ \mathsf{Match\_failure} \to a] \\ c' &= \mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M) \end{aligned}$$

$$env' \supseteq env_{\mathsf{prim}} \cup \{\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T_x}(b) \mid x[\widetilde{a'}] \in \mathsf{fv}(M) \setminus \{x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]\}, E(x[\widetilde{a'}]) = b\}$$

$env(fold) = env'(fold) = \mathsf{letrec}[env_0, \{fold \mapsto \mathbb{G}_{\mathsf{fold}}\}\ \mathsf{in}\ fold]$
where t is a list of CryptoVerif values of type $T_{x_1} \times \cdots \times T_{x_k}$ and all occurrences of $x_1, \ldots, x_k$ in M have indices $\widetilde{a}$.

Then $Th \rightarrow^* Th'$ such that

$$Th' = \langle env'', \mathbb{G}_{\mathsf{tbldeser}}(Tbl, \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t)), stack, store' \rangle$$

for some env'' and $store'$ such that $store' \supseteq store$.

Proof We prove this lemma by induction on the length of t.

In the base case, $t = []$, so $\mathbb{G}_{\mathsf{tbl}}(Tbl, t) = []$, and

$$\begin{aligned}
Th &= \langle env, fold\ c\ []\ [], stack, store \rangle \\
&\rightarrow^* \langle env_1, (\mathsf{match}\ c\ \mathsf{with}\ \mathbb{G}_{\mathsf{fold}})\ []\ [], stack, store \rangle \\
&\quad \text{where } env_1 \stackrel{\text{def}}{=} env_0[fold \mapsto env'(fold)] \\
&\rightarrow^* \langle env'', \mathsf{match}\ []\ \mathsf{with}\ [] \mapsto a \mid x :: l \rightarrow f\ (fold\ f\ a\ l)\ x, stack, store \rangle \\
&\quad \text{where } env'' \stackrel{\text{def}}{=} env_1[f \mapsto c, a \mapsto []] \\
\rightarrow^* Th' &\stackrel{\text{def}}{=} \langle env'', [], stack, store \rangle
\end{aligned}$$

For any $E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}])$, we have $\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), []) = []$. So the lemma is correct for the base case.

In the inductive case, let $t = b :: t'$. Let $y = \mathbb{G}_{\mathsf{tblel}}(Tbl, b)$ and $l' = \mathbb{G}_{\mathsf{tbl}}(Tbl, t')$, so $\mathbb{G}_{\mathsf{tbl}}(Tbl, t) = y :: l'$. Let $b = (a_1, \ldots, a_k)$ and $y = (d_1, \ldots, d_k)$, where each d_i corresponds to a_i. Let $(d'_1, \ldots, d'_k) = \mathbb{G}_{\mathsf{val}T_1, \ldots, T_k}(b)$, where Tbl is a table of type $T_1 \times \cdots \times T_k$.

$$\begin{aligned}
Th &= \langle env, fold\ c\ []\ (y :: l'), stack, store \rangle \\
&\rightarrow^* \langle env_1, (\mathsf{match}\ c\ \mathsf{with}\ \mathbb{G}_{\mathsf{fold}})\ []\ (y :: l'), stack, store \rangle \\
&\quad \text{where } env_1 \stackrel{\text{def}}{=} env_0[fold \mapsto env'(fold)] \\
&\rightarrow^* \langle env_2, \mathsf{match}\ y :: l'\ \mathsf{with}\ [] \mapsto a \mid x :: l \rightarrow f\ (fold\ f\ a\ l)\ x, stack, store \rangle \\
&\quad \text{where } env_2 \stackrel{\text{def}}{=} env_1[f \mapsto c, a \mapsto []] \\
&\rightarrow^* \langle env_3, f\ (fold\ f\ a\ l)\ x, stack, store \rangle \\
&\quad \text{where } env_3 \stackrel{\text{def}}{=} env_2[x \mapsto y, l \mapsto l'] \\
&\rightarrow^* \langle env_3, fold\ f\ a\ l, stack_1, store \rangle \text{ where } stack_1 \stackrel{\text{def}}{=} (env_3, f\ [\cdot]\ y) :: stack \\
&\qquad\qquad \text{(arguments are evaluated from right to left)} \\
&\rightarrow^* \langle env_3, fold\ c\ []\ l', stack_1, store \rangle \\
&\rightarrow^* \langle env_4, \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''), stack_1, store_1 \rangle \qquad \text{by induction hypothesis} \\
&\quad \text{where } t'' \stackrel{\text{def}}{=} \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t')
\end{aligned}$$

$\rightarrow \langle env_3, f\ (\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''))\ y, stack, store_1\rangle$

$\rightarrow^* \langle env_3, c\ (\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''))\ y, stack, store_1\rangle$

$\rightarrow^* \langle env_5, \mathsf{try}\ (c'\ x) :: a\ \mathsf{with\ Match\_failure} \rightarrow a, stack, store_1\rangle$

where $env_5 \stackrel{\text{def}}{=} env'[a \mapsto \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''), x \mapsto y]$

$\rightarrow \langle env_5, (c'\ x) :: a, stack_2, store_1\rangle$

where $stack_2 \stackrel{\text{def}}{=} (env_5, \mathsf{try}\ [\cdot]\ \mathsf{with\ Match\_failure} \rightarrow a) :: stack$

$\rightarrow \langle env_5, c'\ x, stack_3, store_1\rangle$ where $stack_3 \stackrel{\text{def}}{=} (env_5, [\cdot] :: a) :: stack_2$

$\rightarrow^* \langle env_5[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto d_1, \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto d_k],$
$\mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x_1) = \mathbb{G}_{\mathsf{deser}}(T_{x_1})\ \mathbb{G}_{\mathsf{var}}(x_1)\ \mathsf{in} \ldots$
$\mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x_k) = \mathbb{G}_{\mathsf{deser}}(T_{x_k})\ \mathbb{G}_{\mathsf{var}}(x_k)\ \mathsf{in}$
$\mathsf{if}\ (\mathbb{G}_{\mathsf{M}}(M))\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{else\ raise\ Match\_failure},$
$stack_3, store_1\rangle$

$\rightarrow^* \langle env_6,$
$\mathsf{if}\ (\mathbb{G}_{\mathsf{M}}(M))\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{else\ raise\ Match\_failure},$
$stack_3, store_2\rangle$ where $env_6 \stackrel{\text{def}}{=} env_5[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto d'_1, \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto d'_k]$
by Proposition 6.5 applied k times

$\rightarrow Th_1 \stackrel{\text{def}}{=} \langle env_6, \mathbb{G}_{\mathsf{M}}(M), stack_4, store_2\rangle$

where $stack_4 \stackrel{\text{def}}{=} (env_6, \mathsf{if}\ [\cdot]\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))$
$\mathsf{else\ raise\ Match\_failure}) :: stack_3$

The environment env_6 contains env_{prim} and $env(E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k], M)$. Let r be the CryptoVerif value such that $E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k], M \Downarrow r$. So by Lemma 6.8,

$$Th_1 \rightarrow^* Th_2 \stackrel{\text{def}}{=} \langle env_7, \mathbb{G}_{\mathsf{val}bool}(r), stack_4, store_3\rangle$$

and by Lemma 6.8, Proposition 6.5, and the induction hypothesis, we have $store_3 \supseteq store_2 \supseteq store_1 \supseteq store$.

Let us suppose that $r = \text{true}$. In this case, $\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t) = b :: \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t')$. Let us denote $t''' \stackrel{\text{def}}{=} \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t)$.

$Th_2 = \langle env_7, \text{true}, stack_4, store_3\rangle$

$\rightarrow^* \langle env_6, \mathsf{if\ true\ then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{else\ raise\ Match\_failure},$
$stack_3, store_3\rangle$

$\rightarrow \langle env_6, (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)), stack_3, store_3\rangle$

$\rightarrow^* \langle env_6, (d'_1, \ldots, d'_k), stack_3, store_3\rangle$

$\rightarrow \langle env_5, (d'_1, \ldots, d'_k) :: a, stack_2, store_3\rangle$

$\rightarrow^* \langle env_5, \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t'''), stack_2, store_3\rangle$

since $\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''') = (d'_1, \ldots, d'_k) :: (\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''))$

$$\rightarrow^* \langle env_5, \mathsf{try}\ \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''')\ \mathsf{with\ Match\_failure} \rightarrow a, stack, store_3\rangle$$
$$\rightarrow Th' \stackrel{\text{def}}{=} \langle env_5, \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t'''), stack, store_3\rangle$$

So the lemma is correct in this case.

Let us now suppose that $r = \text{false}$. In this case, $\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t) = \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t')$.

$$\begin{aligned}
Th_2 &= \langle env_7, \text{false}, stack_4, store_3\rangle \\
&\rightarrow^* \langle env_6, \mathsf{if}\ \text{false}\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{else\ raise\ Match_failure}, \\
&\qquad stack_3, store_3\rangle \\
&\rightarrow \langle env_6, \mathsf{raise\ Match_failure}, stack_3, store_3\rangle \\
&\rightarrow^* \langle env_5, \mathsf{try\ raise\ Match_failure\ with\ Match_failure} \rightarrow a, stack, store_3\rangle \\
&\rightarrow^* \langle env_5, a, stack, store_3\rangle \\
\rightarrow Th' &\stackrel{\text{def}}{=} \langle env_5, \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''), stack, store_3\rangle
\end{aligned}$$

As in the previous case, the lemma is also correct in this case. □

Proof (of Lemma 6.10) Let us prove this lemma by looking at each case.

- The random number generation construct:

  On the CryptoVerif side, for each element b of type T, we have the following reduction:

  $$E, x[\widetilde{a}] \stackrel{R}{\leftarrow} T; P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_{1/|T|} E[x[\widetilde{a}] \mapsto b], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$$

  The variable $x[\widetilde{a}]$, bound in P, is free in P'.

  On the OCaml side, we have, for all $b \in T$,

  $$\begin{aligned}
  \mathcal{C} &= \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{random}}(T)\ ()\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P')), \\
  &\qquad\qquad stack, store\rangle] \\
  &\rightarrow \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{random}}(T)\ (), stack', store\rangle] \\
  &\qquad \text{where } stack' \stackrel{\text{def}}{=} (env, \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = [\cdot]\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'))) \\
  &\qquad\qquad\qquad :: stack \\
  &\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{random}}(T))\ (), stack', store\rangle] \\
  &\rightarrow^*_{1/|T|} \mathcal{C}[\mathsf{Th} \mapsto \langle env', \mathbb{G}_{\mathsf{val}T}(b), stack', store'\rangle] \qquad \text{by Proposition 6.5} \\
  &\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'), stack, store'\rangle] \\
  &\qquad \text{where } env'' \stackrel{\text{def}}{=} env[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)] \\
  &\rightarrow \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]), ([\cdot]; \mathbb{G}(P')) :: stack, store'\rangle] \\
  &\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env'', (); \mathbb{G}(P'), stack, store''\rangle, \mathsf{globalstore} \mapsto globalstore'] \\
  &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by Lemma D.1} \\
  &\rightarrow \mathcal{C}' = \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}(P'), stack, store''\rangle, \mathsf{globalstore} \mapsto globalstore']
  \end{aligned}$$

This sequence of reductions describes the set of traces $\mathcal{CTS}_b$ that corresponds to the CryptoVerif reduction that adds to its environment the value b bound to $x[\widetilde{a}]$. We have $\Pr[\mathcal{CTS}_b] = 1/|T|$.

Let $E' \stackrel{\text{def}}{=} E[x[\widetilde{a}] \mapsto b]$. We have $env(E', P') = env(E, P)[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(a)]$, so $env'' \supseteq env_{\mathsf{prim}} \cup env(E', P')$. By Lemma D.1, we have $globalstore' \supseteq globalstore(E', \mathcal{T})$ and $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$ for all $l \notin S_{priv}$. By Lemma D.1 and Proposition 6.5, we have $store'' \supseteq store' \supseteq store$. Events are unchanged on both sides. So this construct satisfies the lemma.

- The assignment construct:

  On the CryptoVerif side, let us suppose that $E, M \Downarrow b$. We have the following reduction:

$$E, x[\widetilde{a}] \leftarrow M; P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow E[x[\widetilde{a}] \mapsto b], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$$

  Let us denote $T = T_M$. The variable $x[\widetilde{a}]$, bound in P, is free in P'.

  On the OCaml side, we have:

$$\begin{aligned}
&\mathcal{C} = \mathcal{C}[\mathsf{Th} \mapsto Th] \text{ where } Th \stackrel{\text{def}}{=} \\
&\quad \langle env, \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P')), stack, store\rangle \\
&\rightarrow \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{M}}(M), stack', store\rangle] \\
&\quad \text{where } stack' \stackrel{\text{def}}{=} (env, \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = [\cdot]\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'))) \\
&\qquad\qquad :: stack \\
&\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{val}T}(b), stack', store'\rangle] \qquad \text{by Lemma 6.8} \\
&\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'), stack, store'\rangle] \\
&\quad \text{where } env' \stackrel{\text{def}}{=} env[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)] \\
&\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env', (); \mathbb{G}(P'), stack, store''\rangle, \mathsf{globalstore} \mapsto globalstore'] \\
&\qquad\qquad \text{by Lemma D.1} \\
&\rightarrow \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env', \mathbb{G}(P'), stack, store''\rangle, \mathsf{globalstore} \mapsto globalstore']
\end{aligned}$$

  This sequence of reductions describes the set of traces $\mathcal{CTS}_1$ that corresponds to the CryptoVerif reduction. We have $\Pr[\mathcal{CTS}_1] = 1$.

  Let $E' \stackrel{\text{def}}{=} E[x[\widetilde{a}] \mapsto b]$. We have $env(E', P') = env(E, P)[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)]$, so $env' \supseteq env_{\mathsf{prim}} \cup env(E', P')$. By Lemma D.1, we have $globalstore' \supseteq globalstore(E', \mathcal{T})$ and $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$ for all $l \notin S_{priv}$. By Lemmas D.1 and 6.8, we have $store'' \supseteq store' \supseteq store$. Events are unchanged on both sides. So this construct satisfies the lemma.

- The conditional construct:

  On the CryptoVerif side, let us suppose that $E, M \Downarrow \text{true}$. The same reasoning can be applied in the case that $E, M \Downarrow \text{false}$. We have the

following reduction:

$$E, \mathsf{if}\ M\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to E, P_1, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}.$$

On the OCaml side, we implement the CryptoVerif *bool* type with booleans in OCaml, and we have $\mathbb{G}_{\mathsf{val}bool}(\text{true}) = \text{true}$ and $\mathbb{G}_{\mathsf{val}bool}(\text{false}) = \text{false}$. Let $Th = \mathcal{C}_{Th}(\mathcal{C})$.

$$\begin{aligned}
Th &= \langle env, \mathsf{if}\ \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{then}\ \mathbb{G}(P_1)\ \mathsf{else}\ \mathbb{G}(P_2), stack, store\rangle \\
&\to \langle env, \mathbb{G}_{\mathsf{M}}(M), stack', store\rangle \\
&\quad \text{where } stack' \stackrel{\text{def}}{=} (env, \mathsf{if}\ [\cdot]\ \mathsf{then}\ \mathbb{G}(P_1)\ \mathsf{else}\ \mathbb{G}(P_2)) :: stack \\
&\to^* \langle env, \text{true}, stack', store'\rangle \qquad \text{by Lemma 6.8} \\
&\to \langle env, \mathsf{if}\ \text{true}\ \mathsf{then}\ \mathbb{G}(P_1)\ \mathsf{else}\ \mathbb{G}(P_2), stack, store'\rangle \\
\to Th' &\stackrel{\text{def}}{=} \langle env, \mathbb{G}(P_1), stack, store'\rangle
\end{aligned}$$

By (Globalstore1) and (Toplevel), we obtain $\mathcal{C} \to^* \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto Th']$. This sequence of reductions describes the set of traces $\mathcal{CTS}_1$ that corresponds to the CryptoVerif reduction. We have $\Pr[\mathcal{CTS}_1] = 1$.

The CryptoVerif environment E and tables $\mathcal{T}$, the OCaml environment *env* and global store *globalstore*, and the events on both sides are unchanged. By Lemma 6.8, we have $store' \supseteq store$, so this construct satisfies the lemma.

- The insert construct:

  On the CryptoVerif side, let us suppose that $E, M_i \Downarrow a_i$. We have the following reduction:

  $$E, \mathsf{insert}\ Tbl(M_1, \ldots, M_k); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to E, P', \mathcal{T}', \mathcal{Q}, \mathcal{R}, \mathcal{E}\,,$$

  where $\mathcal{T}' \stackrel{\text{def}}{=} \mathcal{T}[Tbl \mapsto (a_1, \ldots, a_k) :: \mathcal{T}(Tbl)]$. Let the type of the table Tbl be $T_1 \times \cdots \times T_k$.

  On the OCaml side, there exists a unique f such that $(Tbl, f) \in \mathsf{tables}$. Let $globalstore = \mathcal{C}_{globalstore}(\mathcal{C})$. Since $globalstore \supseteq globalstore(E, \mathcal{T})$, we have

  $$globalstore(f) = t \text{ where } t \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{tbl}}(Tbl, \mathcal{T}(Tbl))\,.$$

  Let $t' \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{tblel}}(Tbl, (a_1, \ldots, a_k)) :: t$. By definition of $\mathbb{G}_{\mathsf{tbl}}$, we have $t' = \mathbb{G}_{\mathsf{tbl}}(Tbl, \mathcal{T}'(Tbl))$. Let $globalstore' = globalstore[f \mapsto t']$.

  $$\begin{aligned}
  \mathcal{C} &= \mathcal{C}[\mathsf{Th} \mapsto \langle env, f := e :: (!f); \mathbb{G}(P'), stack, store\rangle] \\
  &\quad \text{where } e \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{ser}}(T_1)\ \mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{ser}}(T_k)\ \mathbb{G}_{\mathsf{M}}(M_k)) \\
  &\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, e :: t, stack', store\rangle] \\
  &\quad \text{where } stack' \stackrel{\text{def}}{=} (env, f := [\cdot]) :: (env, [\cdot]; \mathbb{G}(P')) :: stack
  \end{aligned}$$

$$\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{tblel}}(Tbl, (a_1, \ldots, a_k)) :: t, stack', store' \rangle]$$
by Lemma 6.8 and Proposition 6.5
$$\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, f := t', (env, [\cdot]; \mathbb{G}(P')) :: stack, store' \rangle]$$
$$\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, (); \mathbb{G}(P'), stack, store' \rangle, \mathsf{globalstore} \mapsto globalstore']$$
$$\to \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}(P'), stack, store' \rangle, \mathsf{globalstore} \mapsto globalstore']$$

This sequence of reductions describes the set of traces $\mathcal{CTS}_1$ that corresponds to the CryptoVerif reduction. We have $\Pr[\mathcal{CTS}_1] = 1$.

The global store is modified so that $globalstore' \supseteq globalstore(E, \mathcal{T}')$ and $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$ for all $l \notin S_{priv}$, and the environments and events are unchanged on both sides. Moreover, by Proposition 6.5 and Lemma 6.8, we have $store' \supseteq store$, so this construct satisfies the lemma.

- The get construct:

  On the CryptoVerif side, let us consider a CryptoVerif configuration such that its program is

  $$P = \mathsf{get}\ Tbl(x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P'\ \mathsf{else}\ P''\,.$$

  Let the type of the table Tbl be $T_1 \times \cdots \times T_k$.

  We have two cases depending on whether there is a value in the table Tbl that satisfies M or not. Let $l' \stackrel{\text{def}}{=} \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), \mathcal{T}(Tbl)) = [b_1, \ldots, b_m]$. This list contains every element of $\mathcal{T}(Tbl)$ such that M is true.

  If l' is empty, then:

  $$E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to E, P'', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}.$$

  If l' is not empty, then there is a reduction for each element $b = (a_1, \ldots, a_k)$ in l',

  $$E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_{p_b} E_b, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\,,$$

  where $p_b \stackrel{\text{def}}{=} \sum_{\{j \in \{1,\ldots,m\} | b_j = b\}} \mathrm{among}(\{1, \ldots, m\}, j)$ and $E_b \stackrel{\text{def}}{=} E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k]$.

  On the OCaml side, let us denote

  $$e \stackrel{\text{def}}{=} \mathsf{if}\ l = [\,]\ \mathsf{then}\ \mathbb{G}(P')\ \mathsf{else}$$
  $$\quad \mathsf{let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = \mathsf{random}_l\ l\ \mathsf{in}$$
  $$\quad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P))$$
  $$e_1 \stackrel{\text{def}}{=} \mathsf{try}\ (\mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M)\ x) :: a\ \mathsf{with}\ \mathsf{Match\_failure} \to a$$

  There exists a unique f such that $(Tbl, f) \in \mathsf{tables}$, and we have

  $$\mathcal{C} = \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathsf{let}\ l = e_2\ \mathsf{in}\ e, stack, store \rangle]$$
  $$\text{where } e_2 \stackrel{\text{def}}{=} \mathrm{read\_table}(f, \mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M))$$
  $$= \mathsf{let\ rec}\ fold = \mathsf{function}\ \mathbb{G}_{\mathsf{fold}}\ \mathsf{in}$$
  $$fold\ (\mathsf{function}\ a \to\ x \to\ e_1)\ [\,]\ !f$$

$\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, e_2, stack', store\rangle]$

where $stack'' \stackrel{\text{def}}{=} (env, \mathsf{let}\ l = [\cdot]\ \mathsf{in}\ e) :: stack$

$\to \mathcal{C}[\mathsf{Th} \mapsto \langle env', fold\ (\mathsf{function}\ a \to\ x \to\ e_1)\ [\,]\ !f, stack', store\rangle]$

where $env' \stackrel{\text{def}}{=} env[fold \mapsto \mathsf{letrec}[env, \{fold \mapsto \mathbb{G}_{\mathsf{fold}}\}\ \mathsf{in}\ fold]]$

$\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env', e_3, stack', store\rangle]$

where $e_3 \stackrel{\text{def}}{=}$

$fold\ \mathsf{function}[env', a \to\ \mathsf{function}\ x \to\ e_1]\ [\,]\ \mathbb{G}_{\mathsf{tbl}}(Tbl, \mathcal{T}(Tbl))$

$\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}_{\mathsf{tbldeser}}(Tbl, l'), stack', store'\rangle]$ by Lemma D.4

since $l' = \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), \mathcal{T}(Tbl))$

$\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathsf{let}\ l = \mathbb{G}_{\mathsf{tbldeser}}(Tbl, l')\ \mathsf{in}\ e, stack, store'\rangle]$

$\to \mathcal{C}_1 \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env'', e, stack, store'\rangle]$

where $env'' \stackrel{\text{def}}{=} env[l \mapsto \mathbb{G}_{\mathsf{tbldeser}}(Tbl, l')]$

Now, if l' is empty, then $env''(l) = [\,]$, so

$$\mathcal{C}_1 \to^* \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}(P''), stack, store'\rangle]$$

The sequence of reductions $\mathcal{C} \to^* \mathcal{C}_1 \to^* \mathcal{C}'$ describes the set of traces $\mathcal{CTS}_1$ that corresponds to the unique CryptoVerif reduction that can happen when l' is empty. We have $\Pr[\mathcal{CTS}_1] = 1$.

The CryptoVerif environment E is unchanged and the OCaml environment env'' is an extension of env, so we have $env'' \supseteq env_{\mathsf{prim}} \cup env(E, P'')$. The CryptoVerif tables, the global store, and the events on both sides are unchanged. By Lemma D.4, we have $store' \supseteq store$. So, in this case, the get construct satisfies the lemma.

If l' is not empty, then let $b = (a_1, \ldots, a_k)$ be any element of l', and let $v = \mathbb{G}_{\mathsf{val}T_1,\ldots,T_k}(Tbl, b) = (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k))$. We have $env''(l) = \mathbb{G}_{\mathsf{tbldeser}}(Tbl, l')$. Let $env''(l) = [v_1; \ldots; v_m]$. The set $S \stackrel{\text{def}}{=} \{j \in \{1, \ldots, m\} \mid v = v_j\}$ is equal to the set $\{j \in \{1, \ldots, m\} \mid b = b_j\}$, because the function $b \mapsto \mathbb{G}_{\mathsf{val}T_1,\ldots,T_k}(Tbl, b)$ is injective. Hence, we have $p_b = \sum_{j \in S} \mathrm{among}(\{1, \ldots, m\}, j)$.

$\mathcal{C}_1 \to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env'', e_4, stack, store'\rangle]$

where $e_4 \stackrel{\text{def}}{=} \mathsf{let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = \mathsf{random}_l\ l\ \mathsf{in}$

$(\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P'))$

$\to^*_{p_b} \mathcal{C}[\mathsf{Th} \mapsto \langle env'', e_5, stack, store''\rangle]$ by Proposition 6.5

where $e_5 \stackrel{\text{def}}{=} \mathsf{let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = v\ \mathsf{in}$

$(\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P'))$

$$\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env''', \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P'), stack, store''\rangle]$$

where $env''' \stackrel{\text{def}}{=}$

$$env''[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_k}(a_k)]$$

$$\to^* \mathcal{C}'_b \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env''', \mathbb{G}(P'), stack, store'''\rangle, \mathsf{globalstore} \mapsto globalstore'] \quad \text{by Lemma D.1}$$

The sequence of reductions $\mathcal{C} \to^* \mathcal{C}_1 \to^* \mathcal{C}'_b$ describes the set of traces $\mathcal{CTS}_b$ that corresponds to the CryptoVerif reduction in which the element $b = (a_1, \ldots, a_k)$ of l' is chosen. We have $\Pr[\mathcal{CTS}_b] = p_b$.

By Lemma D.1, $globalstore' \supseteq globalstore(E_b, \mathcal{T})$, and $globalstore'$ and $\mathcal{C}_{globalstore}(\mathcal{C})$ are equal on all locations not in S_{priv}, since $E_b = E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k]$. Since the OCaml environment is $env''' = env[l \mapsto \ldots, \mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_k}(a_k)]$, we have $env''' \supseteq env_{\mathsf{prim}} \cup env(E_b, P')$. The events are unchanged on both sides. By Lemma D.1, Proposition 6.5, and Lemma D.4, we have $store''' \supseteq store'' \supseteq store' \supseteq store$. So, in this case, the get construct also satisfies the lemma.

- The event construct:

  On the CryptoVerif side, let us suppose that $E, M_j \Downarrow a_j$ for all $j \leq l$. We have the following reduction:

  $$E, \mathsf{event}\ ev(M_1, \ldots, M_l); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}'\,,$$

  where $\mathcal{E}' \stackrel{\text{def}}{=} ev(a_1, \ldots, a_l) :: \mathcal{E}$. Let us denote $T_1 \times \cdots \times T_l$ the type of the event ev.

  On the OCaml side, let us denote

  $$events' \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{ev}}(\mathcal{E}') = ev(\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_l}(a_l)) :: events\,.$$

  We have

  $$\begin{aligned}
  \mathcal{C} &= \mathcal{C}[\mathsf{Th} \mapsto \langle env, e; \mathbb{G}(P'), stack, store\rangle] \\
  &\quad \text{where } e \stackrel{\text{def}}{=} \mathsf{event}\ ev(\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_l)) \\
  &\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, e, stack', store, \rangle] \\
  &\quad \text{where } stack' \stackrel{\text{def}}{=} (env, [\cdot]; \mathbb{G}(P')) :: stack \\
  &\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, e', stack', store'\rangle] \quad \text{by Lemma 6.8} \\
  &\quad \text{where } e' \stackrel{\text{def}}{=} \mathsf{event}\ ev(\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_l}(a_l)) \\
  &\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, (), stack', store'\rangle, \mathsf{events} \mapsto events'] \\
  \to^* \mathcal{C}' &\stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}(P'), stack, store'\rangle, \mathsf{events} \mapsto events']
  \end{aligned}$$

  This sequence of reductions describes the set of traces $\mathcal{CTS}_1$ that corresponds to the CryptoVerif reduction. We have $\Pr[\mathcal{CTS}_1] = 1$.

The CryptoVerif environment E and tables $\mathcal{T}$ and the OCaml environment *env* and global store *globalstore* are unchanged. We have $events' = \mathbb{G}_{\mathsf{ev}}(\mathcal{E}')$. By Lemma 6.8, we have $store' \supseteq store$, so this construct satisfies the lemma. □

E Proof of Lemma 6.16 and Proposition 6.17

Proof (of Lemma 6.16) Let us first show by induction on *steps* that, if $\mathcal{CS} \rightarrow^* \mathcal{CS}'$ in at most *steps* steps and $\mathcal{CS}'$ does not reduce, or $\mathcal{CS} \rightarrow^* \mathcal{CS}'$ in exactly *steps* steps, $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$, and $P = x[a] \leftarrow simulate(s[a]); P'$, then $\mathfrak{C}, steps, \mathcal{CS} \rightsquigarrow^* E[x[a] \mapsto simreturn(\mathcal{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$.

- If $steps = 0$ or $\mathcal{CS}$ does not reduce, then $\mathcal{CS}' = \mathcal{CS}$ and $\mathfrak{C}, steps, \mathcal{CS} \rightsquigarrow E[x[a] \mapsto simreturn(\mathcal{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ by (Leave Simulator).
- If $steps > 0$ and $\mathcal{CS} \rightarrow \mathcal{CS}_1$, then $\mathcal{CS}_1 \rightarrow^* \mathcal{CS}'$ in at most $steps - 1$ steps and $\mathcal{CS}'$ does not reduce, or $\mathcal{CS}_1 \rightarrow^* \mathcal{CS}'$ in exactly $steps - 1$ steps, so by induction hypothesis, $\mathfrak{C}, steps - 1, \mathcal{CS}_1 \rightsquigarrow^* E[x[a] \mapsto simreturn(\mathcal{CS})], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$. By (Simulator), $\mathfrak{C}, steps, \mathcal{CS} \rightsquigarrow \mathfrak{C}, steps - 1, \mathcal{CS}_1$, so $\mathfrak{C}, steps, \mathcal{CS} \rightsquigarrow^* E[x[a] \mapsto simreturn(\mathcal{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$.

Let us now prove that, if $\mathfrak{C} \rightarrow_p \mathfrak{C}'$, then there is a trace $\mathfrak{C} \rightsquigarrow_p^* \mathfrak{C}'$ and all intermediate configurations in this trace (if any) are of the form $\mathfrak{C}, steps, \mathcal{CS}$. Let $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$.

- If P is not of the form $x[a] \leftarrow simulate(s[a]); P'$ for any x, a, P', then $\mathfrak{C} \rightsquigarrow_p \mathfrak{C}'$ by (CryptoVerif).
- If $P = x[a] \leftarrow simulate(s[a]); P'$ for some x, a, P', then by the semantics of CryptoVerif, $s[a] \in \mathrm{Dom}(E)$, $E(s[a])$ is of type $T_{\mathcal{CS}}$, and $\mathfrak{C}' = E[x[a] \mapsto simulate(E(s[a]))], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$. Since $E(s[a])$ is of type $T_{\mathcal{CS}}$, there exists a configuration $\mathcal{CS}$ such that $E(s[a]) = \mathsf{repr}(\mathcal{CS})$. By reduction rule (Enter Simulator), $\mathfrak{C} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} = \mathfrak{C}, N_{\mathsf{steps}}, \mathcal{CS}$. Moreover, by definition of *simulate*, $\mathcal{CS} \rightarrow^* \mathcal{CS}'$ in at most N_{steps} steps and $\mathcal{CS}'$ does not reduce, or $\mathcal{CS} \rightarrow^* \mathcal{CS}'$ in exactly N_{steps} steps, and $simulate(\mathsf{repr}(\mathcal{CS})) = simreturn(\mathcal{CS}')$. By the result shown above, $\mathfrak{C} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} \rightsquigarrow^* E[x[a] \mapsto simreturn(\mathcal{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} = \mathfrak{C}'$.

Finally, let us show that, if $\mathfrak{C}$ does not reduce by $\rightarrow$, then it does not reduce by $\rightsquigarrow$ either. Let $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$.

- If P is not of the form $x[a] \leftarrow simulate(s[a]); P'$ for any x, a, P', then the only rule applicable to reduce $\mathfrak{C}$ by $\rightsquigarrow$ is (CryptoVerif), and it cannot be applied because $\mathfrak{C}$ does not reduce by $\rightarrow$. Hence $\mathfrak{C}$ does not reduce by $\rightsquigarrow$.
- If $P = x[a] \leftarrow simulate(s[a]); P'$ for some x, a, P', then either $s[a] \notin \mathrm{Dom}(E)$ or $E(s[a]) \notin T_{\mathcal{CS}}$. The only rule applicable to reduce $\mathfrak{C}$ by $\rightsquigarrow$ is (Enter Simulator), and it does not apply when $s[a] \notin \mathrm{Dom}(E)$ or $E(s[a]) \notin$

$T_{\mathcal{CS}}$. Hence $\mathfrak{C}$ does not reduce by $\rightsquigarrow$. (We could also show that, because the CryptoVerif configurations are well-typed, $\mathfrak{C}$ always reduces when $P = x[a] \leftarrow \mathit{simulate}(s[a]); P'$.) □

Proof (of Proposition 6.17) For $b \in \{\text{true}, \text{false}\}$, let $\mathfrak{CTS}_b$ be the set of complete CryptoVerif traces using $\rightarrow$ starting at $\mathfrak{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = b$. By the first property of Lemma 6.16, we can map each trace $\mathfrak{CT} \in \mathfrak{CTS}_b$ into a trace $\mathfrak{CT}^{\mathsf{cs}}$ using $\rightsquigarrow$ and starting at $\mathfrak{C}$, such that the configurations of the form $\mathfrak{C}$ of $\mathfrak{CT}^{\mathsf{cs}}$ are exactly the same as in $\mathfrak{CT}$ and $\Pr[\mathfrak{CT}^{\mathsf{cs}}] = \Pr[\mathfrak{CT}]$.

Let $\mathfrak{CTS}_b^{\mathsf{cs}}$ be the set of these traces $\mathfrak{CT}^{\mathsf{cs}}$. Let us show that $\mathfrak{CTS}_b^{\mathsf{cs}}$ is the set of complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathfrak{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = b$.

The list of events $\mathcal{E}$ in the last configuration of $\mathfrak{CT}^{\mathsf{cs}}$ is the same as in $\mathfrak{CT}$, so it satisfies $D(\mathcal{E}) = b$. By the second property of Lemma 6.16, since $\mathfrak{CT}$ is complete, $\mathfrak{CT}^{\mathsf{cs}}$ is also complete. Since the mapping from $\mathfrak{CT}$ to $\mathfrak{CT}^{\mathsf{cs}}$ is injective, we have $\Pr[\mathfrak{CTS}_b^{\mathsf{cs}}] = \Pr[\mathfrak{CTS}_b]$.

Moreover, if a configuration $\mathfrak{C}^{\mathsf{cs}}$ reduces by $\rightsquigarrow$ into another configuration, then the sum of the probabilities of all the possible reductions from $\mathfrak{C}^{\mathsf{cs}}$ is 1:

$$\sum_{\{\mathfrak{C}^{\mathsf{cs}\prime} \mid \mathfrak{C}^{\mathsf{cs}} \rightsquigarrow_{p(\mathfrak{C}^{\mathsf{cs}\prime})} \mathfrak{C}^{\mathsf{cs}\prime}\}} p(\mathfrak{C}^{\mathsf{cs}\prime}) = 1\,.$$

Indeed, the rules that define $\rightsquigarrow$ are mutually exclusive. If $\mathfrak{C}^{\mathsf{cs}}$ reduces by rule (CryptoVerif), then the property holds because it holds for the semantics of CryptoVerif. Otherwise, a single reduction is possible, and it has probability 1.

Using the same property for $\rightarrow$, the probability of all complete traces using $\rightarrow$ starting from $\mathfrak{C}$ is 1, so $\Pr[\mathfrak{CTS}_{\text{true}}] + \Pr[\mathfrak{CTS}_{\text{false}}] = 1$. So $\Pr[\mathfrak{CTS}_{\text{true}}^{\mathsf{cs}}] + \Pr[\mathfrak{CTS}_{\text{false}}^{\mathsf{cs}}] = 1$. Since the sum of the probabilities of all the possible reductions from each configuration by $\rightsquigarrow$ is 1, the probability of all complete traces using $\rightsquigarrow$ starting from $\mathfrak{C}$ is 1, so all these traces are in $\mathfrak{CTS}_{\text{true}}^{\mathsf{cs}}$ or $\mathfrak{CTS}_{\text{false}}^{\mathsf{cs}}$. Hence all complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathfrak{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = b$ are in $\mathfrak{CTS}_b^{\mathsf{cs}}$.

So $\Pr[\mathfrak{C}(\rightsquigarrow) : D] = \Pr[\mathfrak{CTS}_{\text{true}}^{\mathsf{cs}}] = \Pr[\mathfrak{CTS}_{\text{true}}] = \Pr[\mathfrak{C} : D]$. □

F Proof of Lemmas 6.28 and 6.29

Proof (of Lemma 6.28) Let $\mathfrak{C}_0^{\mathsf{cs}} \stackrel{\text{def}}{=} \mathfrak{C}^0(Q_0, \mathit{program}_0)$.

$$\mathfrak{C}_0^{\mathsf{cs}} = \emptyset, \mathsf{let}\ x[\,] : \mathit{bitstring} = O_{\text{start}}()\ \mathsf{in}\ \mathsf{return}(x[\,])\ \mathsf{else}\ \mathsf{end}, \mathcal{T}_0, \mathcal{Q}_0, \emptyset, [\,]$$
$$\text{where } \mathcal{Q}_0 \stackrel{\text{def}}{=} \{Q_{\text{start}}(Q_0, \mathit{program}_0)\} \cup \bigcup_{a \leq N_{\mathsf{rand}+\mathsf{calls}}} \{Q_{\mathit{loop}}\{a/i'\}\} \cup \text{reduce}(Q_0)$$

$$
\begin{aligned}
&\rightsquigarrow \emptyset, P_1, \mathcal{T}_0, \mathcal{Q}_1, \mathcal{R}_1, [\,] \\
&\quad \text{where } \mathcal{Q}_1 \stackrel{\text{def}}{=} \mathcal{Q}_0 \setminus \{Q_{\text{start}}(Q_0, \mathit{program}_0)\}, \\
&\qquad\quad P_1 \stackrel{\text{def}}{=} s_0[\,] \leftarrow s_0(Q_0, \mathit{program}_0); P_2, \\
&\qquad\quad P_2 \stackrel{\text{def}}{=} \mathsf{let}\ r[\,] : T_{\mathcal{CS}} = \mathsf{loop}\ O_{\text{loop}}[1](s_0)\ \mathsf{in\ end\ else\ end}, \\
&\qquad\quad \mathcal{R}_1 \stackrel{\text{def}}{=} [x[\,], \mathsf{return}(x[\,]), \mathsf{end}] \\
&\rightsquigarrow E_1, P_2, \mathcal{T}_0, \mathcal{Q}_1, \mathcal{R}_1, [\,] \\
&\quad \text{where } E_1 \stackrel{\text{def}}{=} \{s_0[\,] \mapsto s_0(Q_0, \mathit{program}_0)\} \\
&\rightsquigarrow E_1, P_3, \mathcal{T}_0, \mathcal{Q}_1, \mathcal{R}_1, [\,] \\
&\quad \text{where } P_3 \stackrel{\text{def}}{=} \mathsf{let}\ (r'_{1,r}[\,] : T_{\mathcal{CS}}, b_{1,r}[\,] : \mathit{bool}) = O_{\text{loop}}[1](s_0) \\
&\qquad\qquad\qquad \mathsf{in}\ P_{\mathit{return\text{-}loop}}(1)\ \mathsf{else\ end} \\
&\rightsquigarrow E_2, P_{\mathit{loop}}\{1/i'\}, \mathcal{T}_0, \mathcal{Q}_2, \mathcal{R}_{\mathit{loop}}(1), [\,] \\
&\quad \text{where } E_2 \stackrel{\text{def}}{=} E_1[s[1] \mapsto s_0(Q_0, \mathit{program}_0)], \\
&\qquad\quad \mathcal{Q}_2 \stackrel{\text{def}}{=} \mathcal{Q}_1 \setminus \{Q_{\mathit{loop}}\{1/i'\}\}, \\
\rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} &\stackrel{\text{def}}{=} E_2, P_{\mathit{loop}}\{1/i'\}, \mathcal{T}_0, \mathcal{Q}_2, \mathcal{R}_{\mathit{loop}}(1), [\,], N_{\mathsf{steps}}, \mathcal{CS}_0 \\
&\quad \text{where } \mathcal{CS}_0 \stackrel{\text{def}}{=} ([\langle \emptyset, \mathit{program}_0, [\,], \emptyset \rangle], \mathit{globalstore}_0, 1), \mathcal{RI}_0, \emptyset
\end{aligned}
$$

We have

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

Let $\mathcal{CT}$ be the trace that consists only of the configuration $\mathcal{C}_0(Q_0, \mathit{program}_0)$. Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}$. Properties 1, 2, and 4 hold. The set $\mathcal{O}(\mathcal{RI}_0)$ contains all the oracles that can be called at the beginning, and

$$
\mathcal{Q}_2 = \bigcup_{2 \le a \le N_{\mathsf{rand}+\mathsf{calls}}} \{Q_{\mathit{loop}}\{a/i'\}\} \cup \text{reduce}(Q_0)\,,
$$

so Property 3 holds. As mentioned in Section 3.2.6, the initial program $\mathit{program}_0$ does not contain locations in S_l, so Property 5 holds. As also mentioned in Section 3.2.6, $\mathit{program}_0$ contains no closure, and as mentioned in Section 5, $\mathit{program}_0$ contains no tagged function, no return, and no event except in parts $\mathit{program}(\mu_{\mathsf{role}})$ inside addthread. So Property 6(b) holds. By Assumption 5.1, Property 7 holds. The global store $\mathit{globalstore}_0$ maps each $l \in S_g$ to its initial value $\mathit{initval}_l$ and $\mathit{globalstore}(E_2, \mathcal{T}_0)$ maps each $f \in S_{priv}$ to its initial value $\mathit{initval}_f$ (the empty string "" when $(x[\,], f) \in \mathsf{files}$ and the empty list $[\,]$ when $(\mathit{Tbl}, f) \in \mathsf{tables}$), so Properties 8, 9, and 10 hold. The module set $\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q_0)$ and the role set $\mathcal{RI}_0$ correspond by definition of $\mathcal{RI}_0$, so Property 11 holds. The event lists are empty on both sides, so Property 12 holds. The sets $\mathcal{O}^\infty(\mathcal{I})$ with $\mathcal{I} \stackrel{\text{def}}{=} \emptyset$ and $\mathcal{O}_{\mathsf{call}}(\mathcal{CS}_0)$ are both empty, so Property 13 holds. The sets $\mathcal{O}_{\mathsf{call\text{-}repl}}(\mathit{Th}_1^s)$, $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(\mathit{Th}_1^s))$, and $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathit{Th}_1^s))$ where Th_1^s is the current thread of $\mathcal{CS}_0$ are also empty, so Property 14 holds. We have $0 +$

$N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property 15 holds. We have $\alpha = 1$, so Property 16 holds. The set $\mathcal{I}$ is empty, so Property 17 holds. For all $\mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{RI}_0$, we have $a' = 1$, so Property 18 holds. Therefore, $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}$. □

The following two lemmas serve to prove Property 4 of the invariant.

Lemma F.1 *If* $E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_p E', P', \mathcal{Q}', \mathcal{T}', \mathcal{R}', \mathcal{E}'$, *and* $\mathsf{fv}(P) \cup \mathsf{fv}(\mathcal{Q}) \cup \mathsf{fv}(\mathcal{R}) \subseteq \mathrm{Dom}(E)$, *then* $\mathsf{fv}(P') \cup \mathsf{fv}(\mathcal{Q}') \cup \mathsf{fv}(\mathcal{R}') \subseteq \mathrm{Dom}(E')$.

Proof This result is easily proved by cases on the applied reduction rule. □

We denote by $\mathfrak{C}^{\mathsf{cs}} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{rest}$ an extended CryptoVerif configuration in which *rest* is either nothing or $\mathit{steps}, \mathcal{CS}$.

Lemma F.2 *If* $E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{rest} \rightsquigarrow_p E', P', \mathcal{Q}', \mathcal{T}', \mathcal{R}', \mathcal{E}', \mathit{rest}'$ *and* $\mathsf{fv}(P) \cup \mathsf{fv}(\mathcal{Q}) \cup \mathsf{fv}(\mathcal{R}) \subseteq \mathrm{Dom}(E)$, *then* $\mathsf{fv}(P') \cup \mathsf{fv}(\mathcal{Q}') \cup \mathsf{fv}(\mathcal{R}') \subseteq \mathrm{Dom}(E')$.

Proof This result is easily proved by cases on the applied reduction rule. By Lemma F.1, the rule (CryptoVerif) preserves the invariant. The rules (Enter Simulator) and (Simulator) leave the environment and the set of free variables unchanged. The rule (Leave Simulator) introduces a new free variable and adds it to the environment. □

The following lemma shows that a correct closure always remains correct during execution.

Lemma F.3 *Suppose that* $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$, $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}, \mathcal{I}$, $\mathcal{Q}'_0 \leftrightarrow \mathcal{RI}', \mathcal{I}'$, R *is an oracle reference of the form* $O'[\widetilde{a'}]$ *when oracle* O' *is not under replication and* $O'[\_, \widetilde{a''}]$ *when* O' *is under replication, and one of the following two situations occurs:*

1. $E' \supseteq E$, $\mathcal{I}' = \mathcal{I} - \{O[\widetilde{a}]\}$, $\mathcal{Q}'_0 \supseteq \mathcal{Q}_0 \setminus \{\mathcal{Q}_0(O[\widetilde{a}])\}$, $\tau'_{\mathsf{O}} = \tau_{\mathsf{O}}$, *and* l'_{tok} *is a restriction of* l_{tok} *such that, if* $R = O'[\widetilde{a'}]$, *then* $R \in \mathrm{Dom}(l'_{\mathsf{tok}})$.
2. $E' \supseteq E$, $\mathcal{I}' \supseteq \mathcal{I}$, $\mathcal{Q}'_0 \supseteq \mathcal{Q}_0$, $l'_{\mathsf{tok}} \supseteq l_{\mathsf{tok}}$, $\tau'_{\mathsf{O}} \supseteq \tau_{\mathsf{O}}$, *if* $R = O'[\widetilde{a'}]$, *then* $O'[\widetilde{a'}] \notin \mathcal{I}' \setminus \mathcal{I}$ *and* $O'[\widetilde{a'}] \in \mathrm{Dom}(l_{\mathsf{tok}})$, *and if* $R = O'[\_, \widetilde{a''}]$, *then for all* a, $O'\big[[a, +\infty[, \widetilde{a''}\big] \notin \mathcal{I}' \setminus \mathcal{I}$ *and* $O'[\_, \widetilde{a''}] \in \mathrm{Dom}(\tau_{\mathsf{O}})$.

Then $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

Proof Let us consider the first situation.

- Case $R = O[\widetilde{a}]$. Oracle O is not under replication. Since $\mathcal{I} - \{O[\widetilde{a}]\}$ is defined, we have $O[\widetilde{a}] \in \mathcal{I}$, and since $\mathcal{I}' = \mathcal{I} - \{O[\widetilde{a}]\}$, we have $O[\widetilde{a}] \notin \mathcal{I}'$. We also have $l'_{\mathsf{tok}}(O[\widetilde{a}]) = l_{\mathsf{tok}}(O[\widetilde{a}])$. So, by definition of $\mathsf{correctclosure}$,

$$\begin{aligned}
&\mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \\
&\quad = \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{false}}(Q)] \mid \\
&\qquad \text{for any } Q, env(token) = l'_{\mathsf{tok}}(O[\widetilde{a}])\} \\
&\quad \supseteq \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{false}}(\mathcal{Q}(O[\widetilde{a}]))] \mid \\
&\qquad env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}(O[\widetilde{a}])), env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\} \\
&\quad \supseteq \mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,.
\end{aligned}$$

- Case $R = O[\_, \widetilde{a''}]$ where $\widetilde{a} = a', \widetilde{a''}$ for some a'. Oracle O is under replication. Since $\mathcal{I} - \{O[\widetilde{a}]\}$ is defined, $O\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$, and since $\mathcal{I}' = \mathcal{I} - \{O[\widetilde{a}]\}$, we have $O\big[[a'+1, +\infty[, \widetilde{a''}\big] \in \mathcal{I}'$.

  Suppose that $a' < N_O$. Since $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}, \mathcal{I}$, there exist Q and i such that i does not occur in $\mathsf{fv}(Q)$, $\mathcal{Q}_0(O[a', \widetilde{a''}]) = Q\{a'/i\}$, and $\mathcal{Q}_0(O[a'+1, \widetilde{a''}]) = Q\{a'+1/i\}$. It is easy to see that $pm_{\mathrm{true}}(Q\{a'+1/i\}) = pm_{\mathrm{true}}(Q\{a'/i\})$, since the translation into OCaml does not depend on the indices. Moreover, $\mathsf{fv}(Q\{a'+1/i\}) = \mathsf{fv}(Q\{a'/i\})$ since i does not occur in $\mathsf{fv}(Q)$, so $env(E, Q\{a'+1/i\}) = env(E, Q\{a'/i\})$. Since $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$ and E' is an extension of E, we have $env(E', Q\{a'+1/i\}) = env(E, Q\{a'+1/i\})$. So, by definition of $\mathsf{correctclosure}$,

  $$\begin{aligned}
  &\mathsf{correctclosure}(O[_, \widetilde{a''}], \mathcal{I}', E', \mathcal{Q}_0', l_{\mathsf{tok}}', \tau_{\mathsf{O}}') \\
  &\quad = \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(\mathcal{Q}_0'(O[a'+1, \widetilde{a''}]))] \mid \\
  &\qquad \tau = \tau_{\mathsf{O}}'(O[_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E', \mathcal{Q}_0'(O[a'+1, \widetilde{a''}]))\} \\
  &\quad = \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(\mathcal{Q}_0(O[a', \widetilde{a''}]))] \mid \\
  &\qquad \tau = \tau_{\mathsf{O}}(O[_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}_0(O[a', \widetilde{a''}]))\} \\
  &\quad = \mathsf{correctclosure}(O[_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,.
  \end{aligned}$$

  Suppose that $a' = N_O$. By definition of $\mathsf{correctclosure}$,

  $$\begin{aligned}
  &\mathsf{correctclosure}(O[_, \widetilde{a''}], \mathcal{I}', E', \mathcal{Q}_0', l_{\mathsf{tok}}', \tau_{\mathsf{O}}') \\
  &\quad = \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(Q)] \mid \tau = \tau_{\mathsf{O}}'(O[_, \widetilde{a''}]), \text{for any } Q, env\} \\
  &\quad \supseteq \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(\mathcal{Q}_0(O[a', \widetilde{a''}]))] \mid \\
  &\qquad \tau = \tau_{\mathsf{O}}(O[_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}_0(O[a', \widetilde{a''}]))\} \\
  &\quad \supseteq \mathsf{correctclosure}(O[_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,.
  \end{aligned}$$

  Suppose that $a' > N_O$. We have $\mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}', E', \mathcal{Q}_0', l_{\mathsf{tok}}', \tau_{\mathsf{O}}') = \mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$ since $E, \mathcal{Q}_0, l_{\mathsf{tok}}$ are not used and $\tau_{\mathsf{O}}' = \tau_{\mathsf{O}}$.

- Other cases. All references to $\mathcal{Q}_0'(O'[\widetilde{a'}])$ in the definition of $\mathsf{correctclosure}$ satisfy $O'[\widetilde{a'}] \neq O[\widetilde{a}]$. In this case, we have $\mathcal{Q}_0'(O'[\widetilde{a'}]) = \mathcal{Q}_0(O'[\widetilde{a'}])$. Since $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$ and E' is an extension of E, we have $env(E', \mathcal{Q}_0'(O'[\widetilde{a'}])) = env(E', \mathcal{Q}_0(O'[\widetilde{a'}])) = env(E, \mathcal{Q}_0(O'[\widetilde{a'}]))$. Moreover, when $R = O'[\widetilde{a'}]$, we have $l_{\mathsf{tok}}'(O'[\widetilde{a'}]) = l_{\mathsf{tok}}(O'[\widetilde{a'}])$. Hence, by definition of $\mathsf{correctclosure}$, $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}_0', l_{\mathsf{tok}}', \tau_{\mathsf{O}}') = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

Let us now consider the second situation.

- Case $R = O'[\widetilde{a'}]$. Since $O'[\widetilde{a'}] \notin \mathcal{I}' \setminus \mathcal{I}$, we have $O'[\widetilde{a'}] \in \mathcal{I}'$ if and only if $O'[\widetilde{a'}] \in \mathcal{I}$. We have $l'_{\mathsf{tok}}(O'[\widetilde{a'}]) = l_{\mathsf{tok}}(O'[\widetilde{a'}])$.

  If $O'[\widetilde{a'}] \notin \mathcal{I}$, these points are sufficient to conclude that $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

  If $O'[\widetilde{a'}] \in \mathcal{I}$, there is an oracle $O'[\widetilde{a'}]$ in $\mathcal{Q}_0$; since $\mathcal{Q}'_0 \supseteq \mathcal{Q}_0$, we have $\mathcal{Q}'_0(O'[\widetilde{a'}]) = \mathcal{Q}_0(O'[\widetilde{a'}])$. Since $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$ and E' is an extension of E, we have $env(E', \mathcal{Q}'_0(O'[\widetilde{a'}])) = env(E', \mathcal{Q}_0(O'[\widetilde{a'}])) = env(E, \mathcal{Q}_0(O'[\widetilde{a'}]))$. So $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

- Case $R = O'[\_, \widetilde{a''}]$. Since for all a, $O'\big[[a, +\infty[, \widetilde{a''}\big] \notin \mathcal{I}' \setminus \mathcal{I}$, we have $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}'$ if and only if $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$.

  If there is no a' such that $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$, this point is sufficient to conclude that $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

  If $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$ and $a' \leq N_{O'}$, then there is an oracle $O'[a', \widetilde{a''}]$ in $\mathcal{Q}_0$; since $\mathcal{Q}'_0 \supseteq \mathcal{Q}_0$, we have $\mathcal{Q}'_0(O'[a', \widetilde{a''}]) = \mathcal{Q}_0(O'[a', \widetilde{a''}])$. Since $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$ and E' is an extension of E, we obtain $env(E', \mathcal{Q}'_0(O'[a', \widetilde{a''}])) = env(E', \mathcal{Q}_0(O'[a', \widetilde{a''}])) = env(E, \mathcal{Q}_0(O'[a', \widetilde{a''}]))$. Moreover, we have $\tau'_{\mathsf{O}}(O'[\_, \widetilde{a''}]) = \tau_{\mathsf{O}}(O'[\_, \widetilde{a''}])$. So $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

  If $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$ and $a' > N_{O'}$, then we have $\tau'_{\mathsf{O}}(O'[\_, \widetilde{a''}]) = \tau_{\mathsf{O}}(O'[\_, \widetilde{a''}])$, so $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. □

Let P'_{loop} be the process from line 8 to line 20 of Figure 19. Let P^j_{loop} be the process from line 13 to line 15 for the if $o = o_j$ branch. Let P^R_{loop} be the process from line 19 to line 20. The expansion of the let construct with pattern-matching introduces a fresh variable. Let us denote $xs[i']$ the variable created for the let matching on line 7, $xa_j[i']$ and $xi_j[i']$ the variables created on lines 11 and 12 for oracle number j.

Proof (of Lemma 6.29) Let us consider $\mathfrak{C}^{\mathsf{cs}}$ and $\mathcal{CT}$ such that $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. Let $\mathfrak{C}^{\mathsf{cs}} = E, P_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E}, steps, \mathcal{CS}$ and let $\mathcal{C}$ be the last configuration of $\mathcal{CT}$. Let

$$
\begin{aligned}
\mathcal{CS} &= ([Th_1, \ldots, Th_n], globalstore^s, tj), \mathcal{RI}, \mathcal{I}\,, \\
\mathcal{C} &= [Th'_1, \ldots, Th'_n], globalstore^o, tj, \mathcal{MI}, events\,, \\
Th_{tj} &= Th^s = \langle env^s, pe^s, stack^s, store^s \rangle\,, \\
Th'_{tj} &= Th^o = \langle env^o, pe^o, stack^o, store^o \rangle\,.
\end{aligned}
$$

We use the exponent s for the elements of the simulator configuration and the exponent o for the elements of the OCaml configuration.

Let us first distinguish cases depending on whether Property 6(a) or Property 6(b) is satisfied for the current thread.

Case 1. Property 6(a) is satisfied for the current thread, that is, we are at the beginning of the initialization of a protocol thread. There exists a program $program'$ such that

$$Th^s = \langle \emptyset, program_{\mathsf{prim}};; program'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots ;; program'(\mathsf{role}_m[\widetilde{a_m}]);; \\ program', [\,], \emptyset \rangle \,.$$

There is no closure, no tagged function $\mathsf{tagfunction}^t$ pm, no event, and no return in $program'$, except in $program(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$. The OCaml thread verifies $Th^o = \mathsf{replaceinitpm}(Th^s)$, so

$$Th^o = \langle \emptyset, program_{\mathsf{prim}};; program(\mu_{\mathsf{role}_1});; \ldots ;; program(\mu_{\mathsf{role}_m});; program', [\,], \emptyset \rangle \,.$$

By Assumption 6.2, there is exactly one complete thread trace $\mathcal{TT}$ that begins at $\langle \emptyset, program_{\mathsf{prim}};; , [\,], \emptyset \rangle$, and the last thread of this trace is $\langle env_{\mathsf{prim}}, \varepsilon, [\,], \emptyset \rangle$. So there is no call to the random function inside the initialization of the primitives. Let $\mathcal{TT}(definitions)$ be the trace $\mathcal{TT}$ where, in each thread, we replace the empty definition list ε by *definitions*. As no OCaml reduction rule depends on the contents of a definition list, the trace $\mathcal{TT}(definitions)$ is a valid trace for any definition list *definitions*. So, by taking *definitions* the definitions after $program_{\mathsf{prim}}$ in Th^s and Th^o,

$$Th^s \to^* Th^s_e \stackrel{\text{def}}{=} \langle env_{\mathsf{prim}}, program'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots ;; program'(\mathsf{role}_m[\widetilde{a_m}]);; \\ program', [\,], \emptyset \rangle$$

$$Th^o \to^* Th^o_e \stackrel{\text{def}}{=} \langle env_{\mathsf{prim}}, program(\mu_{\mathsf{role}_1});; \ldots ;; program(\mu_{\mathsf{role}_m});; program', [\,], \emptyset \rangle$$

in exactly the same number of steps.

Let l_j $(j \leq m)$ be m distinct locations. For $j \leq m+1$, let $Th^s_j \stackrel{\text{def}}{=} \langle env^s_j, program_j, [\,], store^s_j \rangle$ where $env^s_j \stackrel{\text{def}}{=} env_{\mathsf{prim}} \cup \{\mu_{\mathsf{role}_i}.\mathsf{init} \mapsto c_i \mid i < j\}$ with $c_i \stackrel{\text{def}}{=} \mathsf{tagfunction}^{\mathsf{role}_i,\tau_i}[env_i, pm'_{\mathsf{role}_i[\widetilde{a_i}]}]$ and $env_i \stackrel{\text{def}}{=} env^s_i[token \mapsto l_i]$, $program_j \stackrel{\text{def}}{=} program'(\mathsf{role}_j[\widetilde{a_j}]);; \ldots ;; program'(\mathsf{role}_m[\widetilde{a_m}]);; program'$ for $j \leq m$ and $program_j \stackrel{\text{def}}{=} program'$ for $j = m+1$, and $store^s_j \stackrel{\text{def}}{=} \{l_i \mapsto \text{true} \mid i < j\}$. For $j \leq m$, the thread Th^s_j reduces as follows:

$$\begin{aligned} Th^s_j &= \langle env^s_j, \mathsf{let}\ \mu_{\mathsf{role}_j}.\mathsf{init} = e^s_j;; program_{j+1}, [\,], store^s_j \rangle \\ &\quad \text{where } e^s_j \stackrel{\text{def}}{=} \mathsf{let}\ token = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}_j}\ pm'_{\mathsf{role}_j[\widetilde{a_j}]} \\ &\to \langle env^s_j, e^s_j, stack^s_j, store^s_j \rangle \\ &\quad \text{where } stack^s_j \stackrel{\text{def}}{=} [env^s_j, \mathsf{let}\ \mu_{\mathsf{role}_j}.\mathsf{init} = [\cdot];; program_{j+1}] \\ &\to^* \langle env_j, \mathsf{tagfunction}^{\mathsf{role}_j}\ pm'_{\mathsf{role}_j[\widetilde{a_j}]}, stack^s_j, store^s_{j+1} \rangle \\ &\quad \text{since } env_j = env^s_j[token \mapsto l_j] \text{ and } store^s_{j+1} = store^s_j[l_j \mapsto \text{true}] \end{aligned}$$

$$
\begin{aligned}
&\to \langle env_j, c_j, stack^s_j, store^s_{j+1}\rangle \\
\to^* \ Th^s_{j+1} &= \langle env^s_{j+1}, program_{j+1}, [\,], store^s_{j+1}\rangle \\
&\quad \text{since } env^s_{j+1} = env^s_j[\mu_{\mathsf{role}_j}.\mathsf{init} \mapsto c_j]
\end{aligned}
$$

Let $Th^o_j \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s_j)$. We have $Th^o_j = \langle env^o_j, program'_j, [\,], store^s_j\rangle$ where env^o_j is the environment env^s_j in which we replace $pm'_{\mathsf{role}_i[\widetilde{a_i}]}$ with $pm_{\mu_{\mathsf{role}_i}}$ for all $i < j$, $program'_j \stackrel{\text{def}}{=} program(\mu_{\mathsf{role}_j});; \ldots ;; program(\mu_{\mathsf{role}_m});; program'$ for $j \leq m$, and $program'_j \stackrel{\text{def}}{=} program'$ for $j = m+1$. For $j \leq m$, the thread Th^o_j reduces as follows:

$$
\begin{aligned}
Th^o_j &= \langle env^o_j, \mathsf{let}\ \mu_{\mathsf{role}_j}.\mathsf{init} = e^o_j;; program'_{j+1}, [\,], store^s_j\rangle \\
&\quad \text{where } e^o_j \stackrel{\text{def}}{=} \mathsf{let}\ token = \mathsf{ref}\ \mathrm{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}_j}\ pm_{\mu_{\mathsf{role}_j}} \\
&\to \langle env^o_j, e^o_j, stack^o_j, store^s_j\rangle \\
&\quad \text{where } stack^o_j \stackrel{\text{def}}{=} [env^o_j, \mathsf{let}\ \mu_{\mathsf{role}_j}.\mathsf{init} = [\cdot];; program'_{j+1}] \\
\to^* &\ \langle env'_j, \mathsf{tagfunction}^{\mathsf{role}_j}\ pm_{\mu_{\mathsf{role}_j}}, stack^o_j, store^s_{j+1}\rangle \\
&\quad \text{where } env'_j \stackrel{\text{def}}{=} env^o_j[token \mapsto l_j] \\
&\to \langle env'_j, c'_j, stack^o_j, store^s_{j+1}\rangle \\
&\quad \text{where } c'_j \stackrel{\text{def}}{=} \mathsf{tagfunction}^{\mathsf{role}_j, \tau_j}[env'_j, pm_{\mu_{\mathsf{role}_j}}] \\
\to^* \ Th^o_{j+1} &= \langle env^o_{j+1}, program'_{j+1}, [\,], store^s_{j+1}\rangle \\
&\quad \text{since } env^o_{j+1} = env^o_j[\mu_{\mathsf{role}_j}.\mathsf{init} \mapsto c'_j]
\end{aligned}
$$

Moreover, $Th^s_e = Th^s_1$ and $Th^o_e = Th^o_1$, so $Th^s \to^* Th^s_{m+1}$ and $Th^o \to^* Th^o_{m+1}$. There are exactly the same number of steps in both traces. Let $steps^s$ be this number of steps.

Let $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}[\mathsf{Th} \mapsto Th^o_{m+1}]$. Since $Th^s \to^* Th^s_{m+1}$ without using (Random), we have $\mathcal{CS} \to^* \mathcal{CS}[\mathsf{Th} \mapsto Th^s_{m+1}]$ by (Globalstore1), (Toplevel), and (Simulator toplevel). Furthermore, by definition of N_{steps}, all traces of the OCaml program have at most N_{steps} steps, so in particular $|\mathcal{CT}_1| = |\mathcal{CT}| + steps^s \leq N_{\mathsf{steps}}$. Hence, by Property 15, $steps \geq N_{\mathsf{steps}} - |\mathcal{CT}| \geq steps^s$. So, with $\mathfrak{C}^{\mathsf{cs}}_1 \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - steps^s, \mathcal{CS}[\mathsf{Th} \mapsto Th^s_{m+1}]$, we have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathfrak{C}^{\mathsf{cs}}_1$ by (Simulator) since $steps$ remains positive during the reduction. (More generally, the same reasoning shows that, if the simulator trace has at most as many steps as the OCaml trace, then the extended CryptoVerif configuration can reduce by (Simulator) because $steps$ remains positive by Property 15. We shall not detail this point in the other cases.)

Let us prove that $\mathfrak{C}^{\mathsf{cs}}_1 \equiv \mathcal{CT}_1$. Properties 1, 2, 3, 4, 7, 8, 9, 10, 11, 12 are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. As mentioned in Section 3.2.6, there are no local store locations in the initial program, so there are no local store locations in $program'$, so the locations $l_1, \ldots, l_m$ are the only local store locations present in Th^s_{m+1}, and they are all in the domain of $store^s_{m+1}$. So Property 5

holds. Let l_{tok} be the empty function. The set $\mathcal{O}_{\mathsf{call}}(Th^s_{m+1})$ is empty. We have that $\mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s_{m+1}), l_{\mathsf{tok}}) = \emptyset$ and $Th^o_{m+1} = \mathsf{replaceinitpm}(Th^s_{m+1}) \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s_{m+1}), \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so Property 6(b)i holds for the current thread. Using the function $l_{\mathsf{init\text{-}tok}}$ that maps $\mathsf{role}_j[\widetilde{a_j}]$ to l_j for $j \leq m$, Property 6(b)ii holds for the current thread. The environment of the tagged closures that we created contains env_{prim}, so Property 6(b)iii holds for the current thread. Since there is no tagged function, no event and no return in $program'$ except in $program(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$, Property 6(b)iv holds for the current thread. Threads that are not the current thread did not change, so Property 6 holds. The only change in the oracle sets is that the roles $\mathsf{role}_j[\widetilde{a_j}]$ are transferred from $\mathcal{R}_{\mathsf{init\text{-}function}}(Th^s)$ to $\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s)$, so Properties 13 and 14 are preserved. We have

$$|\mathcal{CT}_1| + steps - steps^s = |\mathcal{CT}| + steps^s + steps - steps^s \geq N_{\mathsf{steps}}$$

so Property 15 holds. Properties 16, 17, and 18 are preserved, because all components of these inequalities are unchanged. Therefore, we have proved that $\mathfrak{C}^{\mathsf{cs}}_1 \equiv \mathcal{CT}_1$.

Case 2. Property 6(b) is satisfied for the current thread. We now distinguish cases on the form of the simulator configuration $\mathcal{CS}$.

Case 2.1. The current expression of $\mathcal{CS}$ is $pe^s = \mathsf{call}(O_j[\widetilde{a}])\ (v_1, \ldots, v_{m_j})$ and $\mathcal{CS}$ cannot reduce, that is, the configuration $\mathcal{CS}$ makes a successful call to $O_j[\widetilde{a}]$, an oracle not under replication. By definition of $simreturn$, $simreturn(\mathcal{CS}) \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}), o_j, \widetilde{a}, args)$ where $args \stackrel{\text{def}}{=} (b_1, \ldots, b_{m_j})$ and $b_k \stackrel{\text{def}}{=} \mathbb{G}^{-1}_{\mathsf{val}T_{j,k}}(v_k)$ for $k \leq m_j$.

So $\mathfrak{C}^{\mathsf{cs}}$ reduces in several steps into the configuration $E_1, P'_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E}$ that corresponds to line 8 where

$$E_1 \stackrel{\text{def}}{=} E[xs[\alpha] \mapsto (\mathsf{repr}(\mathcal{CS}), o_j, \widetilde{a}, args), \\ s'[\alpha] \mapsto \mathsf{repr}(\mathcal{CS}), o[\alpha] \mapsto o_j, i[\alpha] \mapsto \widetilde{a}, args[\alpha] \mapsto args]\,.$$

Let $a'_1, \ldots, a'_{n_j} \stackrel{\text{def}}{=} \widetilde{a}$. As $E_1(o[\alpha]) = o_j$, this configuration reduces in several steps into the configuration $E_2, P^j_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E}$ where

$$E_2 \stackrel{\text{def}}{=} E_1[xa_j[\alpha] \mapsto args, a_{j,1}[\alpha] \mapsto b_1, \ldots, a_{j,m_j}[\alpha] \mapsto b_{m_j}, \\ xi_j[\alpha] \mapsto \widetilde{a}, i_{j,1}[\alpha] \mapsto a'_1, \ldots, i_{j,n_j}[\alpha] \mapsto a'_{n_j}]\,.$$

The oracle $O_j[\widetilde{a}]$ is in $\mathcal{I}$, otherwise $\mathcal{CS}$ could reduce using (FailedCall1). By Property 3 of the invariant, there exists $\mathcal{Q}_0$ such that $\mathcal{Q}_0 \leftrightarrow \mathcal{I}, \mathcal{RI}$ and $\mathcal{Q} = \{Q_{loop}\{a/i'\} \mid \alpha < a \leq N_{\mathsf{rand}+\mathsf{calls}}\} \cup \mathcal{Q}_0$. Let $Q \stackrel{\text{def}}{=} \mathcal{Q}_0(O_j[\widetilde{a}])$. The oracle definition Q is of the form $O_j[\widetilde{a}](x_1[\widetilde{a}] : T_{j,1}, \ldots, x_{m_j}[\widetilde{a}] : T_{j,m_j}) := P_O$. The previous configuration reduces in one step into $\mathfrak{C} \stackrel{\text{def}}{=} E_3, P_O, \mathcal{T}, \mathcal{Q}_1, \mathcal{R}_1, \mathcal{E}$ where

$$E_3 \stackrel{\text{def}}{=} E_2[x_1[\widetilde{a}] \mapsto b_1, \ldots, x_{m_j}[\widetilde{a}] \mapsto b_{m_j}]$$

$$\mathcal{R}_1 \stackrel{\text{def}}{=} ((r_{j,1},\ldots,r_{j,m'_j}), \mathsf{return}(simulate'_{O_j}(s',(r_{j,1},\ldots,r_{j,m'_j})),\text{true}),$$
$$\mathsf{return}(simulate''_{O_j}(s'),\text{true})) :: \mathcal{R}_{loop}(\alpha)$$
$$\mathcal{Q}_1 \stackrel{\text{def}}{=} \mathcal{Q} \setminus \{Q\}$$

Let us now look at $\mathcal{C}$. By the invariant, there exists an injection l_{tok} that satisfies Property 6(b)i. The current expression pe^o is of the form $c\ (v_1,\ldots,v_{m_j})$, where $c \in \mathsf{correctclosure}(O_j[\widetilde{a}],\mathcal{I},E,\mathcal{Q},l_{\mathsf{tok}},\tau_{\mathsf{O}})$, that is, $c = \mathsf{tagfunction}^{O_j,\tau}[env_1^o, pm_{\text{false}}(Q)]$ where $env_1^o \supseteq env_{\mathsf{prim}} \cup env(E,Q)$ and $env_1^o(token) = l_{\mathsf{tok}}(O_j[\widetilde{a}])$. By the same property, $store^o(l_{\mathsf{tok}}(O_j[\widetilde{a}])) = \text{true}$.

$$\begin{aligned}
Th^o &= \langle env^o, c\ (v_1,\ldots,v_{m_j}), stack^o, store^o\rangle \\
&\to \langle env_1^o, \mathsf{match}\ (v_1,\ldots,v_{m_j})\ \mathsf{with}\ pm_{\text{false}}(Q), stack^o, store^o\rangle \\
\to Th_1^o &\stackrel{\text{def}}{=} \langle env_2^o, e, stack^o, store^o\rangle \\
&\quad \text{where } env_2^o \stackrel{\text{def}}{=} env_1^o[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto v_1,\ldots,\mathbb{G}_{\mathsf{var}}(x_{m_j}) \mapsto v_{m_j}] \text{ and} \\
&\quad e \stackrel{\text{def}}{=} \mathsf{if}\ (!token)\ \&\& \\
&\qquad (\mathbb{G}_{\mathsf{pred}}(T_{j,1})\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \ldots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_{j,m_j})\ \mathbb{G}_{\mathsf{var}}(x_{m_j})) \\
&\quad\ \mathsf{then}\ (token := \text{false}; e')\ \mathsf{else\ raise\ Bad_Call} \\
&\quad e' \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]);\ldots;\mathbb{G}_{\mathsf{file}}(x_{m_j}[\widetilde{a}]);\mathbb{G}(P_O)
\end{aligned}$$

For all $k \le m_j$, there exists b_k such that $\mathbb{G}_{\mathsf{val}T_{j,k}}(b_k) = v_k$, so $\mathbb{G}_{\mathsf{pred}}(T_{j,k})\ \mathbb{G}_{\mathsf{var}}(x_k)$ evaluates to true using Proposition 6.5. Moreover, $env_2^o(token) = l_{\mathsf{tok}}(O_j[\widetilde{a}])$. So, the configuration $\mathcal{C}$ reduces as follows

$$\begin{aligned}
\mathcal{C} &\to^* \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto Th_1^o] \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env_2^o, token := \text{false}; e', stack^o, store_1^o\rangle] \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env_2^o, e', stack^o, store_2^o\rangle] \\
&\quad \text{where } store_2^o \stackrel{\text{def}}{=} store_1^o[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \text{false}] \\
\to^* \mathcal{C}_1 &\stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env_2^o, \mathbb{G}(P_O), stack^o, store_3^o\rangle, \mathsf{globalstore} \mapsto globalstore_1^o] \\
&\qquad\qquad \text{by Lemma D.1 applied } m_j \text{ times}
\end{aligned}$$

where $store_3^o \supseteq store_2^o$, $store_1^o \supseteq store^o$, $globalstore_1^o \supseteq globalstore(E_3,\mathcal{T})$ since $globalstore^o \supseteq globalstore(E,\mathcal{T})$ by Property 9 of the invariant, and $globalstore_1^o(l) = globalstore^o(l)$ for all $l \notin S_{priv}$.

We prove that for any traces $\mathfrak{CT}_1,\ldots,\mathfrak{CT}_m$ beginning at $\mathfrak{C}$ such that $\sum_{i\le m} \Pr[\mathfrak{CT}_i] = 1$, none of these traces is a prefix of another, and there is no intermediate configuration inside any of these traces with a return, end, call, or loop as current process, there exist m disjoint sets of OCaml traces $\mathcal{CTS}_1,\ldots,\mathcal{CTS}_m$ all starting at $\mathcal{C}_1$ such that none of these traces is a prefix of another of these traces, $\Pr[\mathcal{CTS}_i] = \Pr[\mathfrak{CT}_i]$ for all $i \le m$, and if $\mathcal{C}_4$ is the last configuration of a trace $\mathcal{CT}' \in \mathcal{CTS}_i$, then $\mathcal{C}_4 = \mathcal{C}[\mathsf{Th} \mapsto Th_4^o, \mathsf{globalstore} \mapsto$

$globalstore_4^o, \mathsf{events} \mapsto events_4]$ where

$$\begin{aligned} Th_4^o &= \langle env_4^o, \mathbb{G}(P_4), stack^o, store_4^o \rangle \text{ with} \\ &\quad env_4^o \supseteq env_{\mathsf{prim}} \cup env(E_4, P_4) \text{ and } store_4^o \supseteq store_3^o\,, \\ globalstore_4^o &\supseteq globalstore(E_4, \mathcal{T}_4)\,, \\ globalstore_4^o(l) &= globalstore^o(l) \text{ for all } l \notin S_{priv}\,, \\ events_4 &= \mathbb{G}_{\mathsf{ev}}(\mathcal{E}_4)\,, \end{aligned}$$

and the last configuration of $\mathfrak{CT}_i$ is $E_4, P_4, \mathcal{T}_4, \mathcal{Q}_1, \mathcal{R}_1, \mathcal{E}_4$.

The proof proceeds by induction on the total length of the traces $\mathfrak{CT}_1, \ldots,$ $\mathfrak{CT}_m$. In the base case, $m = 1$ and $\mathfrak{CT}_1$ is the trace that consists only of the configuration $\mathfrak{C}$. Let $\mathcal{CTS}_1$ consist of the single trace that contains just the configuration $\mathcal{C}_1$. We have $env_2^o \supseteq env_{\mathsf{prim}} \cup env(E_3, P_O)$ since $env^o \supseteq env_{\mathsf{prim}} \cup env(E, Q)$, the variables $x_1[\widetilde{a}], \ldots, x_{m_j}[\widetilde{a}]$ are added on the CryptoVerif side, and $\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_{m_j})$ are added correspondingly on the OCaml side. As shown above, $globalstore_1^o \supseteq globalstore(E_3, \mathcal{T})$ and $globalstore_1^o(l) = globalstore^o(l)$ for all $l \notin S_{priv}$. By Property 12 of the invariant, $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$. So the property holds for the base case. The inductive case follows from Lemma 6.10.

Let us take the maximal CryptoVerif traces $\mathfrak{CT}_1, \ldots, \mathfrak{CT}_n$ that begin at $\mathfrak{C}$ and that contain no return, end, call, or loop as current process in intermediate configurations. Let $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ the trace sets as defined above. The final configurations of the CryptoVerif traces $\mathfrak{CT}_i$ contain either return or end, since the oracle $O_j[\widetilde{a}]$ does not contain loop or call constructs. Let us take one such trace $\mathfrak{CT}_i$ and a trace $\mathcal{CT}' \in \mathcal{CTS}_i$. Let $\mathfrak{C}_4$ and $\mathcal{C}_4$ be the last configurations of $\mathfrak{CT}_i$ and $\mathcal{CT}'$ respectively. Let $\mathfrak{C}_4 = E_4, P_4, \mathcal{T}_4, \mathcal{Q}_1, \mathcal{R}_1, \mathcal{E}_4$. We distinguish cases on the form of P_4.

- If $P_4 = \mathsf{end}$,

$$\begin{aligned} \mathfrak{C}_4 &\rightsquigarrow E_4, \mathsf{return}(simulate''_{O_j}(s'[\alpha]), \text{true}), \mathcal{T}_4, \mathcal{Q}_1, \mathcal{R}_{loop}(\alpha), \mathcal{E}_4 \\ &\rightsquigarrow E_5, P_{return\text{-}loop}(\alpha), \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4 \\ &\quad \text{where } E_5 \stackrel{\text{def}}{=} E_4[r'_{\alpha,r}[\,] \mapsto s, b_{\alpha,r} \mapsto \text{true}], \\ &\qquad s \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{CS}'), \\ &\qquad \mathcal{CS}' \text{ is } \mathcal{CS} \text{ in which the current expression is replaced} \\ &\qquad \text{with } \mathsf{raise\ Match\_failure} \text{ and the set } \mathcal{I} \text{ is replaced with} \\ &\qquad \mathcal{I}' \stackrel{\text{def}}{=} \mathcal{I} - (O_j[\widetilde{a}]) \\ &\rightsquigarrow E_5, P_5, \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4 \\ &\quad \text{where } P_5 \stackrel{\text{def}}{=} \mathsf{let}\ r[\,] : T_{\mathcal{CS}} = \mathsf{loop}\ O_{\text{loop}}[\alpha + 1](r'_{\alpha,r}[\,]) \\ &\qquad\qquad \mathsf{in\ end\ else\ end} \\ &\rightsquigarrow E_5, P_6, \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4 \\ &\quad \text{where } P_6 \stackrel{\text{def}}{=} \mathsf{let}\ (r'_{\alpha+1,r}[\,] : T_{\mathcal{CS}}, b_{\alpha+1,r}[\,] : bool) = \\ &\qquad\qquad O_{\text{loop}}[\alpha + 1](r'_{\alpha,r})\ \mathsf{in}\ P_{return\text{-}loop}(\alpha + 1)\ \mathsf{else\ end} \end{aligned}$$

$\rightsquigarrow E_6, P_{loop}\{\alpha+1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{R}_{loop}(\alpha+1), \mathcal{E}_4$

where $E_6 \stackrel{\text{def}}{=} E_5[s[\alpha+1] \mapsto s]$,

$\mathcal{Q}_2 \stackrel{\text{def}}{=} \mathcal{Q}_1 \setminus \{Q_{loop}\{\alpha+1/i'\}\}$

(Let $\mathcal{CT}''$ be the trace $\mathcal{CT}$ followed by $\mathcal{CT}'$. By definition of $N_{\mathsf{rand+calls}}$, $N_{\mathsf{rand+calls}} \geq \left(N_{\mathsf{rand}}(\mathcal{CT}'') + \sum_{O,\tau} N_{\mathsf{calls}}(O,\tau,\mathcal{CT}'')\right) + 1 = \left(N_{\mathsf{rand}}(\mathcal{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O,\tau,\mathcal{CT})\right) + 2$ since $\mathcal{CT}''$ makes one more call to O_j than $\mathcal{CT}$. So, by Property 16, $N_{\mathsf{rand+calls}} \geq \alpha+1$. So, by Property 3, $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}$, so $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}_1$.)

$\rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E_6, P_{loop}\{\alpha+1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{R}_{loop}(\alpha+1), \mathcal{E}_4, N_{\mathsf{steps}}, \mathcal{CS}'$

By definition of the translation of end, the current expression of $\mathcal{C}_4$ is raise Match_failure. Let $\mathcal{CT}''$ be the trace $\mathcal{CT}$ followed by $\mathcal{CT}'$. The last configuration of $\mathcal{CT}''$ is $\mathcal{C}_4$.

Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$. By the form of $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{C}_4$, Properties 1 and 2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ where we removed the oracles $O_j[\widetilde{a}]$ and $O_{\text{loop}}[\alpha+1]$. We have $\mathcal{I}' = \mathcal{I} - (O_j[\widetilde{a}])$, so Property 3 is preserved. Property 4 is an immediate consequence of Lemma F.2. No new locations were created in the simulator, and the domains of stores can only grow, so Property 5 is preserved.

For all threads $tj' \neq tj$, the thread tj' does not change so, to prove Property 6, we just have to show that Property 6(b)i is preserved; the other elements of Property 6 are obviously preserved. Suppose that thread tj' satisfies Property 6(b)i initially, with a function l_{tok}. By Lemma F.3, Item 1, for all $\mathsf{call}(R)$ that occur in $Th''_{tj'} \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th_{tj'})$, we have $\mathsf{correctclosure}(R, \mathcal{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $\mathsf{replacecalls}(Th''_{tj'}, \mathcal{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathsf{replacecalls}(Th''_{tj'}, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Furthermore, the oracle $O_j[\widetilde{a}]$ is in $\mathcal{O}_{\mathsf{call}}(Th_{tj})$, so by Property 14 of the invariant, it is not in $\mathcal{O}_{\mathsf{call}}(Th_{tj'})$. Hence, $\mathcal{O}_{\mathsf{call}}(Th_{tj'}) \cap \mathcal{I}' = \mathcal{O}_{\mathsf{call}}(Th_{tj'}) \cap \mathcal{I}$ and $\mathcal{O}_{\mathsf{call}}(Th_{tj'}) \setminus \mathcal{I}' = \mathcal{O}_{\mathsf{call}}(Th_{tj'}) \setminus \mathcal{I}$, so $\mathsf{gettokens}(\mathcal{I}', \mathcal{O}_{\mathsf{call}}(Th_{tj'}), l_{\mathsf{tok}}) = \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th_{tj'}), l_{\mathsf{tok}})$. So thread tj' continues to verify Property 6(b)i with the same function l_{tok}.

Let us now consider the current thread. The current thread of the simulator is $Th_1^s = \langle env^s, \mathsf{raise\ Match\_failure}, stack^s, store^s\rangle$ and the current thread on the OCaml side is $Th_4^o = \langle env_4^o, \mathsf{raise\ Match\_failure}, stack^o, store_4^o\rangle$ where $store_4^o \supseteq store_3^o$. By Property 6(b)i, there exist $store_5^o$ and l_{tok} such that

$$\begin{aligned}&\langle env^o, pe^o, stack^o, store_5^o\rangle \\ &\qquad \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s), \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \\ &store_5^o \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}}) \subseteq store^o\,.\end{aligned}$$

Let us denote $Th'' \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s)$ and $Th_1'' \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th_1^s)$. The thread Th_1'' is the thread Th'' in which the current expression is

replaced with raise Match_failure. This is an exceptional value, so the definition of replacecalls allows any environment in the threads it returns, hence

$$\begin{aligned} Th_5^o \stackrel{\text{def}}{=} \langle env_4^o, \mathsf{raise\ Match\_failure}, stack^o, store_5^o\rangle \\ \in \mathsf{replacecalls}(Th_1'', \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,. \end{aligned}$$

Let $l'_{\mathsf{tok}} \stackrel{\text{def}}{=} l_{\mathsf{tok}|\mathcal{O}_{\mathsf{call}}(Th_1^s)}$. By Lemma F.3, Item 1, for all $\mathsf{call}(R)$ that occur in Th_1'', we have $\mathsf{correctclosure}(R, \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $Th_5^o \in \mathsf{replacecalls}(Th_1'', \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau_{\mathsf{O}})$. We have

$$\begin{aligned} &store_5^o \cup \mathsf{gettokens}(\mathcal{I}', \mathcal{O}_{\mathsf{call}}(Th_1^s), l'_{\mathsf{tok}}) \\ &\quad \subseteq store_5^o \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}})[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \text{false}] \\ &\quad \subseteq store^o[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \text{false}] \\ &\quad \subseteq store_1^o[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \text{false}] = store_2^o \subseteq store_3^o \subseteq store_4^o\,, \end{aligned}$$

so Property 6(b)i holds with the function l'_{tok}. Properties 6(b)ii, 6(b)iii, 6(b)iv are clearly preserved, so Property 6 holds.

Properties 7, 8, and 11 are also preserved. Properties 9, 10, and 12 hold because they are kept inside Lemma 6.10. We have $\mathcal{O}^\infty(\mathcal{I}') = \mathcal{O}^\infty(\mathcal{I}) \setminus \{O_j[\widetilde{a}]\}$. If there remains no occurrence of $\mathsf{call}(O_j[\widetilde{a}])$ in the thread Th_1^s, then $\mathcal{O}_{\mathsf{call}}(Th_1^s) = \mathcal{O}_{\mathsf{call}}(Th^s) \setminus \{O_j[\widetilde{a}]\}$ and $\mathcal{O}_{\mathsf{call}}(\mathcal{CS}') = \mathcal{O}_{\mathsf{call}}(\mathcal{CS}) \setminus \{O_j[\widetilde{a}]\}$, so $\mathcal{O}^\infty(\mathcal{I}') \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS}') = \mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS}) \setminus \{O_j[\widetilde{a}]\}$. Otherwise, $\mathcal{O}_{\mathsf{call}}(Th_1^s) = \mathcal{O}_{\mathsf{call}}(Th^s)$ and $\mathcal{O}_{\mathsf{call}}(\mathcal{CS}') = \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$, so $\mathcal{O}^\infty(\mathcal{I}') \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS}') = \mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$. We also have $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th_1^s) = \mathcal{O}_{\mathsf{call\text{-}repl}}(Th^s)$, $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th_1^s)) = \mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th^s))$, and $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th_1^s)) = \mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$, so Property 14 is preserved. The set $\mathcal{O}^\infty(\mathcal{I}') \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS}')$ is included in $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ and the set $\mathsf{willbeavailable}(\mathcal{CS}')$ is included in $\mathsf{willbeavailable}(\mathcal{CS})$, so Property 13 is preserved. We have $|\mathcal{CT}''| + N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property 15 holds. The number of calls to O_j increases by 1 and α increases by 1, so Property 16 is preserved. Properties 17 and 18 are preserved, because all components of these inequalities are unchanged. So $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$ in this case.

- If $P_4 = \mathsf{return}(M_1, \ldots, M_{m'_j}); Q'$, let c_i $(i \leq m'_j)$ be the CryptoVerif values such that $E_4, M_i \Downarrow c_i$.

$$\begin{aligned} \mathfrak{C}_4 &\rightsquigarrow E_5, \mathsf{return}(simulate'_{O_j}(s'[\alpha], (r_{j,1}[\alpha], \ldots, r_{j,m'_j}[\alpha])), \text{true}), \mathcal{T}_4, \mathcal{Q}_1, \\ &\qquad \mathcal{R}, \mathcal{E}_4 \\ &\quad \text{where } E_5 \stackrel{\text{def}}{=} E_4[r_{j,1} \mapsto c_1, \ldots, r_{j,m'_j} \mapsto c_{m'_j}] \\ \rightsquigarrow^* \mathfrak{C}_1^{\mathsf{cs}} &\stackrel{\text{def}}{=} E_6, P_{loop}\{\alpha + 1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{R}_{loop}(\alpha + 1), \mathcal{E}_4, N_{\mathsf{steps}}, \mathcal{CS}' \\ &\quad \text{where } E_6 \supseteq E_5[s[\alpha + 1] \mapsto \mathsf{repr}(\mathcal{CS}')], \\ &\qquad \mathcal{Q}_2 \stackrel{\text{def}}{=} \mathcal{Q}_1 \cup \text{reduce}(Q') \setminus \{Q_{loop}\{\alpha + 1/i'\}\} \\ &\qquad \mathsf{repr}(\mathcal{CS}') = simulate'_{O_j}(\mathsf{repr}(\mathcal{CS}), (c_1, \ldots, c_{j,m'_j})) \end{aligned}$$

where we show that $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}_1$ using Property 16 as in the case $P_4 = \mathsf{end}$.

Suppose that $\mathsf{reduce}'(Q') = [(Q_1, b_1), \ldots, (Q_l, b_l)]$ and $\mathsf{oracles}'(Q') = [O'_1[\widetilde{a_1}], \ldots, O'_l[\widetilde{a_l}]]$. A thread $\langle env, \mathbb{G}_{\mathsf{O}}(Q_i, b_i), stack, store\rangle$ where $env \supseteq env_{\mathsf{prim}} \cup env(E_4, P_4)$ reduces into $\langle env', c(Q_i, b_i), stack, store'\rangle$ where $c(Q_i, b_i) \stackrel{\text{def}}{=} \mathsf{tagfunction}^{O'_i,\tau_i}[env', pm_{b_i}(Q_i)]$ and

- if $b_i = \text{false}$, then $env' = env[token \mapsto l_i]$ and $store' = store[l_i \mapsto \text{true}]$ where l_i is a fresh location: $l_i \notin \text{Dom}(store)$;
- if $b_i = \text{true}$, then $env' = env$ and $store' = store$.

So in both cases, $env' \supseteq env_{\mathsf{prim}} \cup env(E_4, P_4)$.

Let $Th_4^o = \langle env_4^o, \mathbb{G}(P_4), stack^o, store_4^o\rangle$ be the current thread of $\mathcal{C}_4$. We have $env_4^o \supseteq env_{\mathsf{prim}} \cup env(E_4, P_4)$ and $store_4^o \supseteq store_3^o$.

$$\begin{aligned}
Th_4^o &= \langle env_4^o, (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_l, b_l), \mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{m'_j})),\\
&\qquad stack^o, store_4^o\rangle\\
&\to^* \langle env_4^o, (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_l, b_l), \mathbb{G}_{\mathsf{valT}'_{j,1}}(c_1), \ldots,\\
&\qquad \mathbb{G}_{\mathsf{valT}'_{j,m'_j}}(c_{m'_j})), stack^o, store_5^o\rangle\\
&\qquad\qquad \text{by Lemma 6.8 applied } m'_j \text{ times}\\
&\to^* Th_5^o \stackrel{\text{def}}{=} \langle env_4^o, (c(Q_1, b_1), \ldots, c(Q_l, b_l), \mathbb{G}_{\mathsf{valT}'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{valT}'_{j,m'_j}}(c_{m'_j})),\\
&\qquad stack^o, store_6^o\rangle
\end{aligned}$$

where $store_6^o \stackrel{\text{def}}{=} store_5^o \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} \supseteq store_5^o \supseteq store_4^o$.

Let $\mathcal{CT}''$ be the trace $\mathcal{CT}$ followed by $\mathcal{CT}'$ and extended until $\mathcal{C}_5 \stackrel{\text{def}}{=} \mathcal{C}_4[\mathsf{Th} \mapsto Th_5^o]$. Let $\mathcal{I}'$ be the set $\mathcal{I}$ of $\mathcal{CS}'$.

Let us now prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$. We define $\tau'_{\mathsf{O}} \stackrel{\text{def}}{=} \tau_{\mathsf{O}} \cup \{O'_i[\_, \widetilde{a}] \mapsto \tau_i \mid b_i = \text{true}\}$. (This function is well defined, because $O'_i[\_, \widetilde{a}] \notin \text{Dom}(\tau_{\mathsf{O}})$. Indeed, for any a', $O'_i[a', \widetilde{a}] \in \mathsf{willbeavailable}(\mathcal{CS})$, so by Property 13, $O'_i[a', \widetilde{a}] \notin \mathcal{O}^\infty(\mathcal{I})$, hence for any a', $O'_i\big[[a', +\infty[, \widetilde{a}\big] \notin \mathcal{I}$. The main reason why we introduced the set $\mathcal{O}^\infty(\mathcal{I})$ is that at this point, we are able to distinguish between an oracle under replication that has not been called yet and an oracle whose calls have been exhausted. If we used the set $\mathcal{O}(\mathcal{I})$ instead here, we would not be able to conclude that there is no oracle $O'_i\big[[a', +\infty[, \widetilde{a}\big]$ in $\mathcal{I}$: if $a' > N_{O'_i}$, then $\mathcal{O}(\{O'_i\big[[a', +\infty[, \widetilde{a}\big]\}) = \emptyset$.) By the form of $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{C}_5$, Properties 1 and 2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ where we removed the oracles $O_j[\widetilde{a}]$ and $O_{\text{loop}}[\alpha + 1]$ and where we added the new oracles $\mathsf{reduce}'(Q')$. By definition of $simulate'_{O_j}$, the set $\mathcal{I}'$ is the set $\mathcal{I}$ where we removed $O_j[\widetilde{a}]$ and added the elements of $\mathsf{oracles}'(Q')$. So Property 3 is preserved. We also have $\mathcal{Q}_2(O'_i[\widetilde{a_i}]) = Q_i$ for $i \leq l$. Property 4 is an immediate consequence of Lemma F.2. No new locations were created

in the simulator, and the domains of stores can only grow, so Property 5 is preserved.

For all threads $tj' \neq tj$, the thread tj' does not change so, to prove Property 6, we just have to show that Property 6(b)i is preserved; the other elements of Property 6 are obviously preserved. Suppose that thread tj' satisfies Property 6(b)i initially, with a function l_{tok}. By Lemma F.3, Items 1 and 2, for all $\mathsf{call}(R)$ that occur in $Th''_{tj'} \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th_{tj'})$, we have $\mathsf{correctclosure}(R, \mathcal{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $\mathsf{replacecalls}(Th''_{tj'}, \mathcal{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{replacecalls}(Th''_{tj'}, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Furthermore, the oracle $O_j[\widetilde{a}]$ is in $\mathcal{O}_{\mathsf{call}}(Th_{tj})$ and the oracles $O'_i[\widetilde{a_i}]$ that are not under replication are in $\mathsf{willbeavailable}(\mathcal{CS})$, so by Properties 13 and 14 of the invariant, they are not in $\mathcal{O}_{\mathsf{call}}(Th_{tj'})$. Hence, $\mathcal{O}_{\mathsf{call}}(Th_{tj'}) \cap \mathcal{I}' = \mathcal{O}_{\mathsf{call}}(Th_{tj'}) \cap \mathcal{I}$ and $\mathcal{O}_{\mathsf{call}}(Th_{tj'}) \setminus \mathcal{I}' = \mathcal{O}_{\mathsf{call}}(Th_{tj'}) \setminus \mathcal{I}$, so $\mathsf{gettokens}(\mathcal{I}', \mathcal{O}_{\mathsf{call}}(Th_{tj'}), l_{\mathsf{tok}}) = \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th_{tj'}), l_{\mathsf{tok}})$. So thread tj' continues to verify Property 6(b)i with the same function l_{tok}.

Let us now consider the current thread. The current thread of the simulator is $Th^s_1 = \langle env^s, pe^s_1, stack^s, store^s \rangle$ where $pe^s_1 \stackrel{\text{def}}{=} (\mathsf{call}(O'_1[\widetilde{a_1}]), \ldots, \mathsf{call}(O'_l[\widetilde{a_l}]), \mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$ and the current thread on the OCaml side is $Th^o_5 = \langle env^o_4, pe^o_6, stack^o, store^o_6 \rangle$ where $pe^o_6 \stackrel{\text{def}}{=} (c(Q_1, b_1), \ldots, c(Q_l, b_l), \mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$. By Property 6(b)i, there exist $store^o_7$ and l_{tok} such that

$$\begin{aligned}
&\langle env^o, pe^o, stack^o, store^o_7 \rangle \\
&\qquad \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s), \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \\
&store^o_7 \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}}) \subseteq store^o \,.
\end{aligned}$$

Let us denote $Th'' \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s)$ and $Th''_1 \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s_1)$. The thread Th''_1 is the thread Th'' where the current expression is replaced with pe^s_1. Let $Th^s_2 \stackrel{\text{def}}{=} \langle env^s, (), stack^s, store^s \rangle$ be a thread intermediate between Th^s and Th^s_1, in which the result of the call has not been inserted yet in the thread. When $b_i = \text{false}$, $O'_i[\widetilde{a_i}]$ is in $\mathsf{willbeavailable}(\mathcal{CS})$, so by Property 13, $O'_i[\widetilde{a_i}]$ is not in $\mathcal{O}_{\mathsf{call}}(Th^s)$, so we can define $l'_{\mathsf{tok}} \stackrel{\text{def}}{=} l_{\mathsf{tok}|\mathcal{O}_{\mathsf{call}}(Th^s_2)} \cup \{O'_i[\widetilde{a_i}] \mapsto l_i \mid b_i = \text{false}\}$ and l'_{tok} is an extension of $l_{\mathsf{tok}|\mathcal{O}_{\mathsf{call}}(Th^s_2)}$. By Lemma F.3, Items 1 and 2, for all $\mathsf{call}(R)$ that occur in $Th''_2 \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s_2)$, we have $\mathsf{correctclosure}(R, \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Moreover, $c(Q_i, b_i) \in \mathsf{correctclosure}(O'_i[\widetilde{a_i}], \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$ for $i \leq l$, and pe^s_1 is a value so $\mathsf{replacecalls}$ allows any environment in the threads it returns, so $\langle env^o_4,$

$pe_6^o, stack^o, store_7^o\rangle \in \mathsf{replacecalls}(Th_1'', \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$. We have

$$\begin{aligned}
&store_7^o \cup \mathsf{gettokens}(\mathcal{I}', \mathcal{O}_{\mathsf{call}}(Th_1^s), l'_{\mathsf{tok}}) \\
&\quad \subseteq store_7^o \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}})[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathrm{false}] \\
&\qquad \cup \{l_i \mapsto \mathrm{true} \mid b_i = \mathrm{false}\} \\
&\quad \subseteq store^o[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathrm{false}] \cup \{l_i \mapsto \mathrm{true} \mid b_i = \mathrm{false}\} \\
&\quad \subseteq store_1^o[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathrm{false}] \cup \{l_i \mapsto \mathrm{true} \mid b_i = \mathrm{false}\} \\
&\quad \subseteq store_2^o \cup \{l_i \mapsto \mathrm{true} \mid b_i = \mathrm{false}\} \\
&\quad \subseteq store_5^o \cup \{l_i \mapsto \mathrm{true} \mid b_i = \mathrm{false}\} = store_6^o\,,
\end{aligned}$$

so Property 6(b)i holds with the function l'_{tok}. Properties 6(b)ii, 6(b)iii, 6(b)iv are preserved, so Property 6 holds.

Properties 7, 8, and 11 are also preserved. Properties 9, 10, 12 hold because they are kept inside Lemma 6.10. The oracles coming from $\mathsf{oracles}'(Q')$ are removed from $\mathsf{willbeavailable}(\mathcal{CS})$ and added to $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}^\infty(\mathcal{I})$; it is also removed from $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ if there remains no occurrence of $\mathsf{call}(O_j[\widetilde{a}])$ in the thread Th_1^s. So Property 13 is preserved. The oracles coming from $\mathsf{oracles}'(Q')$ are added to $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ and to $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th_1^s)$ or $\mathcal{O}_{\mathsf{call}}(Th_1^s)$ depending on whether they are under replication or not. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}_{\mathsf{call}}(Th_1^s)$ if and only if it is removed from $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$. So Property 14 is preserved. We have $|\mathcal{CT}''| + N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property 15 holds. The number of calls to O_j increases by 1 and α increases by 1, so Property 16 is preserved. For the oracles $O'_i[\widetilde{a_i}]$ $(i \leq l)$, when O'_i is under replication, $O'_i\big[[1, +\infty[, \widetilde{a}\big]$ is added to $\mathcal{I}$; Property 17 is obviously satisfied for these oracles because the number of calls to these oracles is not negative. Property 17 for the previously present oracles and Property 18 are preserved, because all components of these inequalities are unchanged. So $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$ in this case.

- If $P_4 = \mathsf{return}(M_1, \ldots, M_{m'_j})\}; Q'$, the CryptoVerif process reduces in exactly the same manner as above. The configuration $\mathcal{CS}'$ is the configuration $\mathcal{CS}$ in which we replace the current expression pe^s with $(\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$, the set $\mathcal{I}$ with $\mathcal{I}' \stackrel{\text{def}}{=} \mathcal{I} \setminus \{O_j[\widetilde{a}]\}$, and the set $\mathcal{RI}$ with $\mathcal{RI}' \stackrel{\text{def}}{=} \mathcal{RI} \cup \{\mathsf{role}[\widetilde{a}] \mid (\mu_{\mathsf{role}}, \mathrm{false}) \in \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q')\} \cup \{\mathsf{role}\big[[1, +\infty[, \widetilde{a}\big] \mid (\mu_{\mathsf{role}}, \mathrm{true}) \in \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q')\}$.

  Let Th_4^o be the current thread of $\mathcal{C}_4$. We have

$$\begin{aligned}
Th_4^o = \langle &env_4^o, \mathsf{return}(\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q'), (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{m'_j}))), \\
&stack^o, store_4^o\rangle \\
\to \langle &env_4^o, (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{m'_j})), stack_1^o, store_4^o\rangle \\
&\text{where } stack_1^o \stackrel{\text{def}}{=} (env_4^o, \mathsf{return}(\mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q'), [\cdot])) :: stack^o
\end{aligned}$$

$\rightarrow^* Th_5^o \stackrel{\text{def}}{=} \langle env_4^o, (\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), stack_1^o, store_5^o \rangle$
by Lemma 6.8 applied m'_j times

where $store_5^o \supseteq store_4^o$ and $\mathcal{C}_4 \rightarrow^* \mathcal{C}_4[\mathsf{Th} \mapsto Th_5^o] \rightarrow^* \mathcal{C}_5 \stackrel{\text{def}}{=} \mathcal{C}_4[\mathsf{Th} \mapsto Th_6^o, \mathsf{MI} \mapsto \mathcal{MI}']$ where $Th_6^o \stackrel{\text{def}}{=} \langle env_4^o, (\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), stack^o, store_5^o \rangle$ and $\mathcal{MI}' \stackrel{\text{def}}{=} \mathcal{MI} \cup \mathbb{G}_{\mathsf{get}_{\mathcal{MI}}}(Q')$. Let $\mathcal{CT}''$ be the trace $\mathcal{CT}$ followed by $\mathcal{CT}'$ and extended until $\mathcal{C}_5$.

Let us now prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$. By the form of $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{C}''$, Properties 1 and 2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ from which we removed the oracles $O_j[\widetilde{a}]$ and $O_{\text{loop}}[\alpha + 1]$ and to which we added the new oracles of $\text{reduce}'(Q')$. The set $\mathcal{I}'$ is the set $\mathcal{I}$ from which we removed $O_j[\widetilde{a}]$. We added the elements of $\mathsf{oracles}'(Q')$ to $\mathcal{O}(\mathcal{RI}')$. So Property 3 is preserved. Properties 4 to 10, 12, and 15 to 18 are proved as in the case $P_4 = \mathsf{end}$. We added matching elements in $\mathcal{MI}'$ and in $\mathcal{RI}'$, so Property 11 is preserved. The oracles coming from $\mathsf{oracles}'(Q')$ are removed from $\mathsf{willbeavailable}(\mathcal{CS})$ and added to $\mathcal{O}^\infty(\mathcal{RI})$. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}^\infty(\mathcal{I})$; it is also removed from $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ if there remains no occurrence of $\mathsf{call}(O_j[\widetilde{a}])$ in the thread Th_1^s. So Property 13 is preserved. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}_{\mathsf{call}}(Th_1^s)$ if and only if it is removed from $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$, and the other oracle sets of Property 14 are unchanged, so Property 14 is preserved. So $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$ in this case.

Case 2.2. The current expression of $\mathcal{CS}$ is $pe^s = \mathsf{call}(O_j[\_, \widetilde{a}])\ (v_1, \ldots, v_{m_j})$ and $\mathcal{CS}$ cannot reduce, that is, the configuration $\mathcal{CS}$ makes a successful call to $O_j[\_, \widetilde{a}]$, an oracle under replication. We prove this case by a reasoning similar to the previous case.

We show that a copy of the oracle $O_j[\_, \widetilde{a}]$ is available in $\mathcal{Q}$ using Property 17, as follows. By Property 6(b)i, $pe^o = \mathsf{tagfunction}^{O_j,\tau}[env_1^o, pm_{\text{true}}(Q)]\ (v_1, \ldots, v_{m_j})$, with $\tau = \tau_{\mathsf{O}}(O_j[\_, \widetilde{a}])$ and $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$ for some a'. Let $\mathcal{CT}'$ be the extension of $\mathcal{CT}$ with one step. By definition of N_{O_j}, we have $N_{O_j} \geq N_{\mathsf{calls}}(O_j, \tau, \mathcal{CT}') = N_{\mathsf{calls}}(O_j, \tau, \mathcal{CT}) + 1$, so by Property 17, $a' \leq N_{O_j}$, so $O_j[a', \widetilde{a}] \in \mathcal{O}(\mathcal{I})$, so by Property 3, $\mathcal{Q}_0$ contains a process of the form $O_j[a', \widetilde{a}](x_1[a', \widetilde{a}] : T_{j,1}, \ldots, x_{m_j}[a', \widetilde{a}] : T_{j,m_j}) := P_O$.

Due to the call, the index a' such that $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$ increases by 1 and the number of calls to the closure with tag O_j, τ increases by 1, so Property 17 is preserved.

Case 2.3. The current expression of $\mathcal{CS}$ is $pe^s = \mathsf{random}\ ()$, that is, the configuration $\mathcal{CS}$ samples a random boolean. By Property 6, the current expression of $\mathcal{C}$ is $pe^o = \mathsf{random}\ ()$. For $b \in \{\text{true}, \text{false}\}$, $\mathcal{C} \rightarrow_{1/2} \mathcal{C}_b$ where $\mathcal{C}_b \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env^o, b, stack^o, store^o \rangle]$. Let $\mathcal{CT}_b$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_b$.

The configuration $\mathcal{CS}$ cannot reduce, and $simreturn(\mathcal{CS}) = (\mathsf{repr}(\mathcal{CS}), o_R, (), ())$. Let us denote $s \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{CS})$. The simulator configuration reduces in the

following way for a CryptoVerif value $b \in \{\text{true}, \text{false}\}$.

$$\begin{aligned}
\mathfrak{C}^{\mathsf{cs}} &\rightsquigarrow^* E_1, P'_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\quad \text{where } E_1 \stackrel{\text{def}}{=} E[xs[\alpha] \mapsto (s, o_R, (), ()), s'[\alpha] \mapsto s, o[\alpha] \mapsto o_R, \\
&\qquad\qquad i[\alpha] \mapsto (), args[\alpha] \mapsto ()] \\
&\rightsquigarrow^* E_1, P^R_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\rightsquigarrow_{1/2} E_2, \mathsf{return}(simulate_R(s'[\alpha], b_R[\alpha]), \text{true}), \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\quad \text{where } E_2 \stackrel{\text{def}}{=} E_1[b_R[\alpha] \mapsto b] \\
\rightsquigarrow^* \mathfrak{C}^{\mathsf{cs}}_b &\stackrel{\text{def}}{=} E_3, P_{loop}\{\alpha+1/i'\}, \mathcal{T}, \mathcal{Q}_1, \mathcal{R}_{loop}(\alpha+1), \mathcal{E}, N_{\mathsf{steps}}, \mathcal{CS}_b \\
&\quad \text{where } E_3 \supseteq E_2[s[\alpha+1] \mapsto \mathsf{repr}(\mathcal{CS}_b)], \\
&\qquad\qquad \mathcal{Q}_1 \stackrel{\text{def}}{=} \mathcal{Q} \setminus \{Q_{loop}\{\alpha+1/i'\}\}, \\
&\qquad\qquad \mathcal{CS}_b \stackrel{\text{def}}{=} \mathcal{CS}[\mathsf{Th} \mapsto \langle env^s, \mathbb{G}_{\mathsf{val}bool}(b), stack^s, store^s\rangle]
\end{aligned}$$

We verify that $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}$ using Property 16, as follows. By definition of $N_{\mathsf{rand+calls}}$, $N_{\mathsf{rand+calls}} \geq \left(N_{\mathsf{rand}}(\mathcal{CT}_b) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT}_b)\right) + 1 = \left(N_{\mathsf{rand}}(\mathcal{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT})\right) + 2$ since $\mathcal{CT}_b$ makes one more random number generation than $\mathcal{CT}$. So by Property 16, $N_{\mathsf{rand+calls}} \geq \alpha + 1$. So by Property 3, $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}$. Moreover, we have $\mathbb{G}_{\mathsf{val}bool}(b) = b$, so $\mathcal{CS}_b = \mathcal{CS}[\mathsf{Th} \mapsto \langle env^s, b, stack^s, store^s\rangle]$.

In this step, α becomes $\alpha + 1$, the number of random number generations in the trace increases by 1, the current thread is modified exactly in the same manner on both sides, and the other threads, the oracle sets, the global store, and the events are left unchanged, so it is easy to see that $\mathfrak{C}^{\mathsf{cs}}_{\text{true}} \equiv \mathcal{CT}_{\text{true}}$ and $\mathfrak{C}^{\mathsf{cs}}_{\text{false}} \equiv \mathcal{CT}_{\text{false}}$.

Case 2.4. The configuration $\mathcal{CS}$ does not reduce, and does not make a call to an oracle nor sample a random boolean. In this case, $simreturn(\mathcal{CS}) = (\mathsf{repr}(\mathcal{CS}), o_S, (), ())$. Let us denote $s \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{CS})$. The simulator configuration reduces in the following way.

$$\begin{aligned}
\mathfrak{C}^{\mathsf{cs}} &\rightsquigarrow^* E_1, P'_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\quad \text{where } E_1 \stackrel{\text{def}}{=} E[xs[\alpha] \mapsto (s, o_S, (), ()), s'[\alpha] \mapsto s, o[\alpha] \mapsto o_S, i[\alpha] \mapsto (), \\
&\qquad\qquad args[\alpha] \mapsto ()] \\
&\rightsquigarrow E_1, \mathsf{return}(s'[\alpha], \text{false}), \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\rightsquigarrow E_2, P_{return\text{-}loop}(\alpha), \mathcal{T}, \mathcal{Q}, \mathcal{R}_1, \mathcal{E} \\
&\quad \text{where } E_2 \stackrel{\text{def}}{=} E_1[r'_{\alpha,r}[\,] \mapsto s, b_{\alpha,r}[\,] \mapsto \text{false}], \\
&\qquad\qquad \mathcal{R}_1 \stackrel{\text{def}}{=} [x[\,], \mathsf{return}(x[\,]), \mathsf{end}] \\
&\rightsquigarrow E_2, r[\,] \leftarrow r'_{\alpha,r}[\,]; \mathsf{end}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_1, \mathcal{E} \\
&\rightsquigarrow E_3, \mathsf{end}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_1, \mathcal{E} \\
&\quad \text{where } E_3 \stackrel{\text{def}}{=} E_2[r[\,] \mapsto s]
\end{aligned}$$

$\rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E_3, \mathsf{end}, \mathcal{T}, \mathcal{Q}, [\,], \mathcal{E}$

This configuration cannot reduce. By Property 6, the OCaml configuration $\mathcal{C}$ also cannot reduce. (If it could reduce, then the simulator configuration $\mathcal{CS}$ would reduce by the same rule as the OCaml configuration.) Moreover, by Property 12 of the invariant, $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$, so this case satisfies the second point of Lemma 6.29.

Case 2.5. The current expression of $\mathcal{CS}$ is of the form $pe^s = \mathsf{tagfunction}^{t,\tau}[env, pm]\ v$, that is, $\mathcal{CS}$ calls a tagged closure. By Property 6(b)iii, the only tagged closures present in the current thread are of the form $\mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^s, pm'_{\mathsf{role}[\widetilde{a}]}]$ for a given role $\mathsf{role}[\widetilde{a}]$, with $env_{\mathsf{prim}} \subseteq env_1^s$. Such a closure corresponds to the initialization of the role $\mathsf{role}[\widetilde{a}]$. Since our programs are well-typed, and these closures expect an argument of type unit, the current expression of $\mathcal{CS}$ is $pe^s = \mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^s, pm'_{\mathsf{role}[\widetilde{a}]}]\ ()$.

Let us denote by Q_i, b_i for $i \leq m$ the oracles present in $\mathrm{reduce}'(Q(\mathsf{role})[\widetilde{a}])$, and let $\widetilde{a_i} = \widetilde{a}$ or $\_, \widetilde{a}$ such that $O'_i[\widetilde{a_i}]$ is the oracle associated to Q_i, b_i in $\mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])$.

By Property 6(b)i,

$$pe^o = \mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^o, pm_{\mu_{\mathsf{role}}}]\ ()\,.$$

By Property 6(b)ii, $env_1^s(token) = l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}])$ is a location. Let us denote l this location. By Property 6(b)i, we have $env_1^o(token) = env_1^s(token) = l$. By Property 5, l is in the domain of $store^s$. By Property 6(b)i, l is also in the domain of $store^o$.

Let $x_1[\,], \ldots, x_k[\,]$ be the free variables of the role role.

Let us denote

$$\begin{aligned} pe_e^s &\stackrel{\text{def}}{=} (\mathsf{call}(O'_1[\widetilde{a_1}]), \ldots, \mathsf{call}(O'_m[\widetilde{a_m}])) \\ pe_e^o &\stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{read}}(x_1[\,])\ \mathsf{in}\ \ldots\ \mathsf{in}\ \mathbb{G}_{\mathsf{read}}(x_k[\,])\ \mathsf{in} \\ &\qquad (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_m, b_m)) \end{aligned}$$

The simulator reduces as follows:

$$\begin{aligned} Th^s &= \langle env^s, \mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^s, pm'_{\mathsf{role}}]\ (), stack^s, store^s\rangle \\ &\rightarrow \langle env_1^s, \mathsf{match}\ ()\ \mathsf{with}\ pm'_{\mathsf{role}}, stack^s, store^s\rangle \\ \rightarrow Th_1^s &\stackrel{\text{def}}{=} \langle env_1^s, pe_1^s, stack^s, store^s\rangle \\ &\qquad \text{where } pe_1^s \stackrel{\text{def}}{=} \mathsf{if}\ !token\ \mathsf{then}\ (token := \mathrm{false}; pe_e^s)\ \mathsf{else\ raise\ Bad\_Call} \end{aligned}$$

and the OCaml side reduces as follows:

$$\begin{aligned} Th^o &= \langle env^o, \mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^o, pm_{\mu_{\mathsf{role}}}]\ (), stack^o, store^o\rangle \\ &\rightarrow \langle env_1^o, \mathsf{match}\ ()\ \mathsf{with}\ pm_{\mu_{\mathsf{role}}}, stack^o, store^o\rangle \\ \rightarrow Th_1^o &\stackrel{\text{def}}{=} \langle env_1^o, pe_1^o, stack^o, store^o\rangle \\ &\qquad \text{where } pe_1^o \stackrel{\text{def}}{=} \mathsf{if}\ !token\ \mathsf{then}\ (token := \mathrm{false}; pe_e^o)\ \mathsf{else\ raise\ Bad\_Call} \end{aligned}$$

- If $store^s(l) = \text{false}$, then by Property 6(b)i, $store^o(l) = \text{false}$, so

$$Th_1^s \to^* Th_2^s \stackrel{\text{def}}{=} \langle env_1^s, \mathsf{raise\ Bad\_Call}, stack^s, store^s \rangle$$
$$Th_1^o \to^* Th_2^o \stackrel{\text{def}}{=} \langle env_1^o, \mathsf{raise\ Bad\_Call}, stack^s, store^s \rangle$$

  Let $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}[\mathsf{Th} \mapsto Th_2^o]$, $\mathcal{CS}_1 \stackrel{\text{def}}{=} \mathcal{CS}[\mathsf{Th} \mapsto Th_2^s]$, $steps^s$ the number of steps of the trace $\mathcal{CS} \to^* \mathcal{CS}_1$, and $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - steps^s, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathfrak{C}_1^{\mathsf{cs}}$.

  Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. As the current expression is an exceptional value, replacecalls allows any environment in its image. Moreover, the other elements of the configuration are the same and $\mathcal{I}$ did not change, so Property 6 is preserved. The number of steps in the reduction is the same on both sides, so Property 15 is preserved. All other properties of Definition 6.27 are trivially inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

- Otherwise, $store^s(l) = \text{true}$. By Property 6(b)i, $store^o(l) = \text{true}$.

  On the simulator side:

$$Th_1^s \to^* Th_3^s \stackrel{\text{def}}{=} \langle env_1^s, pe_e^s, stack^s, store_1^s \rangle$$
$$\text{where } store_1^s \stackrel{\text{def}}{=} store^s[l \mapsto \text{false}]$$

  By Property 4, the variables $x_1[\,], \ldots, x_k[\,]$ are present in the environment E. Let $a'_1, \ldots, a'_k$ be the values of these variables in the environment E. By Property 9, $globalstore(E, \mathcal{T}) \subseteq globalstore^o$, so $globalstore^o(f_i) = \mathsf{ser}(T_{x_i}, a'_i)$ where $(x_i[\,], f_i) \in \mathsf{files}$ for all $i \leq k$. Let $\mathcal{C}_1 \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto Th_1^o]$. We have

$$\begin{aligned}
\mathcal{C}_1 &\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env_1^o, pe_e^o, stack^o, store_1^o \rangle] \\
&\quad \text{where } store_1^o \stackrel{\text{def}}{=} store^o[l \mapsto \text{false}] \\
&\to^* \mathcal{C}_2 \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env_2^o, pe_2^o, stack^o, store_2^o \rangle] \\
&\quad \text{where } pe_2^o \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_m, b_m)), \\
&\qquad env_2^o \stackrel{\text{def}}{=} env_1^o[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_{x_1}}(a'_1), \ldots, \\
&\qquad\qquad \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_{x_k}}(a'_k)], \\
&\qquad store_2^o \supseteq store_1^o
\end{aligned}$$

  by Proposition 6.5 applied k times to show the correctness of the deserialization primitives.

  Let $l_1, \ldots, l_m$ be pairwise distinct locations that are not in $\mathrm{Dom}(store_2^o)$ and $\tau_1, \ldots, \tau_m$ be pairwise distinct fresh tags. By the same reasoning as

in Case 2.1, sub-case $P_4 = \mathsf{return}(M_1, \ldots, M_{m'_j}); Q'$, we have

$$\begin{aligned}\mathcal{C}_2 \to^* \mathcal{C}_3 \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env_2^o, pe_3^o, stack^o, store_3^o\rangle] \\ \text{where } pe_3^o \stackrel{\text{def}}{=} (\mathsf{tagfunction}^{O'_1,\tau_1}[env_{c,1}^o, pm_{b_1}(Q_1)], \ldots, \\ \mathsf{tagfunction}^{O'_m,\tau_m}[env_{c,m}^o, pm_{b_m}(Q_m)]), \\ store_3^o \stackrel{\text{def}}{=} store_2^o \cup \{l_i \mapsto \text{true} \mid i \leq m, b_i = \text{false}\}\end{aligned}$$

where, for all $i \leq m$, $env_{c,i}^o$ is env_2^o when b_i is true and $env_2^o[token \mapsto l_i]$ otherwise.

Let $\mathcal{CT}_2$ be an extension of the trace $\mathcal{CT}$ until $\mathcal{C}_3$. Let $\mathcal{CS}_3 \stackrel{\text{def}}{=} \mathcal{CS}[\mathsf{Th} \mapsto Th_3^s]$. Let $steps^s$ be the number of steps of $\mathcal{CS} \to^* \mathcal{CS}_3$. Let $\mathfrak{C}_2^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - steps^s, \mathcal{CS}_3$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathfrak{C}_2^{\mathsf{cs}}$.

Let us prove that $\mathfrak{C}_2^{\mathsf{cs}} \equiv \mathcal{CT}_2$. We define τ'_{O} as τ_{O} except that for all $i \leq m$, if $b_i = \text{true}$, then $\tau'_{\mathsf{O}}(O'_i[\_, \widetilde{a}]) = \tau_i$. Properties 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 16, 18 hold because they hold for $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

Let us prove Property 6. First, we prove Property 6(b) for the current thread. For all $i \leq m$, the free variables of Q_i are contained in $\{x_1[], \ldots, x_k[]\}$, so $env_{c,i}^o \supseteq env(E, Q_i)$. Moreover, by Properties 6(b)iii and 6(b)i, $env_{\mathsf{prim}} \subseteq env_1^o$, so $env_{c,i}^o \supseteq env_2^o \supseteq env_1^o \supseteq env_{\mathsf{prim}}$. We have $\mathsf{role}[\widetilde{a}] \in \mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s)$. By Property 14, $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$ is included in $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$, and furthermore $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$ is disjoint from $\mathcal{O}_{\mathsf{call}}(Th_i)$ for all $i \leq n$, so from $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$, so $\mathcal{O}^\infty(\{\mathsf{role}[\widetilde{a}]\})$ is included in $\mathcal{O}^\infty(\mathcal{I})$. Hence, when O'_i is not under replication (that is, $b_i = \text{false}$), $O'_i[\widetilde{a_i}] \in \mathcal{I}$, and when O'_i is under replication, $\widetilde{a_i} = \_, \widetilde{a}$ and $O'_i[[1, +\infty[, \widetilde{a}] \in \mathcal{I}$. By Property 3, when O'_i is not under replication, $Q_i = \mathcal{Q}(O'_i[\widetilde{a_i}])$, and when O'_i is under replication, $Q_i = \mathcal{Q}(O'_i[1, \widetilde{a}])$.

By Property 6(b)i, there exist $store_4^o$ and l_{tok} such that

$$\begin{aligned}&\langle env^o, pe^o, stack^o, store_4^o\rangle \\ &\quad \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s), \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \\ &store_4^o \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}}) \subseteq store^o\,.\end{aligned}$$

Since $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$ is disjoint from $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ as noticed above, the oracles $O'_i[\widetilde{a_i}]$ are not present in $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$. So we can define the injective function $l'_{\mathsf{tok}} \stackrel{\text{def}}{=} l_{\mathsf{tok}} \cup \{O'_i[\widetilde{a_i}] \mapsto l_i \mid i \leq m, b_i = \text{false}\}$. By Lemma F.3, Item 2, for all $\mathsf{call}(R)$ that occur in $\mathsf{replaceinitpm}(Th^s)$, $\mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, noticing that, when $i \leq m$ and $b_i = \text{true}$, $O'_i[N_{O'_i} + 1, \widetilde{a}] \in \mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$, so by Property 14, $O'_i[N_{O'_i} + 1, \widetilde{a}] \notin \mathcal{O}_{\mathsf{call\text{-}repl}}(Th^s)$, so $\mathsf{call}(O'_i[\_, \widetilde{a}])$ does not occur in $\mathsf{replaceinitpm}(Th^s)$, so the transformation of τ_{O} into τ'_{O} does not affect the computation of these correct closures. Moreover, $\mathsf{tagfunction}^{O'_i,\tau_i}[env_{c,i}^o, pm_{b_i}(Q_i)] \in \mathsf{correctclosure}(O'_i[\widetilde{a_i}], \mathcal{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$ for $i \leq m$ and pe_e^s is a value so $\mathsf{replacecalls}$ allows any environment in the threads it returns,

so $\langle env_2^o, pe_3^o, stack^o, store_4^o[l \mapsto \text{false}]\rangle \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th_3^s), \mathcal{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$. We have

$$\begin{aligned}
&store_4^o[l \mapsto \text{false}] \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th_3^s), l'_{\mathsf{tok}}) \\
&\quad \subseteq store_4^o[l \mapsto \text{false}] \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}}) \\
&\qquad \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} \\
&\quad \subseteq store^o[l \mapsto \text{false}] \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} \\
&\quad \subseteq store_1^o \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} \\
&\quad \subseteq store_2^o \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} = store_3^o\,,
\end{aligned}$$

so Property 6(b)i holds with the function l'_{tok}. Properties 6(b)ii, 6(b)iii, 6(b)iv are preserved, so Property 6 holds for the current thread. The other threads and $\mathcal{I}, E, \mathcal{Q}$ are unchanged, and as above, the transformation of τ_{O} into τ'_{O} does not affect the computation of correct closures in these threads, so Property 6 holds for all threads.

The role $\mathsf{role}[\widetilde{a}]$ is removed from $\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s)$, so the elements added to $\mathcal{O}_{\mathsf{call}}(Th^s)$ and $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th^s)$ are removed from $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$, hence Properties 13 and 14 are preserved. There are more steps on the OCaml side than on the CryptoVerif side, so Property 15 is preserved. For the oracles $O'_i[\widetilde{a_i}]$ $(i \leq l)$, when O'_i is under replication, we have already shown that $O'_i\big[[1, +\infty[, \widetilde{a}\big] \in \mathcal{I}$; Property 17 is obviously satisfied for these oracles because the number of calls to these oracles is not negative. Property 17 is preserved for the other oracles, because all components of these inequalities are unchanged.

Case 2.6. The current expression of $\mathcal{CS}$ is of the form $pe^s = \mathsf{addthread}(program)$, that is, we add a new thread to the current configuration. By Property 6(b)i, the expression pe^o is $\mathsf{addthread}(program)$, and by Property 6(b)iv, *program* contains no closure, no tagged function, no event, no return except in parts $program(\mu_{\mathsf{role}})$, and in $program(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$.

Suppose first that *program* is an attacker program: it does not contain $program(\mu_{\mathsf{role}})$ except in arguments of $\mathsf{addthread}$. In this case,

$$\begin{aligned}
\mathcal{CS} \to \mathcal{CS}_1 \stackrel{\text{def}}{=} ([&Th_1, \ldots, Th_{tj-1}, \langle env^s, (), stack^s, store^s\rangle, Th_{tj+1}, \ldots, Th_n, \\
&\langle \emptyset, program, [\,], \emptyset\rangle], globalstore^s, tj), \mathcal{RI}, \mathcal{I} \\
\mathcal{C} \to \mathcal{C}_1 \stackrel{\text{def}}{=} [&Th'_1, \ldots, Th'_{tj-1}, \langle env^o, (), stack^o, store^o\rangle, Th_{tj+1}, \ldots, Th_n, \\
&\langle \emptyset, program, [\,], \emptyset\rangle], globalstore^o, tj, \mathcal{MI}, events
\end{aligned}$$

Let $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_1$ and $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - 1, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}}$. Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. The new thread contains no closures and no tagged functions. It contains no call since *program* is an OCaml program (not a simulator program), so it satisfies Property 6(b). The other properties are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

Otherwise, the program *program* is of the form

$$program_{\mathsf{prim}};; program(\mu_{\mathsf{role}_1});; \ldots ;; program(\mu_{\mathsf{role}_m});; program' ,$$

where $program'$ does not contain $program(\mu)$ for any $\mu \in \mathcal{M}_g$. By Assumption 4.1, for $\mathcal{M} \stackrel{\text{def}}{=} \{\mu_{\mathsf{role}_1}, \ldots, \mu_{\mathsf{role}_m}\}$, we have $\mathcal{M} \subseteq \mathcal{M}_g$ and $\forall \mu \in \mathcal{M}, \exists b$, $(\mu, b) \in \mathcal{MI}$. By Property 11, for each $i \leq m$, if role_i is not under replication, then the set $\mathcal{RI}$ contains $\mathsf{role}_i[\widetilde{a}]$ for some $\widetilde{a}$, and if role_i is under replication, then the set $\mathcal{RI}$ contains $\mathsf{role}_i\big[[a', +\infty[, \widetilde{a}\big]$ for some $a', \widetilde{a}$. By Property 13, the oracles present in $\mathcal{RI}$ are not in $\mathcal{I}$. We can then define

$$\begin{aligned}
\widetilde{a_1} &\stackrel{\text{def}}{=} \mathsf{smallest}(\mathcal{RI}, \mathsf{role}_1), \ldots, \widetilde{a_m} \stackrel{\text{def}}{=} \mathsf{smallest}(\mathcal{RI}, \mathsf{role}_m) \\
\mathcal{RI}'' &\stackrel{\text{def}}{=} \{\mathsf{role}_1[\widetilde{a_1}], \ldots, \mathsf{role}_m[\widetilde{a_m}]\} \\
\mathcal{RI}' &\stackrel{\text{def}}{=} \mathcal{RI} - \mathcal{RI}'' \qquad \mathcal{I}' \stackrel{\text{def}}{=} \mathsf{add}(\mathcal{I}, \mathcal{RI}'') \\
program^b &\stackrel{\text{def}}{=} program_{\mathsf{prim}};; program'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots ;; program'(\mathsf{role}_m[\widetilde{a_m}]);; \\
&\qquad program' \\
\mathcal{MI}' &\stackrel{\text{def}}{=} \{(\mu, \text{false}) \mid \mu \in \mathcal{M} \wedge (\mu, \text{false}) \in \mathcal{MI}\}
\end{aligned}$$

We have

$$\begin{aligned}
\mathcal{CS} \to \mathcal{CS}_2 \stackrel{\text{def}}{=} ([&Th_1, \ldots, Th_{tj-1}, \langle env^s, (), stack^s, store^s\rangle, Th_{tj+1}, \ldots, Th_n, \\
&\langle \emptyset, program^b, [\,], \emptyset\rangle], globalstore^s, tj), \mathcal{RI}', \mathcal{I}' \\
\mathcal{C} \to \mathcal{C}_2 \stackrel{\text{def}}{=} [&Th'_1, \ldots, Th'_{tj-1}, \langle env^o, (), stack^o, store^o\rangle, Th_{tj+1}, \ldots, Th_n, \\
&\langle \emptyset, program, [\,], \emptyset\rangle], globalstore^o, tj, \mathcal{MI} \setminus \mathcal{MI}', events
\end{aligned}$$

Let $\mathcal{CT}_2$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_2$ and $\mathfrak{C}_2^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}$, $\mathcal{E}, steps - 1, \mathcal{CS}_2$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_2^{\mathsf{cs}}$. Let us prove that $\mathfrak{C}_2^{\mathsf{cs}} \equiv \mathcal{CT}_2$. The oracles under replication added to $\mathcal{I}$ are the oracles $O\big[[1, +\infty[, \widetilde{a_i}\big]$ such that $O[\_, \widetilde{a_i}] \in \mathsf{oracles}'(Q(\mathsf{role}_i)[\widetilde{a_i}])$ for any $i \leq m$. We define τ'_{O} as the extension of τ_{O} that maps all the oracles $O[\_, \widetilde{a_i}]$ to fresh distinct tags τ. Properties 1, 2, 4, 8, 9, 10, 12, 15, and 16 are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. By Property 3, $\mathcal{Q} = \{Q_{loop}\{a/i'\} \mid \alpha < a \leq N_{\mathsf{rand}+\mathsf{calls}}\} \cup \mathcal{Q}_0$ and $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}, \mathcal{I}$. If role_i is under replication, then by definition of $\mathsf{smallest}$, $\mathsf{role}_i\big[[a'_i, +\infty[, \widetilde{a''_i}\big] \in \mathcal{RI}$ with $\widetilde{a_i} = a'_i, \widetilde{a''_i}$. By definition of N_{role_i}, $N_{\mathsf{role}_i} \geq N_{\mathsf{exec}}(\mathsf{role}_i, \mathcal{CT}_2) = N_{\mathsf{exec}}(\mathsf{role}_i, \mathcal{CT}) + 1$. By Property 18, $a'_i \leq N_{\mathsf{role}_i}$. Therefore, the set $\mathcal{O}(\mathcal{RI})$ contains the first oracles of $\mathsf{role}_i[\widetilde{a_i}]$ for $i \leq m$. The set $\mathcal{O}(\mathcal{RI}')$ is the set $\mathcal{O}(\mathcal{RI})$ from which we remove the first oracles of $\mathsf{role}_i[\widetilde{a_i}]$ for $i \leq m$ and $\mathcal{O}(\mathcal{I}')$ is the set $\mathcal{O}(\mathcal{I})$ to which we add these oracles. So $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}', \mathcal{I}'$ and Property 3 holds. There are no local store locations in *program*, so Property 5 holds. For each thread Th_i of the simulator except the new thread, let us show that Property 6 is preserved. The only changes are that the current expression is replaced with () and that $\mathcal{I}' = \mathsf{add}(\mathcal{I}, \mathcal{RI}'')$, so we just have to show that Property 6(b)i is preserved; the other elements of Property 6 are obviously preserved. By Lemma F.3, Item 2, the correct closures

are preserved. By Property 14, the set $\mathcal{O}_{\mathsf{call}}(Th_i)$ does not contain any of the oracles added to $\mathcal{I}$, so the tokens are preserved. Hence, Property 6(b)i is preserved. Since $program'$ already occurs in the initial program, it does not contain closures. By Property 6(b)iv, it does not contain tagged functions, events, or returns, except in $program(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$, so Property 6(a) holds for the new thread. By Property 7, $program'$ does not contain any location $l \in S_{priv}$ except in $program(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$, so Property 7 holds. When role_i is not under replication, we remove one copy of the module μ_{role_i} from the multiset $\mathcal{MI}$, and correspondingly, we remove $\mathsf{role}_i[\widetilde{a_i}]$ from $\mathcal{RI}$. When role_i is under replication, we add 1 to the first index of roles $\mathsf{role}_i[\widetilde{a_i}]$ in $\mathcal{RI}$, and $\mathcal{MI}$ is not affected by this change. (The role role_i can still be called.) So Property 11 is preserved. The first oracles of $\mathsf{role}_1[\widetilde{a_1}], \ldots, \mathsf{role}_m[\widetilde{a_m}]$ are transferred from $\mathcal{O}^\infty(\mathcal{RI})$ to $\mathcal{O}^\infty(\mathcal{I})$, so Property 13 is preserved. More precisely, these oracles are added to $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th_{n+1}))$, where $Th_{n+1} \stackrel{\text{def}}{=} \langle \emptyset, program^b, [\,], \emptyset \rangle$ is the new thread, so Property 14 is preserved. For the oracles $O\big[[1, +\infty[, \widetilde{a_i}\big]$ added to $\mathcal{I}$, Property 17 is obviously satisfied because the number of calls to an oracle is not negative. Property 17 is preserved for the previously present oracles, because all components of these inequalities are unchanged. For the roles $\mathsf{role}_i\big[[a'_i, +\infty[, \widetilde{a''_i}\big] \in \mathcal{RI}$, with $\widetilde{a_i} = a'_i, \widetilde{a''_i}$, we have $\mathsf{role}_i\big[[a'_i + 1, +\infty[, \widetilde{a''_i}\big] \in \mathcal{RI}$; the elements $\mathsf{role}_i[\ldots]$ in $\mathcal{RI}$ with indices that do not end with $\widetilde{a''_i}$ are unchanged; and $N_{\mathsf{exec}}(\mathsf{role}_i, \mathcal{CT})$ increases by 1, so Property 18 is preserved for the roles $\mathsf{role}_1, \ldots, \mathsf{role}_m$. Property 18 is preserved for the other roles, because all components of the inequalities are unchanged. Therefore, $\mathfrak{C}_2^{\mathsf{cs}} \equiv \mathcal{CT}_2$.

Case 2.7. The current expression of $\mathcal{CS}$ is of the form $pe^s = \mathsf{call}(O[\widetilde{a}])\ v$ and $\mathcal{CS}$ reduces, that is, we call an oracle but the call fails. By reduction rule (FailedCall1) or (FailedCall2),

$$Th^s \to Th_1^s \stackrel{\text{def}}{=} \langle env^s, \mathsf{raise\ Bad\_Call}, stack^s, store^s \rangle,$$

and $\mathcal{CS} \to \mathcal{CS}_1 \stackrel{\text{def}}{=} \mathcal{CS}[\mathsf{Th} \mapsto Th_1^s]$.

By Property 6(b)i, $pe^o = c\ v'$, where $c \in \mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

We suppose that the program is well typed, so the value v is a k-tuple $(v_1, \ldots, v_k)$, where k is the number of arguments of oracle O. Let $T_1, \ldots, T_k$ be their CryptoVerif types. Let $x_1, \ldots, x_k$ be the CryptoVerif variables that are the arguments of O. By Assumption 6.3, the value v' does not contain closures nor locations, so $v' = v$.

Let us first suppose that the oracle O is under replication. In this case, $\widetilde{a} = \_, \widetilde{a'}$. There exists a'' such that $O\big[[a'', +\infty[, \widetilde{a'}\big] \in \mathcal{I}$, because otherwise we would have $\mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \emptyset$. The closure c is of the form $\mathsf{tagfunction}^{O,\tau}[env_1^o, pm_{\mathrm{true}}(Q)]$. Let $\mathcal{CT}'$ be the extension of $\mathcal{CT}$ by one step. By definition of N_O, $N_O \geq N_{\mathsf{calls}}(O, \tau, \mathcal{CT}') = N_{\mathsf{calls}}(O, \tau, \mathcal{CT}) + 1$. Hence, by Property (17), $a'' \leq N_O$. Therefore, by definition of $\mathsf{correctclosure}$, $Q = \mathcal{Q}(O[a'', \widetilde{a'}])$. Since (FailedCall2) applies, there exists i such that $\forall a \in T_i$, $v_i \neq \mathbb{G}_{\mathsf{val}T_i}(a)$. By Proposition 6.5, for any environment env, stack $stack$ and

store *store*,

$$\begin{array}{l}\langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{pred}}(T_i))\ v_i, stack, store\rangle\\ \rightarrow^* \langle env', \text{false}, stack, store'\rangle \text{ where } store' \supseteq store\end{array}$$

So,

$$\begin{array}{l}Th^o = \langle env^o, \mathsf{tagfunction}^{O,\tau}[env_1^o, pm_{\text{true}}(Q)]\ v, stack^o, store^o\rangle\\ \quad\rightarrow^* \langle env_2^o, pe_2^o, stack^o, store^o\rangle\\ \qquad \text{where } env_2^o \stackrel{\text{def}}{=} env_1^o[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto v_1, \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto v_k],\\ \qquad\qquad pe_2^o \stackrel{\text{def}}{=} \mathsf{if}\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \ldots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k))\\ \qquad\qquad\qquad \mathsf{then}\ (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P))\\ \qquad\qquad\qquad \mathsf{else\ raise\ Bad\_Call}\\ \rightarrow^*\ Th_1^o \stackrel{\text{def}}{=} \langle env_2^o, \mathsf{raise\ Bad\_Call}, stack^o, store_1^o\rangle\end{array}$$

where $store_1^o \supseteq store^o$ by Proposition 6.5 applied k times.

If the oracle O is not under replication, then (FailedCall1) applies, so either $O[\widetilde{a}] \notin \mathcal{I}$ and in this case by Property 6(b)i, $store^o[l_{\mathsf{tok}}(O[\widetilde{a}])] = \text{false}$, or there exists i such that $\forall a \in T_i$, $v_i \neq \mathbb{G}_{\mathsf{val}T_i}(a)$, so we have a reduction similar to the case in which O is under replication.

Let $\mathcal{CT}_1$ be an extension of the trace $\mathcal{CT}$ until $\mathcal{C}[\mathsf{Th} \mapsto Th_1^o]$ and $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps-1, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}}$. Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. The current expression is an exceptional value, so replacecalls allows any environment in the current thread, and $store_1^o \supseteq store^o$, so Property 6(b)i is preserved for the current thread. The OCaml side uses more reductions than the simulator side, so Property 15 is preserved. There is one more oracle call, and α and $\mathcal{I}$ are unchanged, so Properties 16 and 17 are preserved. The other properties are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

Case 2.8. Let us finally deal with the remaining cases. Cases 2.1, 2.2, 2.3, 2.4 present all cases in which $\mathcal{CS}$ does not reduce. Case 2.6 covers the reduction rule (Simulator add thread). So $\mathcal{CS}$ reduces using rule (Simulator toplevel). So let us denote $\mathcal{CS}_1$ the configuration such that $\mathcal{CS} = \mathcal{C}^s, \mathcal{RI}, \mathcal{I} \rightarrow \mathcal{CS}_1 = \mathcal{C}_1^s, \mathcal{RI}, \mathcal{I}$. Since the case of failed oracle calls is already handled in Case 2.7, $\mathcal{C}^s \rightarrow \mathcal{C}_1^s$ is obtained by rules of the OCaml semantics, not by (FailedCall1) or (FailedCall2).

If $pe^s = \mathsf{schedule}(tj')$, then by Property 6(b)i, $pe^o = \mathsf{schedule}(tj')$, so $\mathcal{C}^s$ and $\mathcal{C}$ reduce in the same way using rules (Toplevel schedule1) or (Toplevel schedule2) for $\mathcal{C}^s$ and (New toplevel schedule1) or (New toplevel schedule2) for $\mathcal{C}$. Let $\mathcal{C}_1$ be the configuration such that $\mathcal{C} \rightarrow \mathcal{C}_1$ and $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_1$. Let $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps-1, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}}$ and $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$.

In all other cases, $\mathcal{C}^s$ reduces by (Toplevel). By Property 6(b)i, the current thread of the OCaml configuration has the same form as in the simulator

configuration: the semantic rules are parametric in the elements that are replaced by replaceinitpm and replacecalls, so the OCaml configuration $\mathcal{C}$ reduces by (New toplevel), using a reduction $Th, globalstore \longrightarrow_p Th', globalstore'$ obtained by exactly the same semantic rules as on the simulator side. Let $\mathcal{C}_1$ be the configuration such that $\mathcal{C} \to \mathcal{C}_1$ and $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_1$. Let $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - 1, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}}$ and we briefly show that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$.

If the reduction touches a local store location l, then by Properties 6(b)i and 6(b)ii, l cannot be in the image of l_{tok} or $l_{\mathsf{init\text{-}tok}}$. Moreover, in all cases, the reduction commutes with replaceinitpm and replacecalls, so Property 6 holds for $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{CT}_1$. (Since calls to tagged closures are already handled in Case 2.5, we do not consider this case here. This is important, because the reduction would not commute with replaceinitpm in this case: replaceinitpm replaces the pattern-matching inside the tagged closure before the call, but would not replace it in the reduced configuration.) If the reduction touches the global store, that is, it uses rule (Globalstore2), let l be the concerned location; by Property 7, the location l is not in S_{priv}, and in OCaml the same operation is carried out on l. So in all cases, Properties 7, 8, 9, and 10 hold for $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{CT}_1$. The oracle sets may only decrease, in case a subexpression is removed by reduction, so Properties 13 and 14 are preserved. The reduction is performed in one step on both sides, so Property 15 is preserved. The other properties are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$, so $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. □

G Proof of Proposition 6.30

Definition G.1 *The relation* $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ *is verified when the following properties hold:*

1. *All traces* $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$ *start at* $\mathfrak{C}_0(Q_0, program_0)$*, and none of these traces is a prefix of another of these traces.*
2. *The trace sets* $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ *are pairwise disjoint, all traces in these sets start at* $\mathcal{C}_0(Q_0, program_0)$*, and none of these traces is a prefix of another of these traces.*
3. $\forall i \leq n, \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] = \Pr[\mathcal{CTS}_i]$.
4. $\sum_{i \leq n} \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] = 1$.
5. *For each trace* $\mathfrak{CT}_i^{\mathsf{cs}}$*,* $i \leq n$*,*
   (a) *either* $\mathfrak{CT}_i^{\mathsf{cs}}$ *is complete, every trace* $\mathcal{CT} \in \mathcal{CTS}_i$ *is complete, and the event list* $\mathcal{E}$ *of the last configuration of* $\mathfrak{CT}_i^{\mathsf{cs}}$ *and the event list events of the last configuration of* $\mathcal{CT}$ *verify* $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$*,*
   (b) *or for every trace* $\mathcal{CT} \in \mathcal{CTS}_i$*, the last configuration* $\mathfrak{C}^{\mathsf{cs}}$ *of* $\mathfrak{CT}_i^{\mathsf{cs}}$ *verifies* $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$*.*

The next lemma applies to any traces, so in particular to OCaml traces and CryptoVerif traces.

Lemma G.2 *Let $\mathcal{CT}_1, \ldots, \mathcal{CT}_n$ be traces such that none of these traces is a prefix of another of these traces. If $\mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$ are extensions of $\mathcal{CT}_n$ such that none of these traces is a prefix of another, then none of the traces $\mathcal{CT}_1, \ldots, \mathcal{CT}_{n-1}, \mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$ is a prefix of another of these traces.*

In particular, this is true when, for all $i \leq n'$, $\mathcal{CT}_i{}''$ is the concatenation of $\mathcal{CT}_n$ and $\mathcal{CT}_i{}'$ where $\mathcal{CT}_1{}', \ldots, \mathcal{CT}_{n'}{}'$ are traces that start at the last configuration of $\mathcal{CT}_n$ such that none of these traces is a prefix of another of these traces.

Proof Let us prove the first point. Consider two traces among $\mathcal{CT}_1, \ldots, \mathcal{CT}_{n-1}, \mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$. If they are both among $\mathcal{CT}_1, \ldots, \mathcal{CT}_{n-1}$, they are not prefix of one another by hypothesis. If they are both among $\mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$, they are also not prefix of one another by hypothesis. Now consider $\mathcal{CT}_i$ with $i \leq n-1$ and $\mathcal{CT}_j{}''$ with $j \leq n'$. If $\mathcal{CT}_i$ was a prefix of $\mathcal{CT}_j{}''$, then either its length is less or equal to the length of $\mathcal{CT}_n$, so $\mathcal{CT}_i$ would be a prefix of $\mathcal{CT}_n$, which is impossible by hypothesis, or its length is greater than the length of $\mathcal{CT}_n$, so $\mathcal{CT}_i$ would be an extension of $\mathcal{CT}_n$, that is, $\mathcal{CT}_n$ would be a prefix of $\mathcal{CT}_i$, which is also impossible by hypothesis. If $\mathcal{CT}_j{}''$ was a prefix of $\mathcal{CT}_i$, then a fortiori $\mathcal{CT}_n$ would be a prefix of $\mathcal{CT}_i$, which is impossible by hypothesis. Hence, none of the traces $\mathcal{CT}_1, \ldots, \mathcal{CT}_{n-1}, \mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$ is a prefix of another of these traces.

To show the second point, if $\mathcal{CT}_i{}''$ was a prefix of $\mathcal{CT}_j{}''$, then $\mathcal{CT}_i{}'$ would be a prefix of $\mathcal{CT}_j{}'$, which is a contradiction. So we can apply the first point in this case. □

Lemma G.3 *Suppose that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$. Either all traces $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$ are complete, or there exist $\mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}}{}'$ and $\mathcal{CTS}'_1, \ldots, \mathcal{CTS}'_{n'}$ such that there are strictly more reduction steps in traces $\mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}}{}'$ than in traces $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$ and $\mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}}{}' \equiv_{\mathsf{t}} \mathcal{CTS}'_1, \ldots, \mathcal{CTS}'_{n'}$.*

Proof Suppose that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ and there is a trace $\mathfrak{CT}_i^{\mathsf{cs}}$ that is not complete. We can renumber the traces so that the last trace $\mathfrak{CT}_n^{\mathsf{cs}}$ is not complete.

By Property 5(b), the last configuration $\mathfrak{C}^{\mathsf{cs}}$ of the trace $\mathfrak{CT}_n^{\mathsf{cs}}$ and all traces $\mathcal{CT} \in \mathcal{CTS}_n$ verify $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. By Property 2, $\mathcal{CT}$ is a trace beginning at $\mathcal{C}_0(Q_0, \mathit{program}_0)$. Let us denote $\mathcal{CTS}_n = \{\mathcal{CT}_1, \ldots, \mathcal{CT}_m\}$. We can then apply Lemma 6.29 to $\mathfrak{C}^{\mathsf{cs}}$.

- Either there exist n' configurations $\mathfrak{C}_1^{\mathsf{cs}}, \ldots, \mathfrak{C}_{n'}^{\mathsf{cs}}$, n' traces $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow_{p_1}^{+} \mathfrak{C}_1^{\mathsf{cs}}$, $\ldots, \mathfrak{C}^{\mathsf{cs}} \rightsquigarrow_{p_{n'}}^{+} \mathfrak{C}_{n'}^{\mathsf{cs}}$ such that none of these traces is a prefix of another, $\sum_{i \leq n} p_i = 1$, and for each trace $\mathcal{CT}_j, j \leq m$, there exist n' pairwise disjoint trace sets $\mathcal{CTS}_{j,1}, \ldots, \mathcal{CTS}_{j,n'}$ such that all traces in these sets are extensions of $\mathcal{CT}_j$, none of these traces is a prefix of another, and for each trace $\mathcal{CT} \in \mathcal{CTS}_{j,i}, \mathfrak{C}_i^{\mathsf{cs}\prime} \equiv \mathcal{CT}$ and $\Pr[\mathcal{CTS}_{j,i}] = p_i \cdot \Pr[\mathcal{CT}_j]$. Let us

denote $\mathcal{CTS}'_i \stackrel{\text{def}}{=} \bigcup_{j\leq m} \mathcal{CTS}_{j,i}$. Let us also denote $\mathfrak{C}_i^{\mathsf{cs}\prime}$ the extension of the trace $\mathfrak{C}^{\mathsf{cs}}$ until $\mathfrak{C}_i^{\mathsf{cs}}$, for $i \leq n'$. There is at least one new reduction step, so there are more reduction steps in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime}$ than in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$. Let us prove that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_{n-1}, \mathcal{CTS}'_1, \ldots, \mathcal{CTS}'_{n'}$. All traces $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime}$ start at $\mathfrak{C}_0(Q_0, \mathit{program}_0)$ and by Lemma G.2, none of these traces is a prefix of another of these traces, so Property 1 is verified. Similarly, by applying Lemma G.2 to each trace $\mathcal{CT}_j$ for $j \leq m$, Property 2 is verified. By Property 3 on the initial traces, $\forall i \leq n, \Pr[\mathfrak{C}_i^{\mathsf{cs}}] = \Pr[\mathcal{CTS}_i]$. We have that $\Pr[\mathcal{CTS}'_i] = \sum_{j\leq m} \Pr[\mathcal{CTS}_{j,i}]$ because all the sets $\mathcal{CTS}_{j,i}$ are disjoint. So,

$$\Pr[\mathcal{CTS}'_i] = \sum_{j\leq m} p_i \cdot \Pr[\mathcal{CT}_j] = p_i \cdot \Pr[\mathcal{CTS}_n]\,, \text{ so}$$
$$\Pr[\mathfrak{CT}_i^{\mathsf{cs}\prime}] = p_i \cdot \Pr[\mathfrak{CT}_n^{\mathsf{cs}}] = \Pr[\mathcal{CTS}'_i]\,,$$

Property 3 is verified. By Property 4 on the initial traces, $\sum_{i\leq n} \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] = 1$. We have that

$$\sum_{i\leq n'} \Pr[\mathfrak{CT}_i^{\mathsf{cs}\prime}] = \sum_{i\leq n'} p_i \cdot \Pr[\mathfrak{CT}_n^{\mathsf{cs}}] = \Pr[\mathfrak{CT}_n^{\mathsf{cs}}]\,, \text{ so}$$
$$\sum_{i\leq n-1} \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] + \sum_{i\leq n'} \Pr[\mathfrak{CT}_i^{\mathsf{cs}\prime}] = \sum_{i\leq n} \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] = 1\,.$$

So Property 4 is verified. We inherit Property 5 for the $n-1$ first elements. For all $i \leq n'$, for all traces $\mathcal{CT} \in \mathcal{CTS}'_i$, we have $\mathfrak{C}_i^{\mathsf{cs}\prime} \equiv \mathcal{CT}$, and $\mathfrak{C}_i^{\mathsf{cs}\prime}$ is the last configuration of $\mathfrak{CT}_i^{\mathsf{cs}\prime}$. So Property 5(b) is verified for all the new elements. Hence Property 5 is verified. Therefore, $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_{n-1}, \mathcal{CTS}'_1, \ldots, \mathcal{CTS}'_{n'}$.

- Otherwise, each trace $\mathcal{CT} \in \mathcal{CTS}_n$ is complete, $\mathfrak{C}^{\mathsf{cs}} \to^* \mathfrak{C}_1^{\mathsf{cs}}$, $\mathfrak{C}_1^{\mathsf{cs}}$ cannot reduce, and the event list $\mathcal{E}$ of $\mathfrak{C}_1^{\mathsf{cs}}$ and the event list *events* of the last configuration of $\mathcal{CT}$ satisfy $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$. Let $\mathfrak{CT}_n^{\mathsf{cs}\prime}$ be the extension of the trace $\mathfrak{CT}_n^{\mathsf{cs}}$ until $\mathfrak{C}_1^{\mathsf{cs}}$. The trace $\mathfrak{CT}_n^{\mathsf{cs}\prime}$ contains more steps than $\mathfrak{CT}_n^{\mathsf{cs}}$, so there are more reduction steps in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_n^{\mathsf{cs}\prime}$ than in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$. Let us prove that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_n^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$. By Lemma G.2, no traces in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_n^{\mathsf{cs}\prime}$ are prefixes of one another, so Property 1 is verified. Property 2 is inherited. We have that $\Pr[\mathfrak{CT}_n^{\mathsf{cs}\prime}] = \Pr[\mathfrak{CT}_n^{\mathsf{cs}}]$, so Properties 3 and 4 are verified. The trace $\mathfrak{CT}_n^{\mathsf{cs}\prime}$ is complete, every trace $\mathcal{CT} \in \mathcal{CTS}_n$ is complete, and the event list *events* of the last configuration of traces in $\mathcal{CTS}_n$ and the event list $\mathcal{E}$ of $\mathfrak{C}_1^{\mathsf{cs}}$ verify $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$, so Property 5(a) holds for the last elements. Other elements inherit Property 5, so Property 5 holds. Therefore, $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_n^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$. □

Proof (of Proposition 6.30) By Lemma 6.28, we have a trace $\mathfrak{CT}_0 = \mathfrak{C}_0(Q_0, \mathit{program}_0) \rightsquigarrow^* \mathfrak{C}^{\mathsf{cs}}$ where $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}_0$ and $\mathcal{CT}_0 = \mathcal{C}_0(Q_0, \mathit{program}_0)$. We prove easily that $\mathfrak{CT}_0 \equiv_{\mathsf{t}} \{\mathcal{CT}_0\}$. The number of steps in complete traces from configuration $\mathfrak{C}_0(Q_0, \mathit{program}_0)$ is finite. Let us consider traces such that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ with the maximum number of reduction steps. By Lemma G.3, the traces $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$ are complete. (Otherwise, we could extend them.) Since the sum of their probabilities is 1, these are all complete traces starting at $\mathfrak{C}_0(Q_0, \mathit{program}_0)$. The proposition follows. □