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@normalesup.org,bruno.blanchet@inria.fr

January 5, 2015

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 [10].

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 proposed the following approach in [9]. 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 [6, 8, 7]. 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. The specification of the protocol given as input to CryptoVerif then consists of a process representing the protocol to prove (the initial game of the proof), assumptions on the cryptographic primitives (such as "encryption is IND-CPA" and "decrypting a ciphertext with the correct key yields the initial cleartext"), and the security properties to prove. CryptoVerif then looks for a proof of the desired security properties, and when it finds one, it also provides a formula that bounds the probability of success of an attack against the desired properties as a function of the runtime of the adversary, the number of sessions of the protocol, and the probability of breaking each primitive.

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. They also specify which CryptoVerif variables should be written into files, because they are communicated from one role to another. For instance, the key generation typically writes long-term keys into files, so that they can be used by subsequent roles. 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 code, responsible in particular for sending and receiving messages from the network, which we call the *network code*. For instance, in the case of the client-server protocol, both the client and server programs consist of a mix of our generated modules, which deal with the heart of the cryptographic protocol, and manually written network code, which deals with non-cryptographic details.

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 [9]. It is the topic of this paper. To make this proof, we need a formal semantics of OCaml. We adapt the operational small-step semantics of a core part of

OCaml by Owens et al. [17, 18]. We add to this language support for simplified modules, multiple threads where only one thread can run at any given time, and communication between threads by a shared part of the store.

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

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

A1. The random number generator used by the OCaml cryptographic library is perfect.

A2. The implementation of each cryptographic primitive is a pure function and satisfies the assumptions made on it in the specification.

A3. 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).

A4. The adversary and the network code do not access files created by our implementation (e.g. private key files).

A5. The network code is a well-typed OCaml program, which does not use unsafe OCaml functions to bypass the type system.

A6. The network code does not mutate data passed to or received from generated code. This property can be guaranteed by representing such data by immutable OCaml types. However, such data includes bitstrings and the most natural type for representing bitstrings is the OCaml type `string`, which is mutable[1]. Immutable strings can be implemented in OCaml using an abstract type instead of `string`. In our semantics, strings are immutable values.

A7. Our semantics of threads is obeyed, which implies that only one thread can run at any given time and that one cannot fork in the middle of a role.

[1]From version 4.02, OCaml has a command-line option that makes `string` immutable.

Because the network code and our generated modules run inside the same programs, we use Assumptions A5 and A6 to make sure that the network code does not interfere with the generated code. In particular, Assumption A5 prevents the network code from accessing the variables contained in the environment of functions returned by our generated code. These variables may contain secret keys, which the network code could send to the adversary if it had access to them. Moreover, our generated code may return both a public key and a function that includes this public key in its environment. If the network code could modify the returned public key, it would modify the key used by the function as well, so the protocol would use an unexpected public key. Assumption A6 avoids that. Assumptions A5 and A6 are the only requirements on the network code needed to prove security so, except for these two assumptions, we consider the network code as part of the adversary. In Assumption A7, the requirement that only one thread can run at any given time can be weakened as we discuss informally at the end of Section 5.2.5: the essential requirement is that two processes that read or write the same file are not run concurrently, which can be enforced using locks. Assumption A7 also limits forking: forking is allowed when the local store is empty. In case one needs to fork in the middle of a role, one can split the role into two, which has the effect of transmitting the store via files between the two roles. It may also be possible to extend our result with an explicit fork instruction in the OCaml language.

Assumptions A1, A5, and A7 are built into our semantics of OCaml, defined in Section 5. Assumption A2 is formalized below by Assumption 8.4, with additional technical details formalized in Assumptions 8.1 and 8.2. Assumptions A3, A4, and A6 are formalized by Assumptions 6.1, 7.2, and 8.3, respectively.

In this work, we do not consider side-channel attacks, such as timing and power consumption attacks, nor physical attacks. Like other mechanized tools for cryptographic proofs, CryptoVerif does not deal with these attacks.

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. [16, 19]. Other approaches analyze implementations by extracting a specification verified by a symbolic protocol verifier, e.g. [5, 1], or analyze them by other tools such as the model-checker ASPIER [11], the general-purpose C verifier VCC [13], symbolic execution [12], or typing [4, 20].

In contrast, the following approaches provide computational security guarantees, by analyzing implementations. The tool FS2CV [15] translates a subset of F# to the input language of CryptoVerif, which can then prove the protocol secure. The tool F7 [4], which uses a dependent type system to prove security properties on protocols implemented in F#, has been adapted to the computational model in [14]; 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.

Recently, Almeida et al [3] introduced a new approach for generating implementations with a computational proof. They extend the cryptographic prover EasyCrypt to support C-like programs, then they generate proved assembly code using an extended version of the CompCert certified C compiler. They mainly target cryptographic primitives (for instance, OAEP), and using EasyCrypt requires the user to give the games of the cryptographic proof, while in our approach CryptoVerif generates them.

To the best of our knowledge, our approach and that of [3] are the only ones for generating implementations with a computational proof. [2], [3], and our work are the only ones to provide an explicit bound on the probability of success of an attack against the verified protocol implementation.

2 Intuitive Overview

In order to prove the correctness of a compiler, we first need a formal semantics of the source and target languages, and a formal definition of the compiler. Handling all this formalism is probably the main challenge of this paper; it explains its length.

After introducing some notations (Section 3), our first task is to formally define the common input language of CryptoVerif and of our compiler (Section 4). We define the semantics of this language as a probabilistic transition system on semantic configurations. CryptoVerif uses events to define security properties. For instance, a security property may be "if event Baccepts(m') has been executed, then event Asends(m') has also been executed". For each security property, we define a *distinguisher* D that is true when the executed sequence of events breaks the security property. We denote by $\Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}) :^{(\mathsf{CV})} D]$ the probability that the security property associated to D is broken starting from the initial configuration $\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}})$, which runs the protocol Q_0 in parallel with the adversary Q_{adv}. In other words, $\Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}) :^{(\mathsf{CV})} D]$ is the probability that the adversary Q_{adv} breaks the desired security property of the protocol Q_0. When it proves the security property, CryptoVerif provides a formula that bounds this probability for any adversary Q_{adv}, as a function of the runtime of the adversary, the execution time of the cryptographic primitives and of various CryptoVerif constructs, the number of calls to each oracle, the probability of collisions between random numbers, and the probability of breaking each primitive.

Section 5 defines the OCaml language. We rely on the operational small-step semantics of a core part of OCaml by Owens et al. [17, 18], which we adapt to our setting. We add a primitive for random choices, which makes the semantics probabilistic. We also add support for simplified modules, multiple threads, and communication between threads by a shared part of the store. We adopt a simplified model of parallelism: only one thread runs at a time, and the adversary is in charge of scheduling. This model of parallelism is close to what happens in CryptoVerif; we explain informally why it is sufficient for our purpose in Section 5. Like the semantics of the CryptoVerif input language, the semantics of OCaml is defined as a probabilistic transition system on semantic configurations.

In order to prove our compiler, we instrument OCaml code in three ways (Section 6). We add events to the language, so that we can specify security properties in OCaml as we do in CryptoVerif. We introduce tagged functions and closures, which have the same semantics as ordinary functions and closures, but contain additional tags used in our code generation to indicate from which role or oracle the function comes. Each CryptoVerif role is translated by our compiler into an OCaml module; we add to the OCaml semantics a multiset of callable modules, which indicates which modules can be called to guarantee that only allowed roles are executed, as required by Assumption A3. We 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. More precisely, we show a weak bisimulation between the non-instrumented and the instrumented semantics.

Section 7 defines the translation from CryptoVerif to OCaml. In this translation, each role generates a module, and the oracles are represented by closures. Basically, the translation implements in OCaml the semantics of CryptoVerif given in Section 4. The translation is the same as the one given [9], except that the generated OCaml code is instrumented. The generated modules are combined with manually written network code (which is in particular responsible for inputting and outputting messages on the network) to produce the complete programs that implement the protocol. These programs are run in interaction with an adversary, which we also represent by an OCaml program. This is possible because OCaml with random choices is probabilistic Turing complete. The code generated from the CryptoVerif process Q_0, the network code, and the adversary are all grouped into the OCaml program $\mathit{program}_0$, and we denote by $\mathbb{C}_0(Q_0, \mathit{program}_0)$ the initial (instrumented) OCaml semantic configuration that runs $\mathit{program}_0$. The probability $\Pr[\mathbb{C}_0(Q_0, \mathit{program}_0) :^{(\mathsf{ML})} D]$ denotes the probability that the OCaml adversary defined in $\mathit{program}_0$ breaks the security property associated to the distinguisher D of the protocol Q_0.

Section 8 proves the correctness of this compiler. Informally, when CryptoVerif shows that Q_0 satisfies a certain security property, it shows that for any CryptoVerif adversary Q_{adv}, the probability $\Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}) :^{(\mathsf{CV})} D]$ is bounded by a certain bound. Our goal is to show that the same probability bound also applies to the generated implementation, that is, the probability $\Pr[\mathbb{C}_0(Q_0, \mathit{program}_0) :^{(\mathsf{ML})} D]$ that $\mathit{program}_0$ breaks the security property is bounded by the same bound for any $\mathit{program}_0$.

The presence of an arbitrary adversary complicates the proof. As illustrated in Figure 1 and detailed in Section 8.3, to solve this problem, we build from the OCaml adversary defined in $\mathit{program}_0$ a CryptoVerif adversary $Q_{\mathsf{adv}}(Q_0, \mathit{program}_0)$ that simulates $\mathit{program}_0$. Basically, we run the OCaml program $\mathit{program}_0$ inside a CryptoVerif function $\mathrm{simulate}_{\mathsf{ML}}$. Since these functions can

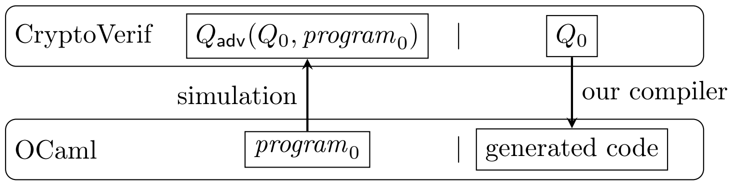

Figure 1: Overview of our proof

represent deterministic Turing machines, when $program_0$ needs to generate a random number, the function $\text{simulate}_{\mathsf{ML}}$ returns and this generation is performed by CryptoVerif. Similarly, when $program_0$ would call the generated implementation of an oracle, the function $\text{simulate}_{\mathsf{ML}}$ returns and $Q_{\mathsf{adv}}(Q_0, program_0)$ calls the corresponding CryptoVerif oracle in Q_0.

The initial CryptoVerif configuration is then $\mathcal{C}_0(Q_0, program_0) = \mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}(Q_0, program_0))$. We prove that, for all protocols Q_0, OCaml adversaries defined in $program_0$, and distinguishers D, we have

$$\Pr[\mathcal{C}_0(Q_0, program_0) :^{(\mathsf{CV})} D] = \Pr[\mathbb{C}_0(Q_0, program_0) :^{(\mathsf{ML})} D]\,. \qquad (1)$$

From this property, it is easy to see that, if CryptoVerif bounds the probability $\Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}) :^{(\mathsf{CV})} D]$ for any adversary Q_{adv} for Q_0, then the same bound also holds for the probability $\Pr[\mathbb{C}_0(Q_0, program_0) :^{(\mathsf{ML})} D]$ corresponding to the generated implementation. Indeed, $\Pr[\mathbb{C}_0(Q_0, program_0) :^{(\mathsf{ML})} D] = \Pr[\mathcal{C}_0(Q_0, program_0) :^{(\mathsf{CV})} D] = \Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}(Q_0, program_0)) :^{(\mathsf{CV})} D]$ and $Q_{\mathsf{adv}}(Q_0, program_0)$ is an adversary for Q_0.

To prove (1), we basically need to show that $Q_0 \mid Q_{\mathsf{adv}}(Q_0, program_0)$ and $program_0$ using the code generated from Q_0 behave similarly. This proof proceeds in several steps:

- First, we state our assumptions on the implementation of the cryptographic primitives, and show that the primitives behave correctly independently of the rest of the program (Section 8.1).
- Second, we prove that the OCaml translation of a CryptoVerif oracle behaves like the oracle (Section 8.2).
- Finally, in Section 8.4, we prove that the CryptoVerif adversary interacting with Q_0 behaves like the OCaml adversary interacting with the generated implementation. This proof is done by establishing a precise relation between the CryptoVerif and OCaml semantic configurations.

Therefore, we obtain the desired proof of (1) (Theorem 8.38). Because of the length of this proof, details are postponed to the appendix. An index of notations can be found in Appendix H.

3 Notations

Let us introduce some basic notations. When f is a function, we denote by $\mathit{Dom}(f)$ 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 $x \in \mathit{Dom}(f_1)$ and $f'(x) = f_2(x)$ if $x \in \mathit{Dom}(f_2)$. When f_1 and f_2 are functions, we write $f_1 \subseteq f_2$ (or $f_2 \supseteq f_1$) when $\mathit{Dom}(f_1) \subseteq \mathit{Dom}(f_2)$ and, for all $x \in \mathit{Dom}(f_1)$, we have $f_2(x) = f_1(x)$. We denote by $\emptyset$ any function whose domain is the empty set $\emptyset$.

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 $[x_1; \dots; x_k]$ be the list $x_1 :: \dots :: x_k :: [\,]$. 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 order as in l. This construct is defined by induction on lists:

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

The concatenation of lists $l_1 \mathbin{@} l_2$ 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 $\mathit{nth}(l, n)$ be the nth element of list l.

We define the function $\mathit{almostunif}(A, b)$ as the probability that the element $b \in A$ is chosen among elements of the set A, according to an almost uniform distribution: we require that, for every set A, $\sum_{b \in A} \mathit{almostunif}(A, b) = 1$, $\mathit{almostunif}(A, b) > 0$ for all $b \in A$, and $\sum_{b \in A} \left| \mathit{almostunif}(A, b) - \frac{1}{|A|} \right| \leq \epsilon$ 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.

Fonts. We use a sans-serif font for CryptoVerif keywords (e.g., foreach) and role names (e.g., keygen), a roman font for CryptoVerif function, constant, event, and oracle symbols, and an italic font for CryptoVerif types, variables, and file names. We use a bold font for OCaml keywords (e.g., **match**) and constructors (e.g., **Callable**), and an italic font for OCaml types and other identifiers. We use uppercase italic letters (e.g., E, P) and a calligraphic font (e.g. $\mathcal{C}$) for CryptoVerif semantic elements, while we use lowercase italic words (e.g., *env*) and a blackboard font (e.g., $\mathbb{C}$) for OCaml semantic elements. We use an italic font for most other mathematical symbols, and a sans-serif font for constant elements.

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

Figure 2: Syntax of the CryptoVerif language

4 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 [6] uses channels to pass data between the adversary and the protocol, and the oracle front-end [8] 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 [6] for the channel front-end to the oracle front-end.

4.1 Syntax and Informal Semantics

Let us first introduce the syntax of the CryptoVerif language in Figure 2. The language is typed, and types T are subsets of $\mathit{bitstring}_\perp \stackrel{\text{def}}{=} \mathit{bitstring} \cup \{\perp\}$ where $\mathit{bitstring}$ 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 $\mathit{bool} \stackrel{\text{def}}{=} \{\text{true}, \text{false}\}$, where true is the bitstring 1 and false 0, and the types $\mathit{bitstring}$ and $\mathit{bitstring}_\bot$ are predefined.

Variables $x[i_1, \dots, i_m]$ are arrays of bitstrings of a given type T. As formalized by Property 4.3 below, each variable $x[i_1, \dots, i_m]$ has a single definition and the indices $i_1, \dots, i_m$ are the indices of the replications $\mathsf{foreach}\ i_m \leq N_m\ \mathsf{do}\ \dots \mathsf{foreach}\ i_1 \leq N_1\ \mathsf{do}\ Q$ present above the definition of x: each replication $\mathsf{foreach}\ i \leq N\ \mathsf{do}\ Q$ creates N copies of the process Q, in which i is set to 1, …, N respectively. Then the indices $i_1, \dots, i_m$ have different values in different executions of the definition of x, so that each cell of the array x is assigned at most once. Therefore, arrays allow us to remember all values of the variables during the execution of the process. We call the indices $i_1, \dots, i_m$ *replication indices*, and we abbreviate $i_1, \dots, i_m$ by $\widetilde{i}$. The indices $i_1, \dots, i_m$ are ordered from the inner-most to the outer-most replication. Since the indices of x are entirely determined by the replications above the definition of x, we often omit them to lighten notations. Each function f comes with its type $T_1 \times \dots \times T_m \to T$; all CryptoVerif functions are deterministic and efficiently computable. Some functions are predefined, and some are infix, like the equality test $=$ and boolean operations. The cryptographic primitives used in the protocol are represented by CryptoVerif functions. Terms M represent computations over bitstrings: they can be variable accesses $x[i_1, \dots, i_m]$ or function applications $f(M_1, \dots, M_m)$.

The oracle definitions Q represent the oracles that will become available to the adversary at this point. The nil construct 0 provides no oracle. The parallel composition $Q \mid Q'$ provides oracles in Q and Q'. The replication $\mathsf{foreach}\ i \leq N\ \mathsf{do}\ Q$ provides N copies of Q, indexed by $i \in \{1, \dots, N\}$. 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 $O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k) := P$ makes available the oracle $O[\widetilde{i}]$; when $O[\widetilde{i}]$ is called by the adversary with arguments $a_1, \dots, a_k$, it executes the oracle body P with $x_j[\widetilde{i}]$ set to a_j.

The oracle bodies P represent the behavior of the oracle. A return statement $\mathsf{return}(M_1, \dots, M_k); Q$ returns the result of $M_1, \dots, M_k$ to the caller, and makes available oracles in Q. An end statement end returns to the caller 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 $\mathsf{insert}\ \mathit{Tbl}(M_1, \dots, M_k); P$ inserts the result of $M_1, \dots,$ M_k into the table Tbl. Tables are lists of tuples, used for example to store tables of keys. Each table Tbl has a type $T_1 \times \dots \times T_k$, which means that Tbl contains k-tuples $a_1, \dots, a_k$ such that a_j is of type T_j for all $j \leq k$. A get statement $\mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{i}], \dots, x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$ searches for an element $a_1, \dots, a_k$ in the table Tbl such that the term M evaluates to true when $x_1[\widetilde{i}] = a_1, \dots, x_k[\widetilde{i}] = a_k$. If there is no such element, we continue by executing P', and otherwise we choose almost-uniformly one of the elements that correspond, store it in the variables $x_1[\widetilde{i}], \dots, x_k[\widetilde{i}]$, then execute P. An event statement $\mathsf{event}\ ev(M_1, \dots, M_k); P$ is used to log events. Events serve for specifying security properties of protocols, but do not change the execution of the process.

An oracle call $\mathsf{let}\ (x_1[\widetilde{i}] : T_1, \dots, x_{k'}[\widetilde{i}] : T_{k'}) = O[M_1, \dots, M_l](M'_1, \dots, M'_k)$ $\mathsf{in}\ P\ \mathsf{else}\ P'$ calls oracle $O[M_1, \dots, M_l]$ with arguments $M'_1, \dots, M'_k$, stores its returned values in the variables $x_1[\widetilde{i}], \dots, x_{k'}[\widetilde{i}]$, 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 $\mathsf{let}\ x[\widetilde{i}] : T = \mathsf{loop}\ O[M_1, \dots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P'$ 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 b indicating whether the loop should continue or not. For clarity, we use continue as a synonym for true and stop for false in this context. $O[M_1, \dots, M_l](M')$ is first called. If it returns $(a_1, \text{continue})$, $O[M_1 + 1, M_2, \dots, M_l](a_1)$ is called. If it returns $(a_2, \text{continue})$, $O[M_1 + 2, M_2, \dots, M_l](a_2)$ is called, and so on, until $O[M_1 + k, M_2, \dots, M_l](a_k)$ returns (a_{k+1}, stop). 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. Some protocols use loops or recursion, for instance for certificate checking; such protocols could in principle be encoded in our language by using replicated processes and transmitting internal state from one iteration to the next using a table or an encrypted message. However, this idea leads to contrived encodings and a native loop construct would be more convenient. Including loops in protocols would not cause major problems for generating implementations, but would considerably complicate the prover CryptoVerif itself, since it would have to discover loop invariants. That is why we leave the inclusion of loops in protocols for future work.

Example 4.1 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:

$$\begin{aligned}
&\text{Okeygen}() := \\
&\qquad rk \overset{R}{\leftarrow} \mathit{keyseed}; pk \leftarrow \text{pkgen}(rk); sk \leftarrow \text{skgen}(rk); \\
&\qquad \mathsf{return}(pk); (\mathsf{foreach}\ i_1 \leq N_1\ \mathsf{do}\ P_A \mid \mathsf{foreach}\ i_2 \leq N_2\ \mathsf{do}\ P_B) \\
&P_A \overset{\text{def}}{=} \text{OA}() := \\
&\qquad m \overset{R}{\leftarrow} \mathit{nonce}; s \overset{R}{\leftarrow} \mathit{seed}; \mathsf{event}\ \text{Asends}(m); \mathsf{return}(m, \text{sign}(m, sk, s))
\end{aligned}$$

```
$P_B \overset{\text{def}}{=} \mathrm{OB}(m' : \mathit{nonce}, s' : \mathit{signature}) :=$
    if $\mathrm{check}(m', \mathit{pk}, s')$ then (event $\mathrm{Baccepts}(m')$; return()) else end
```

The only callable oracle at the beginning is the oracle Okeygen, which generates the signature key pair $(\mathit{pk}, \mathit{sk})$ 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 Okeygen 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. Then, it executes the event $\mathrm{Asends}(m)$. This event just records that A sends the nonce m, without changing the execution of the process; we use it below to specify a security property. Finally, OA 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 executes the event $\mathrm{Baccepts}(m')$ and returns normally. Otherwise, the oracle terminates with end.

The goal of this protocol is to guarantee that, with high probability, if B accepts a nonce m', then A sent this nonce m', that is, if event $\mathrm{Baccepts}(m')$ has been executed, then event $\mathrm{Asends}(m')$ has also been executed. This property is proved by CryptoVerif when signatures are unforgeable under chosen-message attacks (UF-CMA), as detailed in Example 4.9 below.

Example 4.2 The previous toy example is not very realistic, in particular because B accepts messages only from A. In a more realistic setting, B could be a server that would process messages coming from several different clients. B would then use a table of keys to relate the identity of each client to its public key. We would then use the following process:

```
Okeygen() :=
    $\mathit{rk} \overset{R}{\leftarrow} \mathit{keyseed}; \mathit{pk} \leftarrow \mathrm{pkgen}(\mathit{rk}); \mathit{sk} \leftarrow \mathrm{skgen}(\mathit{rk});$
    insert $\mathit{KeyTbl}(A, \mathit{pk})$; return($\mathit{pk}$);
    (foreach $i_1 \leq N_1$ do $P_A$ | foreach $i_2 \leq N_2$ do $P_B$ | foreach $i_3 \leq N_3$ do $P_R$)
$P_A \overset{\text{def}}{=} \mathrm{OA}() :=$
    $m \overset{R}{\leftarrow} \mathit{nonce}; s \overset{R}{\leftarrow} \mathit{seed}$; event $\mathrm{Asends}(m)$; return($A, m, \mathrm{sign}(m, \mathit{sk}, s)$)
$P_B \overset{\text{def}}{=} \mathrm{OB}(h' : \mathit{host}, m' : \mathit{nonce}, s' : \mathit{signature}) :=$
    get $\mathit{KeyTbl}(h, \mathit{pkh})$ suchthat $h' = h$ in
    if $\mathrm{check}(m', \mathit{pkh}, s')$ then (event $\mathrm{Baccepts}(h', m')$; return()) else end
$P_R \overset{\text{def}}{=} \mathrm{OR}(h : \mathit{host}, \mathit{pkh} : \mathit{pkey}) :=$
    if $h \neq A$ then insert $\mathit{KeyTbl}(h, \mathit{pkh})$
```

When A's key pair is created, the pair (A, pk) is added to the key table $KeyTbl$, to record that pk is the public key of A. The additional oracle OR allows the adversary to record its own public keys in the key table for any host name other than the honest host A. Hence, the model allows B to interact both with the honest participant A and with any other dishonest participants. The message sent by A additionally contains the host name A, and B uses the host name h' to get the corresponding key pkh, which he uses to verify the signature. The event Baccepts also contains h' as additional argument: it means that B accepts the message m' coming from h'. The desired security property is that, with high probability, if B accepts the message m' coming from A, then A sent the message m', that is, if event Baccepts(A, m') has been executed, then event Asends(m') has also been executed.

Tables of keys appear in many realistic protocols. For instance, the SSH client stores a table that contains the public keys and the names of the servers it connected to.

4.2 Formal Semantics

We present the semantics of the language in Figures 3 and 4. The semantics is defined as a reduction relation on semantic configurations, which are tuples of the form $\mathcal{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$.

- 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 environment keeps every binding ever bound, thanks to replication indices, so it is ever increasing.
- 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 $\mathcal{S}$ is the call stack, which consists of triplets containing the variables with which the result should be bound and two oracle bodies, the first will be executed if the oracle returns a result with a `return` statement, and the second will be executed if the oracle terminates with an `end` statement.
- The list $\mathcal{E}$ is the list of events $ev(a_1, \dots, a_k)$ executed so far, by the construct `event` $ev(M_1, \dots, M_k)$.

During execution, terms may be reduced into constant bitstrings, so we add constant bitstrings a to the grammar of terms M. The notation $E \cdot M \Downarrow a$ 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 3.

Terms:

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

$$\frac{x[a_1, \dots, a_m] \in \mathit{Dom}(E)}{E \cdot x[a_1, \dots, a_m] \Downarrow E(x[a_1, \dots, a_m])} \tag{Var}$$

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

Oracle bodies (1):

$$\frac{T \text{ fixed-length type} \qquad a \in T}{E, x[\widetilde{a'}] \overset{R}{\leftarrow} T; P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \to_{\frac{1}{|T|}} E[x[\widetilde{a'}] \mapsto a], P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}} \tag{New}$$

$$\frac{E \cdot M \Downarrow a}{E, x[\widetilde{a'}] \leftarrow M; P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \to_1 E[x[\widetilde{a'}] \mapsto a], P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}} \tag{Let}$$

$$\frac{E \cdot M \Downarrow \text{true}}{E, \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else} P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \to_1 E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}} \tag{If1}$$

$$\frac{E \cdot M \Downarrow \text{false}}{E, \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else} P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \to_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}} \tag{If2}$$

$$\frac{\forall j \leq k, E \cdot M_j \Downarrow a_j}{\begin{array}{l}E, \mathsf{insert}\ \mathit{Tbl}(M_1, \dots, M_k); P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \to_1 \\ \quad E, P, \mathcal{T}[\mathit{Tbl} \mapsto (a_1, \dots, a_k) :: \mathcal{T}(\mathit{Tbl})], \mathcal{Q}, \mathcal{S}, \mathcal{E}\end{array}} \tag{Insert}$$

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

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

Figure 3: Semantics (1)

Oracle bodies (2):

$$\frac{\begin{array}{c}\forall i \leq l, E \cdot M_i \Downarrow a'_i \qquad \widetilde{a'} = a'_1, \dots, a'_l \qquad \forall j \leq k, E \cdot N_j \Downarrow b_j \\ \exists x'_1, \dots, x'_k, P'' \text{ such that} \\ Q_0 = (O[\widetilde{a'}](x'_1[\widetilde{a'}] : T'_1, \dots, x'_k[\widetilde{a'}] : T'_k) := P'') \in \mathcal{Q} \\ E' = E[x'_1[\widetilde{a'}] \mapsto b_1, \dots, x'_k[\widetilde{a'}] \mapsto b_k]\end{array}}{\begin{array}{l}E, \mathsf{let}\ (x_1[\widetilde{a}] : T_1, \dots, x_{k'}[\widetilde{a}] : T_{k'}) = O[M_1, \dots, M_l](N_1, \dots, N_k) \\ \quad \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \quad \rightarrow_1 \\ \qquad E', P'', \mathcal{T}, \mathcal{Q} \setminus \{Q_0\}, ((x_1[\widetilde{a}], \dots, x_{k'}[\widetilde{a}]), P, P') :: \mathcal{S}, \mathcal{E}\end{array}} \quad \text{(Call)}$$

$$\frac{\forall j \leq k, E \cdot N_j \Downarrow b_j \qquad \mathcal{Q}' = \mathit{oracledefset}(Q'')}{\begin{array}{l}E, \mathsf{return}(N_1, \dots, N_k); Q'', \mathcal{Q}, ((x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]), P, P') :: \mathcal{S}, \mathcal{E} \\ \quad \rightarrow_1 E[x_1[\widetilde{a}] \mapsto b_1, \dots, x_k[\widetilde{a}] \mapsto b_k], P, \mathcal{T}, \mathcal{Q} \cup \mathcal{Q}', \mathcal{S}, \mathcal{E}\end{array}} \quad \text{(Return)}$$

$$E, \mathsf{end}, \mathcal{T}, \mathcal{Q}, ((x_1[\widetilde{a}], \dots, x_{k'}[\widetilde{a}]), P, P') :: \mathcal{S}, \mathcal{E} \rightarrow_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \quad \text{(End)}$$

$$\frac{\forall j \leq l, E \cdot M_j \Downarrow a_j}{E, \mathsf{event}\ ev(M_1, \dots, M_l); P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \rightarrow_1 E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, ev(a_1, \dots, a_l) :: \mathcal{E}} \quad \text{(Event)}$$

$$\frac{\begin{array}{c}\forall i \leq l, E \cdot M_i \Downarrow a'_i \qquad E \cdot M' \Downarrow c \\ \text{the last replication above the definition of } O \text{ is } \mathsf{foreach}\ i_1 \leq N_1 \qquad a'_1 \leq N_1\end{array}}{\begin{array}{l}E, \mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[M_1, \dots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \\ \quad \rightarrow_1 E, \\ \qquad (\mathsf{let}\ (r'_{a'_1,r}[\widetilde{a}] : T, b_{a'_1,r}[\widetilde{a}] : \mathit{bool}) = O[a'_1, \dots, a'_l](c)\ \mathsf{in} \\ \qquad\quad \mathsf{if}\ b_{a'_1,r}[\widetilde{a}]\ \mathsf{then} \\ \qquad\qquad (\mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[a'_1 + 1, a'_2, \dots, a'_l](r'_{a'_1,r}[\widetilde{a}] : T) \\ \qquad\qquad\ \mathsf{in}\ P\ \mathsf{else}\ P') \\ \qquad\quad \mathsf{else}\ r[\widetilde{a}] \leftarrow r'_{a'_1,r}[\widetilde{a}]; P \\ \qquad \mathsf{else}\ P'), \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}\end{array}} \quad \text{(Loop1)}$$

$$\frac{\begin{array}{c}\forall i \leq l, E \cdot M_i \Downarrow a'_i \qquad E \cdot M' \Downarrow c \\ \text{the last replication above the definition of } O \text{ is } \mathsf{foreach}\ i_1 \leq N_1 \qquad a'_1 > N_1\end{array}}{\begin{array}{l}E, \mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[M_1, \dots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \\ \quad \rightarrow_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}\end{array}} \quad \text{(Loop2)}$$

Figure 4: Semantics (2)

The semantics is defined by probabilistic reduction rules between configurations: $\mathcal{C} \rightarrow_p \mathcal{C}'$ means that $\mathcal{C}$ reduces into $\mathcal{C}'$ with probability p. This relation is defined in the part "Oracle bodies (1)" of Figure 3 and in Figure 4.

The rule (New) evaluates $x[\widetilde{a'}] \xleftarrow{R} T$ 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 $E(x_1[\widetilde{a'}])$, ..., $E(x_k[\widetilde{a'}])$, and takes the in branch. The j-th element of the list l is chosen with probability $\mathit{almostunif}(\{1, \dots, |l|\}, j)$. In case the same element $a_1^0, \dots, a_k^0$ 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 $a_1^0, \dots, a_k^0$ 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 $\mathsf{let}\ (x_1[\widetilde{a}] : T_1, \dots, x_{k'}[\widetilde{a}] : T_{k'}) = O[M_1, \dots, M_l](N_1, \dots, N_k)\ \mathsf{in}\ P\ \mathsf{else}\ P'$. It evaluates the indices $M_1, \dots, M_l$ of the oracle to call into $\widetilde{a'}$ and its arguments $N_1, \dots, N_k$ into $b_1, \dots, b_k$; after evaluation, we want to call the oracle $O[\widetilde{a'}](b_1, \dots, b_k)$. 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 $b_1, \dots, b_k$ in the arguments of Q_0, and running its body P''. The element $(x_1[\widetilde{a}], \dots, x_{k'}[\widetilde{a}]), P, P')$ is pushed on the stack $\mathcal{S}$: $x_1[\widetilde{a}], \dots, x_{k'}[\widetilde{a}]$ 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 $((x_1[\widetilde{a}], \dots, x_{k'}[\widetilde{a}]), P, P')$ from the stack, stores the return value in $x_1[\widetilde{a}], \dots, x_{k'}[\widetilde{a}]$, 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 set $\mathit{oracledefset}(Q)$ contains all oracle definitions provided by the oracle definition Q, with replication indices instantiated to all their possible values, defined as follows:

$$\mathit{oracledefset}(0) \stackrel{\text{def}}{=} \emptyset \tag{Nil}$$

$$\mathit{oracledefset}(Q_1 \mid Q_2) \stackrel{\text{def}}{=} \mathit{oracledefset}(Q_1) \cup \mathit{oracledefset}(Q_2) \tag{Par}$$

$$\mathit{oracledefset}(\mathsf{foreach}\ i \le n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} \bigcup_{a=1}^{n} \mathit{oracledefset}(Q\{a/i\}) \tag{Repl}$$

$$\begin{aligned}&\mathit{oracledefset}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\text{def}}{=} \\ &\qquad \{(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k) := P\}\end{aligned} \tag{Oracle}$$

The notation $Q\{a/i\}$ means that we replace all occurrences of i by a in Q.

The rule (End) also pops an element $((x_1[\widetilde{a}], \dots, x_{k'}[\widetilde{a}]), P, P')$ 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 value $b_{a'_1,r}[\widetilde{a}]$ returned by O is stop, that 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 continue, that 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_0 is $\mathcal{C}_{\mathsf{i}}(Q_0) \overset{\text{def}}{=} \emptyset, \mathsf{let}\ x[\,] : \mathit{bitstring} = \mathrm{O}_{\mathsf{start}}()\ \mathsf{in}\ \mathsf{return}(x)\ \mathsf{else}\ \mathsf{end}, \mathcal{T}_0, \mathit{oracledefset}(Q_0), \emptyset, [\,]$, where $\mathcal{T}_0(\mathit{Tbl}) = [\,]$ for all tables Tbl. This configuration starts by calling oracle $\mathrm{O}_{\mathsf{start}}$. The oracle definition Q_0 typically contains a protocol in parallel with an adversary.

CryptoVerif verifies the following requirements on Q_0:

Property 4.3 *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 4.3 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.)

Property 4.4 *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 [6] for a similar type system.)*

Property 4.4 requires the adversary to be well-typed. This requirement does not restrict its computing power, because well-typed processes are Turing-complete, since primitives can implement any deterministic Turing machine. The type system also does not restrict the class of protocols that we consider, since the protocol may contain type-cast functions $f : T \to T'$ to bypass the type system. The type system just makes explicit which set of bitstrings may appear at each point of the protocol.

Property 4.5 *We define types of oracles as follows. The type of a* $\mathsf{return}(M_1, \dots, M_k); Q$ *statement consists of the types of $M_1, \dots, M_k$ 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 4.5 guarantees that the various definitions of an oracle are consistent, and can in fact be compiled into a single function in OCaml. The oracles at the beginning of Q are the oracles found in Q without recursively looking into oracle definitions.

Property 4.6 *Oracles with the same name can be defined only in different branches of an* if *or* get *construct. In an oracle definition $O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k) := P$, the indices $\widetilde{i}$ are always the indices of replications above that oracle definition.*

Property 4.6 guarantees that there exists a single callable definition for each oracle. This property is formalized by the following lemma, proved in Appendix A.

Lemma 4.7 (Oracle name and indices unicity) *If the configuration $\mathcal{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$ is reachable from the initial configuration $\mathcal{C}_\mathsf{i}(Q_0)$ by reductions $\rightarrow_p$, 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.

As a consequence, if a configuration $\mathcal{C}$ is non-blocking (that is, $\mathcal{C} \rightarrow_p \mathcal{C}'$ for some p and $\mathcal{C}'$), then the sum of the probabilities of all the possible reductions from $\mathcal{C}$ is 1:

$$\sum_{\{\mathcal{C}' \mid \mathcal{C} \rightarrow_{p(\mathcal{C}')} \mathcal{C}'\}} p(\mathcal{C}') = 1\,.$$

Definition 4.8 (Traces) *Let us denote traces with the symbol $\mathcal{CT}$. A trace is a sequence of reductions $\mathcal{CT} = \mathcal{C}_0 \rightarrow_{p_1} \cdots \rightarrow_{p_n} \mathcal{C}_n$ where $\mathcal{C}_0, \dots, \mathcal{C}_n$ are semantic configurations such that $\mathcal{C}_i \rightarrow_{p_{i+1}} \mathcal{C}_{i+1}$ for $i = 0, \dots, n-1$.*

A complete *trace is a trace whose last configuration is blocking.*

The probability of the trace $\mathcal{CT}$ is $\Pr[\mathcal{CT}] = p_1 \times \cdots \times p_n$. When no trace in a set of traces $\mathcal{CTS}$ is a prefix of another, the probability of $\mathcal{CTS}$ is the sum of the probabilities of its elements.

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

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

The notation $\mathcal{C} \rightarrow^* \mathcal{C}'$ means $\mathcal{C} \rightarrow_1^* \mathcal{C}'$. We denote the number of steps in the trace $\mathcal{CT}$ as $|\mathcal{CT}| = n$.

Intuitively, when no trace in $\mathcal{CTS}$ is a prefix of another, the traces in $\mathcal{CTS}$ correspond to disjoint cases, so the probability of $\mathcal{CTS}$ is the sum of probabilities of the traces in $\mathcal{CTS}$. (When $\mathcal{CT}$ is a prefix of $\mathcal{CT}'$, the trace $\mathcal{CT}'$ is a particular case of $\mathcal{CT}$.) In the notation $\mathcal{C} \rightarrow_p^* \mathcal{C}'$, we consider the set $\mathcal{CTS}$ of all traces beginning at $\mathcal{C}$ and stopping at their first occurrence of $\mathcal{C}'$. No trace in this set is a prefix of another: if a trace $\mathcal{CT}_1$ was a prefix of $\mathcal{CT}_2$ with both traces in $\mathcal{CTS}$, then $\mathcal{CT}_2$ would contain $\mathcal{C}'$ in the middle, at the end of the prefix $\mathcal{CT}_1$, so it would not stop at the first occurrence of $\mathcal{C}'$, which contradicts the definition of $\mathcal{CTS}$. Therefore, the probability $p = \Pr[\mathcal{CTS}]$ is well defined.

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 4.8 are also used for OCaml where there could be several configurations reducing to the same configuration, so they support this situation.

Finally, 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 $\Pr[\mathcal{C} :^{(\mathsf{CV})} D]$ the probability of the set of complete CryptoVerif traces starting at $\mathcal{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = \text{true}$. We define D such that $D(\mathcal{E}) = \text{true}$ if and only if $\mathcal{E}$ does not satisfy the desired security property. We 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 CryptoVerif bounds the probability $\Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}) :^{(\mathsf{CV})} D]$, that is, the probability that the adversary Q_{adv} breaks the desired security property in Q_0, for any adversary Q_{adv} for Q_0.

Example 4.9 To show that the protocol Q_0 of Example 4.1 satisfies the correspondence c "for all m', if Baccepts(m') has been executed, then Asends(m') 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 Baccepts(m') but not Asends(m') for some m'. Then CryptoVerif shows that for all Q_{adv},

$$\Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}) :^{(\mathsf{CV})} D_c] \leq \mathsf{Succ}_{\text{sign}}^{\mathsf{uf-cma}}(t + (N_2 - 1)t_{\text{check}}, N_1)$$

where t is the execution time of the adversary Q_{adv}, t_{check} is the maximum execution time of a call to check, N_1 is the maximum number of calls to oracle OA, N_2 is the maximum number of calls to oracle OB, and $\mathsf{Succ}_{\text{sign}}^{\mathsf{uf-cma}}(t', n')$ is the probability of forging a signature in time t' with at most n' calls to the signature oracle. When the signatures are UF-CMA, the probability $\mathsf{Succ}_{\text{sign}}^{\mathsf{uf-cma}}(t', n')$ is small for reasonable values of t' and n', then so is $\Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}) :^{(\mathsf{CV})} D_c]$, so the desired security property holds.

We can also define secrecy using events and distinguishers [7].

4.3 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 a role role is specified by adding the annotation $\mathsf{role}\{$ just before the oracle definition Q that starts role, where role is the name of a role, such as alice or bob; the end of the role role is specified by a closing brace $\}$ between a $\mathsf{return}(\dots)$ and its following oracle definition Q'. We denote by $Q(\mathsf{role})$ the part of the process corresponding to the role role. A role can contain several oracles, and can thus represent a protocol participant that receives or sends several messages, for instance as follows:

$$\mathsf{role}\ \{O_1(x_1 : T_1) := \dots \mathsf{return}(M_2); O_2(x_3 : T_3) := \dots \mathsf{return}(M_4)\}\,.$$

In this example, the role role receives x_1, replies with M_2, then receives x_3 and replies with M_4. The adversary schedules this exchange by calling $O_1(M_1)$, getting M_2 as answer, then calling $O_2(M_3)$, and getting M_4 as answer.

The process for a role $Q(\mathsf{role})$ 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 $Q(\mathsf{role})$, 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. Using files is not the only possible implementation: we only need an implementation that provides persistent storage and guarantees that only our generated code has access to stored data. In particular, the adversary must not have access to stored data.) The user must also declare, for each free variable $x[\,]$ in a role, the file *file* in which the variable will be stored. Let *Files* be the set of these pairs $(x[\,], \mathit{file})$. Let also *Tables* be the set of pairs $(\mathit{Tbl}, \mathit{file})$ such that the table *Tbl* will be stored in file *file*.

Example 4.10 Let us annotate the protocol of Example 4.1.

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

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

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 $\mathbb{G}_{\mathsf{T}}(T)$ as well as several OCaml functions:

- The function $\mathbb{G}_{\mathsf{random}}(T) : \mathit{unit} \rightarrow \mathbb{G}_{\mathsf{T}}(T)$ generates random numbers uniformly in T (when T is used in a random number generation).
- The serialization function $\mathbb{G}_{\mathsf{ser}}(T) : \mathbb{G}_{\mathsf{T}}(T) \rightarrow \mathit{string}$ converts an element of type $\mathbb{G}_{\mathsf{T}}(T)$ to an OCaml string. The deserialization function $\mathbb{G}_{\mathsf{deser}}(T) : \mathit{string} \rightarrow \mathbb{G}_{\mathsf{T}}(T)$ 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 $\mathbb{G}_{\mathsf{pred}}(T) : \mathbb{G}_{\mathsf{T}}(T) \rightarrow \mathit{bool}$ 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 $f : T_1 \times \cdots \times T_m \rightarrow T$, a corresponding OCaml function $\mathbb{G}_{\mathsf{f}}(f) : \mathbb{G}_{\mathsf{T}}(T_1) \times \cdots \times \mathbb{G}_{\mathsf{T}}(T_m) \rightarrow \mathbb{G}_{\mathsf{T}}(T)$. We assume that these functions are all provided in an OCaml module μ_{prim}.

CryptoVerif verifies the following properties:

Property 4.11 *There is a single occurrence of each role* role*. If a role is defined after an oracle O, this oracle O must have globally at most one* return*, and must be in a role.*

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 4.12 *There are no nested roles.*

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

Assumption 4.13 *All oracle definitions are included in a role.*

This assumption is relaxed in the implementation: we accept all processes in which all oracles in a role are not preceded by oracles not in a role. In practice, oracles outside a role serve in representing features, such as corruption of protocol participants or registration of dishonest participants, that are needed in the proof of the security property but not in the implementation of the protocol. For instance, in Example 4.2, the oracle OR would typically not be included in a role. To extend our proof to the general case, if the process Q_0 does not satisfy Assumption 4.13, we transform it into a process Q'_0 that satisfies Assumption 4.13 by adding roles or by continuing existing roles till the end of the process instead of terminating them. The generated OCaml modules for Q'_0 contain unused OCaml code, which is not generated for Q_0. It is fairly obvious that removing this code preserves the security of the implementation.

Assumption 4.14 *No replication occurs above a parallel composition or a replication. When the definition of a role* role *is under replication* $\mathsf{foreach}\ i \leq N\ \mathsf{do}\ \mathsf{role}\{Q$*, its contents* Q *consists of an oracle definition* $O[\widetilde{i}](\dots) := \dots$ *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: $\mathsf{foreach}\ i \leq N\ \mathsf{do}\ (Q_1 \mid Q_2)$ can be transformed into $(\mathsf{foreach}\ i_1 \leq N\ \mathsf{do}\ Q_1) \mid (\mathsf{foreach}\ i_2 \leq N\ \mathsf{do}\ Q_2)$: both processes allow calling the oracles defined in Q_1 and Q_2 at most N times. We can encode nested replications by adding a dummy oracle between the two replications: the process $\mathsf{foreach}\ i \leq N\ \mathsf{do}\ \mathsf{foreach}\ j \leq N'\ \mathsf{do}\ Q$ can be transformed into $\mathsf{foreach}\ i \leq N\ \mathsf{do}\ O() := \mathsf{return}(); \mathsf{foreach}\ j \leq N'\ \mathsf{do}\ Q$.

By Properties 4.6, 4.5, and 4.11, 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 4.5, the bound of the replication above a definition of an oracle O is the same for all definitions of O.

Assumption 4.15 *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.*

After transforming the process so that it satisfies Assumption 4.14, we can transform it into a process that satisfies Assumption 4.15 by renaming the bounds of replications above distinct roles or oracles to distinct bounds. For instance, $(\mathsf{foreach}\ i_1 \leq N\ \mathsf{do}\ Q_1) \mid (\mathsf{foreach}\ i_2 \leq N\ \mathsf{do}\ Q_2)$ becomes $(\mathsf{foreach}\ i_1 \leq N_1\ \mathsf{do}\ Q_1) \mid (\mathsf{foreach}\ i_2 \leq N_2\ \mathsf{do}\ Q_2)$. Using distinct bounds for each oracle and role allows us to be more precise when counting the number of times an oracle has been called.

Assumption 4.14 and 4.15 are relaxed in our implementation: we warn the user when the process does not satisfy them, but we accept the process. Not heeding these warnings will lead to CryptoVerif returning imprecise, but sound, probabilities of security. We use these assumptions because they simplify the proof without losing much generality. Our result can be extended to the general case as follows: if the process Q_0 does not satisfy Assumption 4.14 or 4.15, we transform it into a process Q_0' that satisfies these assumptions as outlined after each assumption. We apply our theorem to Q_0' and argue that the implementation generated from Q_0 is also secure since it is basically the same as the one generated from Q_0'.

5 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 small step operational semantics of the core part of OCaml by Scott Owens et al. [17, 18].

5.1 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 $pat_1 \to e_1 \mid \dots \mid pat_n \to e_n$, we match v sequentially to the patterns $pat_1, \dots, pat_n$. 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 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 [17, 18]. It formalizes Assumption A1 that our implementation uses a perfect random number generator. 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 $\{0, \dots, 255\}$ 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 $()$, exceptions, and constant constructors. 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 pat-

$\mathit{pat} ::=$	pattern
x	variable
$\_$	universal pattern
$(\mathit{pat}_1, \ldots, \mathit{pat}_n)$	tuple
$\mathit{pat}_1 :: \mathit{pat}_2$	list constructor
$\mathit{pm} ::=$	pattern matching
$\mathit{pat}_1 \to e_1 \mid \ldots \mid \mathit{pat}_n \to e_n$	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 $\mathit{pat} = e_1$ **in** e_2	let
let rec $x_1 =$ **function** pm_1 **and** ... **and** $x_n =$ **function** pm_n **in** e	let rec
function$[\mathit{env}, \mathit{pm}]$	closure
letrec$[\mathit{env}, \{x_1 \mapsto$ **function** $\mathit{pm}_1, \ldots, x_n \mapsto$ **function** $\mathit{pm}_n\}$ **in** $x_i]$	let rec closure, $1 \leq i \leq n$
addthread$(\mathit{program})$	addition of a thread
schedule(e)	schedule
$d ::=$	definition
let $\mathit{pat} = e$	let
let rec $x_1 =$ **function** pm_1 **and** ... **and** $x_n =$ **function** pm_n	letrec
$\mathit{definitions} ::=$	definitions
ε	empty definition list
d;; $\mathit{definitions}$	definition list
$\mathit{program} ::=$	program
$\mathit{definitions}$	list of definitions
raise e	exception

Figure 5: OCaml syntax

$v ::=$	value
$\mathit{prim}\ v_1\ \dots\ v_j$	partially applied primitives (prim is n-ary and $0 \leq j < n$)
c	constant ([ ], (), 0, false, ...)
l	location
$(v_1, \dots, v_n)$	tuple
$v_1 :: v_2$	list constructor
$\textbf{function}[\mathit{env}, \mathit{pm}]$	closure
$\textbf{letrec}[\mathit{env}, \{x_1 \mapsto \textbf{function}\ \mathit{pm}_1, \dots, x_n \mapsto \textbf{function}\ \mathit{pm}_n\}\ \textbf{in}\ x_i]$	let rec closure

Figure 6: OCaml values

tern 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 $x_1, \dots, x_n$, 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 $\textbf{function}[\mathit{env}, \mathit{pm}]$ comes from the function **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 $\textbf{letrec}[\mathit{env}, \{x_1 \mapsto \textbf{function}\ \mathit{pm}_1, \dots, x_n \mapsto \textbf{function}\ \mathit{pm}_n\}\ \textbf{in}\ x_i]$ 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, **addthread**(*program*) and **schedule**(e). The expression **addthread**(*program*) creates a new thread that runs the program *program*. The expression **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 $[e_1; e_2; \dots; e_n]$ as syntactic sugar for $e_1 :: (e_2 ::$

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

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

$$\frac{\forall 1 \leq i \leq n, v_i \ \mathit{matches}\ \mathit{pat}_i \triangleright \mathit{env}_i}{(v_1, \dots, v_n)\ \mathit{matches}\ (\mathit{pat}_1, \dots, \mathit{pat}_n) \triangleright \bigoplus_{i=1}^{n} \mathit{env}_i} \qquad \text{(Tuple)}$$

$$\frac{v_1 \ \mathit{matches}\ \mathit{pat}_1 \triangleright \mathit{env}_1 \qquad v_2 \ \mathit{matches}\ \mathit{pat}_2 \triangleright \mathit{env}_2}{v_1 :: v_2 \ \mathit{matches}\ \mathit{pat}_1 :: \mathit{pat}_2 \triangleright \mathit{env}_1 \oplus \mathit{env}_2} \qquad \text{(List)}$$

Figure 7: Matches predicate

$\dots :: (e_n :: [\,])\dots)$. The expression $e \,\&\&\, e'$ is syntactic sugar for **if** e **then** e' **else false**, and $e \,||\, e'$ is syntactic sugar for **if** e **then true 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 functional values like closures, constants c, locations l, and tuples and lists of values. An exceptional value is **raise** v, where v is an exception value (a constant).

5.2 Formal Semantics

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

5.2.1 Pattern matching

We define the predicate *matches* in Figure 7: we have $v\ \mathit{matches}\ \mathit{pat} \triangleright \mathit{env}$ 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 $\mathit{env} \oplus \mathit{env}' \overset{\text{def}}{=} \mathit{env}_{|\overline{\mathit{Dom}(\mathit{env}')}} \cup \mathit{env}'$ 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 $\mathit{env} \oplus \mathit{env}'$ is always used with environments *env* and env' that have disjoint domains; the general case is used below. We also define $v\ \mathit{matches}\ \mathit{pat}$ as $\exists \mathit{env}, v\ \mathit{matches}\ \mathit{pat} \triangleright \mathit{env}$.

5.2.2 Primitives

The semantics of primitives is defined in Figure 8. This semantics is defined by rules of the form $\mathit{prim}\ v_1\ \dots\ v_n \xrightarrow{L}_p e$ where *prim* is an n-ary primitive. Such a rule means that $\mathit{prim}\ v_1\ \dots\ v_n$ reduces to e with probability p. In contrast to [17, 18], 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 5.1. 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.

5.2.3 Expressions and Programs

The semantics of [17, 18] 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_{m}), where C_{m} 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:

$C_{\mathsf{me}} ::=$	minimal expression evaluation context
$\quad e\ [\cdot]$	apply
$\quad [\cdot]\ v$	apply function
$\quad \textbf{let}\ pat = [\cdot]\ \textbf{in}\ e$	let
$\quad [\cdot]; e$	sequence
$\quad \textbf{if}\ [\cdot]\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	if
$\quad \textbf{match}\ [\cdot]\ \textbf{with}\ pm$	match
$\quad \textbf{try}\ [\cdot]\ \textbf{with}\ pm$	try
$\quad (e_1, \dots, e_{i-1}, [\cdot], v_{i+1}, \dots, v_n)$	tuple
$\quad e :: [\cdot]$	cons1
$\quad [\cdot] :: v$	cons2
$\quad \textbf{schedule}([\cdot])$	schedule
$C_{\mathsf{mp}} ::=$	minimal program evaluation context
$\quad \textbf{let}\ pat = [\cdot];;\ \textit{definitions}$	let

For example, we evaluate the argument of applications first, and when it becomes a value, we evaluate the function, so $e\ [\cdot]$ and $[\cdot]\ v$ are evaluation contexts. Tuples and lists are evaluated from right to left. We denote by $C_{\mathsf{me}}[e]$ the context C_{me} with the hole $[\cdot]$ replaced by e, and similarly for C_{mp}. The stack contains a minimal program evaluation context C_{mp} in the last element of the list and expression evaluation contexts C_{me} 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 5.2.2. These reductions are probabilistic; the probability p is omitted when it is 1. Most rules are straightforward. In order to evaluate an expression $C_{\mathsf{me}}[e]$, we need to reduce e under the context C_{me}. To do that, we push the context

Functional values:

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

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

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

Primitives:

$$\mathbf{not} \begin{cases} \mathbf{not\ false} \longrightarrow \mathbf{true} & \text{(Not1)} \\ \mathbf{not\ true} \longrightarrow \mathbf{false} & \text{(Not2)} \end{cases}$$

$$(=) \begin{cases} \dfrac{\mathit{funval}(v) \text{ or } \mathit{funval}(v')}{v = v' \longrightarrow \mathbf{raise\ Invalid\_argument}} & \text{(Funval)} \\ c = c \longrightarrow \mathbf{true} & \text{(Constant1)} \\ \dfrac{c \neq c'}{c = c' \longrightarrow \mathbf{false}} & \text{(Constant2)} \\ v_1 :: v_2 = v_1' :: v_2' \longrightarrow v_1 = v_1' \mathbin{\&\&} v_2 = v_2' & \text{(List1)} \\ v_1 :: v_2 = [\,] \longrightarrow \mathbf{false} & \text{(List2)} \\ [\,] = v_1' :: v_2' \longrightarrow \mathbf{false} & \text{(List3)} \\ (v_1, \ldots, v_n) = (v_1', \ldots, v_n') \longrightarrow v_1 = v_1' \mathbin{\&\&} \ldots \mathbin{\&\&} v_n = v_n' & \text{(Tuples)} \end{cases}$$

$$\mathbf{ref} \begin{cases} \mathbf{ref}\ v \xrightarrow{\mathbf{ref}\ v = l} l & \text{(New ref)} \end{cases}$$

$$(:=) \begin{cases} l := v \xrightarrow{l := v} () & \text{(Assign)} \end{cases}$$

$$! \begin{cases} !l \xrightarrow{!l = v} v & \text{(Dereference)} \end{cases}$$

$$\mathbf{random} \begin{cases} \dfrac{a \in \{\mathbf{true}, \mathbf{false}\}}{\mathbf{random}\ () \longrightarrow_{1/2} a} & \text{(Random)} \end{cases}$$

Figure 8: Rules for OCaml primitives

C_{me} 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_{me} from the stack by rule (Context out), inserting the obtained value v in C_{me}, yielding $C_{\mathsf{me}}[v]$. In case the expression e raises an exception v, we use rules (Context raise1) and (Context raise2). If the context C_{me} is not a **try** context, the result of $C_{\mathsf{me}}[e]$ is also **raise** v by (Context raise1). If C_{me} 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 5.1 Let us present as an example the reduction of a simple program in an empty environment and an empty stack:

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

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 insert this value back into the context. So we first reduce the previous configuration by (Let ctx in) into:

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

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

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

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

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

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

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

By (If1), we evaluate the **if**:

$$\emptyset, 0, [(\emptyset, \mathbf{let}\ x = [\cdot];;)]$$

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

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

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 5.2.5.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{l} env, \textbf{try raise}\ v\ \textbf{with}\ pm, stack \longrightarrow \\ env, \textbf{match}\ v\ \textbf{with}\ pm \mid \_ \to \textbf{raise}\ v, stack \end{array} \qquad \text{(Try2)}$$

Figure 9: Rules for expressions

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

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

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

(Closure let rec)

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

(Expr letrec apply)

Figure 10: Rules for expressions (continued)

5.2.4 Store

As usual, the contents of locations are stored in a *store*, which maps locations to their current values. Figure 12 defines the relation $\mathit{store} \xrightarrow{L} \mathit{store}'$. If a program or an expression reduces by $\mathit{env}, \mathit{pe}, \mathit{stack} \xrightarrow{L}_p \mathit{env}', \mathit{pe}', \mathit{stack}'$, then the store *store* will be updated into store' such that $\mathit{store} \xrightarrow{L} \mathit{store}'$. 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 $\textbf{ref}\ v = l$, 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).

5.2.5 Toplevel Reduction

As mentioned in Section 5.1, and in contrast to [17, 18], we consider several threads running in parallel. Each thread has a configuration $\mathit{th}_i = \langle \mathit{env}_i, \mathit{pe}_i, \mathit{stack}_i, \mathit{store}_i \rangle$ that contains the current $\mathit{env}_i, \mathit{pe}_i, \mathit{stack}_i$ as explained in Section 5.2.3, as well as the contents of locations local to this thread, in a store store_i, as explained in Section 5.2.4. The complete semantic configuration is then

$$\mathbb{C} = [\mathit{th}_1, \dots, \mathit{th}_n], \mathit{globalstore}, \mathit{tj}$$

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

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

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

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

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

$$\frac{\mathit{letrecenv} = \{x_1 \mapsto \textbf{function}\ \mathit{pm}_1, \ldots, x_n \mapsto \textbf{function}\ \mathit{pm}_n\}}{\begin{array}{l}\mathit{env}, \textbf{let rec}\ x_1 = \textbf{function}\ \mathit{pm}_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n = \textbf{function}\ \mathit{pm}_n;; \\ \quad \mathit{definitions}, [\,] \longrightarrow \mathit{env}[x_1 \mapsto \textbf{letrec}[\mathit{env}, \mathit{letrecenv}\ \textbf{in}\ x_1], \\ \qquad\qquad\qquad \ldots, \\ \qquad\qquad\qquad x_n \mapsto \textbf{letrec}[\mathit{env}, \mathit{letrecenv}\ \textbf{in}\ x_n]], \mathit{definitions}, [\,]\end{array}} \quad \text{(Closure let rec)}$$

Figure 11: Rules for programs

$$\mathit{store} \longrightarrow \mathit{store} \quad \text{(Store empty)}$$

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

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

$$\frac{l \notin \mathit{Dom}(\mathit{store})}{\mathit{store} \xrightarrow{\textbf{ref}\ v=l} \mathit{store}[l \mapsto v]} \quad \text{(Store alloc)}$$

Figure 12: Store rules

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 A5, since it may bypass the type system.) The global store contains locations in a set $Loc_{\mathtt{g}}$, while the local stores contain locations in an infinite set Loc_ℓ, with $Loc_{\mathtt{g}} \cap Loc_\ell = \emptyset$.

The reduction rules for semantic configurations $\mathbb{C}$ are defined in Figure 13. Actually, this figure defines three relations. The relation $th \to_p th'$, 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 Loc_ℓ.)

Second, the relation $th, globalstore \to_p th', globalstore'$, 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 $th \to_p th'$ 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 $Loc_{\mathtt{g}}$, 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 $\mathbb{C} \to_p \mathbb{C}'$, 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 $th, globalstore \to_p th', globalstore'$. 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, as mentioned in Assumption A7. Moreover, we reduce only the active thread, and we change threads only with **schedule**. Since neither the primitives nor the generated modules use **schedule**, thread scheduling is entirely under the control of the adversary. This seemingly restrictive semantics, in which only one thread is active at a time and oracle calls cannot be interleaved with other threads, is justified for two reasons.

- First, it is sufficient to represent all program executions under the weaker assumption that two threads that read or write the same file are not run concurrently. Indeed, two oracles can interfere with each other only through files, and such interferences are forbidden by this assumption. Hence, by swapping execution steps, a trace that obeys this assumption with any interleaving of the oracle calls can be transformed into an equivalent trace in which the oracle calls are never interrupted, that is, a trace that can be scheduled in our semantics.

- Second, it resembles the CryptoVerif semantics, which also has a single active thread and processes one oracle call after the other. This point facilitates the proof of our compiler.

5.2.6 Modules

OCaml programs typically contain several modules. We adopt a very simple model of modules. A module named μ consists of an OCaml program $\mathit{program}(\mu)$ and its interface $\mathit{interface}(\mu)$ that is the set of OCaml identifiers defined in μ and usable in other modules. The program $\mathit{program}(\mu)$ initializes the module μ and makes available the identifiers defined in the interface of the module μ. 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 $\mathit{program} = \mathit{program}(\mu_1);; \ldots ;; \mathit{program}(\mu_n);;$, where, for all $i \leq n$, the free variables of $\mathit{program}(\mu_i)$ are defined in the interfaces of μ_j with $j < i$, and $\mathit{program}(\mu_i)$ is a list of definitions. (The initial program of a module is never **raise** e, but it may reduce into **raise** e during execution.)

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

$$\mathbb{C}_0(\mathit{program}) = [\langle \emptyset, \mathit{program}, [\,], \emptyset \rangle], \mathit{globalstore}_0, 1$$

where $\mathit{globalstore}_0 = \{l \mapsto \mathit{initval}_l \mid l \in \mathit{Loc}_{\mathrm{g}}\}$ is the initial value of the global store, and $\mathit{initval}_l$ is the default value for location l: the empty list $[\,]$ for lists, the empty string `""` for strings, 0 for integers, **false** for booleans. Values in the global store cannot contain locations and closures, so we do not define a default value for them. The program $\mathit{program}$ does not contain closures nor locations in Loc_ℓ, but may contain locations in $\mathit{Loc}_{\mathrm{g}}$. (Closures are created by **function** and **let rec**; locations in Loc_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).

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

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

$$\frac{\begin{array}{l} \mathit{globalstore} \xrightarrow{L} \mathit{globalstore}' \\ L \text{ is } !l = v \text{ or } l := v \text{ with } l \in \mathit{Loc}_{\mathtt{g}} \\ \mathit{env}, \mathit{pe}, \mathit{stack} \xrightarrow{L}_p \mathit{env}', \mathit{pe}', \mathit{stack}' \end{array}}{\langle \mathit{env}, \mathit{pe}, \mathit{stack}, \mathit{store} \rangle, \mathit{globalstore} \longrightarrow_p \langle \mathit{env}', \mathit{pe}', \mathit{stack}', \mathit{store} \rangle, \mathit{globalstore}'}$$

(Globalstore2)

$$\frac{\mathit{th}, \mathit{globalstore} \longrightarrow_p \mathit{th}', \mathit{globalstore}'}{\begin{array}{l} [\mathit{th}_1, \ldots, \mathit{th}_{tj-1}, \mathit{th}, \mathit{th}_{tj+1}, \ldots, \mathit{th}_n], \mathit{globalstore}, \mathit{tj} \longrightarrow_p \\ [\mathit{th}_1, \ldots, \mathit{th}_{tj-1}, \mathit{th}', \mathit{th}_{tj+1}, \ldots, \mathit{th}_n], \mathit{globalstore}', \mathit{tj} \end{array}} \quad \text{(Toplevel)}$$

$$\begin{array}{l} [\mathit{th}_1, \ldots, \mathit{th}_{tj-1}, \langle \mathit{env}, \mathbf{addthread}(\mathit{program}), \mathit{stack}, \mathit{store} \rangle, \mathit{th}_{tj+1}, \ldots, \mathit{th}_n], \\ \quad \mathit{globalstore}, \mathit{tj} \longrightarrow \\ [\mathit{th}_1, \ldots, \mathit{th}_{tj-1}, \langle \mathit{env}, (), \mathit{stack}, \mathit{store} \rangle, \mathit{th}_{tj+1}, \ldots, \mathit{th}_n, \langle \emptyset, \mathit{program}, [\,], \emptyset \rangle], \\ \quad \mathit{globalstore}, \mathit{tj} \end{array}$$

(Toplevel add thread)

$$\frac{1 \leq \mathit{tj}' \leq n}{\begin{array}{l} [\mathit{th}_1, \ldots, \mathit{th}_{tj-1}, \langle \mathit{env}, \mathbf{schedule}(\mathit{tj}'), \mathit{stack}, \mathit{store} \rangle, \mathit{th}_{tj+1}, \ldots, \mathit{th}_n], \\ \quad \mathit{globalstore}, \mathit{tj} \longrightarrow \\ [\mathit{th}_1, \ldots, \mathit{th}_{tj-1}, \langle \mathit{env}, (), \mathit{stack}, \mathit{store} \rangle, \mathit{th}_{tj+1}, \ldots, \mathit{th}_n], \mathit{globalstore}, \mathit{tj}' \end{array}}$$

(Toplevel schedule1)

$$\frac{\mathit{tj}' < 1 \text{ or } \mathit{tj}' > n}{\begin{array}{l} [\mathit{th}_1, \ldots, \mathit{th}_{tj-1}, \langle \mathit{env}, \mathbf{schedule}(\mathit{tj}'), \mathit{stack}, \mathit{store} \rangle, \mathit{th}_{tj+1}, \ldots, \mathit{th}_n], \\ \quad \mathit{globalstore}, \mathit{tj} \longrightarrow \\ [\mathit{th}_1, \ldots, \mathit{th}_{tj-1}, \langle \mathit{env}, \mathbf{raise\ Invalid\_argument}, \mathit{stack}, \mathit{store} \rangle, \mathit{th}_{tj+1}, \ldots, \mathit{th}_n], \\ \quad \mathit{globalstore}, \mathit{tj} \end{array}}$$

(Toplevel schedule2)

Figure 13: Top level rules

5.2.7 Equivalence Modulo Renaming of Locations

The rule (Store alloc) is non-deterministic, since the new location l can be any unused location in Loc_ℓ. To remove this non-determinism, we consider equivalence classes of OCaml semantic configurations modulo renaming of locations in Loc_ℓ. We still denote these equivalence classes as OCaml configurations $\mathbb{C}$, 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 $\mathbb{C}$ can reduce, then the sum of the probabilities of all possible reductions is 1:

$$\sum_{\{\mathbb{C}' \mid \mathbb{C} \rightarrow_{p(\mathbb{C}')} \mathbb{C}'\}} p(\mathbb{C}') = 1$$

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

We will also use notations similar to Definition 4.8 for the OCaml semantics. We denote by $\mathbb{CT}$ an OCaml trace, $\mathbb{CTS}$ a set of OCaml traces, and we also use the notation $\rightarrow^*$ for reductions with several steps.

6 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 $\mathbf{tagfunction}^t\ pm$ for tagged functions and $\mathbf{tagfunction}^{t,\tau}[env, pm]$ for the corresponding closures. We also add $\mathbf{tagfunction}^{t,\tau}[env, pm]$ 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 8 to count the number of calls to the same 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 **event** $ev(e_1, \dots, e_k)$ that adds the event $ev(v_1, \dots, v_k)$ to *events* when $e_1, \dots, e_k$ evaluate to the values $v_1, \dots, v_k$ respectively. We consider a new minimal expression evaluation

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

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

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

Figure 14: Semantics of tagged functions

context **event** $ev(e_1, \dots, e_{i-1}, [\cdot], v_{i+1}, \dots, v_n)$, for evaluating the arguments of events via rules (Context in) and (Context out), and for handling exceptions inside events via rule (Context raise1). 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 $\mathbb{MI}$ that contains pairs $(\mu_{\mathsf{role}}, \gamma)$ of a module μ_{role} and a flag $\gamma \in \{\mathsf{Once}, \mathsf{Any}\}$, indicating, if Once, that the module can be called only once and if Any that the module can be called any number of times. Hence, the instrumented semantic configuration is

$$\mathbb{CI} = [\mathit{th}_1, \dots, \mathit{th}_n], \mathit{globalstore}, \mathit{tj}, \mathbb{MI}, \mathit{events}$$

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 $\mathbb{MI}, \mathit{events}$. The rule (Toplevel event) gives the semantics of **event**: it adds its argument $ev(v_1, \dots, v_n)$ to the list *events* in the configuration and returns $(v_1, \dots, v_n)$. The rule (New toplevel add thread) gives the instrumented semantics of **addthread**: the **addthread** construct is modified to reject new programs that contain a module that cannot be called. We let $\mathbb{M}_{\mathsf{g}}$ 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 $\mathbb{M}_{\mathsf{g}}$, or they are *protocol programs* that first contain the module corresponding to the primitives μ_{prim}, then the necessary generated modules $\mu_1, \dots, \mu_l$ in $\mathbb{M}_{\mathsf{g}}$, and finally any non-generated program *program*$'$. (We require this order on the modules for simplicity.) The generated modules $\mu_1, \dots, \mu_l$ must be callable according to the value of $\mathbb{MI}$. The

$$\frac{th, \mathit{globalstore} \longrightarrow_p th', \mathit{globalstore}'}{\begin{array}{l}[th_1, \ldots, th_{tj-1}, th, th_{tj+1}, \ldots, th_n], \mathit{globalstore}, tj, \mathbb{MI}, \mathit{events} \longrightarrow_p \\ [th_1, \ldots, th_{tj-1}, th', th_{tj+1}, \ldots, th_n], \mathit{globalstore}', tj, \mathbb{MI}, \mathit{events}\end{array}}$$

(New toplevel)

$$\frac{\begin{array}{l}\mathit{program} = \mathit{program}(\mu_{\mathsf{prim}});;\ \mathit{program}(\mu_1);;\ \ldots;;\ \mathit{program}(\mu_l);;\ \mathit{program}' \\ \mathit{program}' \text{ does not contain } \mathit{program}(\mu_{\mathsf{prim}}) \text{ nor any } \mathit{program}(\mu) \text{ for } \mu \in \mathbb{M}_{\mathsf{g}} \\ \mathbb{M} = \{\mu_1, \ldots, \mu_l\} \subseteq \mathbb{M}_{\mathsf{g}} \\ \forall \mu \in \mathbb{M}, \exists \gamma, (\mu, \gamma) \in \mathbb{MI} \\ \mathbb{MI}' = \{(\mu, \mathsf{Once}) \mid \mu \in \mathbb{M} \wedge (\mu, \mathsf{Once}) \in \mathbb{MI}\} \\ \text{or} \\ \mathit{program} \text{ does not contain } \mathit{program}(\mu_{\mathsf{prim}}) \text{ nor any } \mathit{program}(\mu) \text{ for } \mu \in \mathbb{M}_{\mathsf{g}} \\ \mathbb{M} = \emptyset, \mathbb{MI}' = \emptyset\end{array}}{\begin{array}{l}[th_1, \ldots, th_{tj-1}, \langle \mathit{env}, \mathbf{addthread}(\mathit{program}), \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj, \mathbb{MI}, \mathit{events} \longrightarrow \\ [th_1, \ldots, th_{tj-1}, \langle \mathit{env}, (), \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n, \langle \emptyset, \mathit{program}, [\,], \emptyset\rangle], \\ \quad \mathit{globalstore}, tj, \mathbb{MI} \setminus \mathbb{MI}', \mathit{events}\end{array}}$$

(New toplevel add thread)

$$\frac{1 \leq tj' \leq n}{\begin{array}{l}[th_1, \ldots, th_{tj-1}, \langle \mathit{env}, \mathbf{schedule}(tj'), \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj, \mathbb{MI}, \mathit{events} \longrightarrow \\ [th_1, \ldots, th_{tj-1}, \langle \mathit{env}, (), \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj', \mathbb{MI}, \mathit{events}\end{array}}$$

(New toplevel schedule1)

$$\frac{tj' < 1 \text{ or } tj' > n}{\begin{array}{l}[th_1, \ldots, th_{tj-1}, \langle \mathit{env}, \mathbf{schedule}(tj'), \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj, \mathbb{MI}, \mathit{events} \longrightarrow \\ [th_1, \ldots, th_{tj-1}, \langle \mathit{env}, \mathbf{raise\ Invalid\_argument}, \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj, \mathbb{MI}, \mathit{events}\end{array}}$$

(New toplevel schedule2)

$$\begin{array}{l}[th_1, \ldots, th_{tj-1}, \langle \mathit{env}, \mathbf{return}(\mathbb{MI}', v), \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj, \mathbb{MI}, \mathit{events} \longrightarrow \\ [th_1, \ldots, th_{tj-1}, \langle \mathit{env}, v, \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj, \mathbb{MI} \cup \mathbb{MI}', \mathit{events}\end{array}$$

(Toplevel return)

$$\begin{array}{l}[th_1, \ldots, th_{tj-1}, \langle \mathit{env}, \mathbf{event}\ \mathit{ev}(v_1, \ldots, v_n), \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj, \mathbb{MI}, \mathit{events} \longrightarrow \\ [th_1, \ldots, th_{tj-1}, \langle \mathit{env}, (v_1, \ldots, v_n), \mathit{stack}, \mathit{store}\rangle, th_{tj+1}, \ldots, th_n], \\ \quad \mathit{globalstore}, tj, \mathbb{MI}, \mathit{ev}(v_1, \ldots, v_n) :: \mathit{events}\end{array}$$

(Toplevel event)

Figure 15: Updated toplevel rules for the instrumented semantics

modules $\mu_1, \ldots, \mu_l$ that can be called only once are removed from the callable modules by removing the multiset $\mathbb{MI}'$ from $\mathbb{MI}$.

We also add the expression **return**$(\mathbb{MI}', e)$ that adds to the multiset $\mathbb{MI}$ the generated modules present in $\mathbb{MI}'$, 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 **return**$(\mathbb{MI}, [\cdot])$ 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 A3. This assumption is formalized as follows:

Assumption 6.1 (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 **raise** v, the environment does not matter. To prove this property, we define the following equivalence.

Definition 6.2 *We define the equivalence $\approx_{vth}$ on threads by*

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

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

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

- $\mathbb{C} = [th_1, \ldots, th_n], globalstore, tj$,
- $\mathbb{C}' = [th'_1, \ldots, th'_n], globalstore, tj$,
- $\forall tj' \leq n, th_{tj'} \approx_{vth} th'_{tj'}$.

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

Lemma 6.3 *If $\mathbb{C} \approx_v \mathbb{C}'$ and $\mathbb{C} \to_p \mathbb{C}''$, then $\mathbb{C}' \to_p \mathbb{C}'''$ and $\mathbb{C}'' \approx_v \mathbb{C}'''$.*

We prove this lemma in Appendix B.

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

Definition 6.4 *The function $noinstr_{th1}$ applied to a thread replaces*

1. *every* **return**$(\mathbb{MI}, e)$ *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 $noinstr_{th2}$ modifies the stack of the thread by

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

Let $noinstr_{th} \overset{\text{def}}{=} noinstr_{th1} \circ noinstr_{th2}$.

Finally, let us define

$$\begin{gathered} noinstr_{\mathbb{CI}}([th_1, \dots, th_n], globalstore, tj, \mathbb{MI}, events) \overset{\text{def}}{=} \\ [noinstr_{th}(th_1), \dots, noinstr_{th}(th_n)], globalstore, tj \end{gathered}$$

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 6.1, 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 6.5 *1. If $\mathbb{C} \approx_v noinstr_{\mathbb{CI}}(\mathbb{CI})$ and $\mathbb{C}_1, \dots, \mathbb{C}_n$ are pairwise distinct configurations such that for all $i \leq n$, we have $\mathbb{C} \rightarrow_{p_i} \mathbb{C}_i$ with $\sum_{i \leq n} p_i = 1$, then there exist pairwise distinct instrumented configurations $\mathbb{CI}_1, \dots, \mathbb{CI}_n$ such that for all $i \leq n$, we have $\mathbb{CI} \rightarrow^*_{p_i} \mathbb{CI}_i$ and $\mathbb{C}_i \approx_v noinstr_{\mathbb{CI}}(\mathbb{CI}_i)$.*

2. If $\mathbb{C} \approx_v noinstr_{\mathbb{CI}}(\mathbb{CI})$ and $\mathbb{CI}_1, \dots, \mathbb{CI}_n$ are pairwise distinct instrumented configurations such that for all $i \leq n$, we have $\mathbb{CI} \rightarrow_{p_i} \mathbb{CI}_i$ with $\sum_{i \leq n} p_i = 1$, then there exist pairwise distinct configurations $\mathbb{C}_1, \dots, \mathbb{C}_n$ such that for all $i \leq n$, we have $\mathbb{C} \rightarrow^*_{p_i} \mathbb{C}_i$ and $\mathbb{C}_i \approx_v noinstr_{\mathbb{CI}}(\mathbb{CI}_i)$.

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

7 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.

Let us recall that the function $\mathbb{G}_{\mathsf{f}}(f)$, defined in Section 4.3, returns the name of the OCaml function corresponding to the CryptoVerif function f. The function $\mathbb{G}_{\mathsf{M}}$ transforms a CryptoVerif term M into an OCaml term. It is defined as follows:

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

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

The OCaml code generated by this definition matches the semantics of CryptoVerif terms given in Figure 3.

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 $T_M \overset{\text{def}}{=} T_x$, and if M is a function application with a function of type $T_1 \times \cdots \times T_n \to T$, then $T_M \overset{\text{def}}{=} T$.

We say that an oracle or role definition occurs *at the beginning of* Q when it is found in Q just under replication or parallel composition, without recursively looking into oracle definitions. We define the function *oracledeflist* that returns a description of the oracles made available by an oracle definition Q. In more detail, *oracledeflist*(Q) is a list $[(Q_1, \gamma_1), \dots, (Q_l, \gamma_l)]$ such that $Q_1, \dots, Q_l$ are the oracle definitions at the beginning of Q, from left to right, and γ_l is Any when Q_l is under replication, and Once otherwise. In this function, the replication indices $\widetilde{i}$ can be partially instantiated into integer values. In contrast to the function *oracledefset*, *oracledeflist*$(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q)$ does not instantiate the replication index i.

$$\mathit{oracledeflist}(0) \overset{\text{def}}{=} [\,] \qquad \text{(Nil)}$$

$$\mathit{oracledeflist}(Q_1 \mid Q_2) \overset{\text{def}}{=} \mathit{oracledeflist}(Q_1) \mathbin{@} \mathit{oracledeflist}(Q_2) \qquad \text{(Par)}$$

$$\mathit{oracledeflist}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) \overset{\text{def}}{=} [(Q_1, \mathsf{Any}), \dots, (Q_l, \mathsf{Any})]$$
$$\text{when } \mathit{oracledeflist}(Q) = [(Q_1, \gamma_1), \dots, (Q_l, \gamma_l)] \text{ for some } \gamma_1, \dots, \gamma_l \qquad \text{(Repl)}$$

$$\mathit{oracledeflist}(O[\widetilde{i}](x_1[\widetilde{i}], \dots, x_k[\widetilde{i}]) := P) \overset{\text{def}}{=} [(O[\widetilde{i}](x_1[\widetilde{i}], \dots, x_k[\widetilde{i}]) := P, \mathsf{Once})] \qquad \text{(Oracle)}$$

$$\mathit{oracledeflist}(\mathsf{role}\,\{Q) \overset{\text{def}}{=} [\,] \qquad \text{(Role)}$$

The function *oracledeflist* takes processes Q that follow return statements that do not end a role. By Assumption 4.13, we are inside a role, so by Property 4.12, the construct $\mathsf{role}\,\{Q'$ cannot appear in Q before a return statement that ends the current oracle. So, the function *oracledeflist* will never be called on $\mathsf{role}\,\{Q'$.

We also define the function $\mathbb{G}_{\mathsf{get}\mathbb{MI}}$ that returns a description of the modules that correspond to roles defined at the beginning of an oracle definition Q. The function $\mathbb{G}_{\mathsf{get}\mathbb{MI}}$ is similar to the function *oracledeflist* 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 *oracledeflist*, it returns a set and not a list.

$$\mathbb{G}_{\mathsf{get}\mathbb{MI}}(0) \overset{\text{def}}{=} \emptyset \qquad \text{(Nil)}$$

$$\mathbb{G}_{\mathsf{getM}\mathbb{I}}(Q_1 \mid Q_2) \overset{\text{def}}{=} \mathbb{G}_{\mathsf{getM}\mathbb{I}}(Q_1) \cup \mathbb{G}_{\mathsf{getM}\mathbb{I}}(Q_2) \tag{Par}$$

$$\mathbb{G}_{\mathsf{getM}\mathbb{I}}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) \overset{\text{def}}{=} \{(\mu, \mathsf{Any}) \mid \exists \gamma, (\mu, \gamma) \in \mathbb{G}_{\mathsf{getM}\mathbb{I}}(Q)\} \tag{Repl}$$

$$\mathbb{G}_{\mathsf{getM}\mathbb{I}}(O[\widetilde{i}](x_1[\widetilde{i}], \dots, x_k[\widetilde{i}]) := P) \overset{\text{def}}{=} \emptyset \tag{Oracle}$$

$$\mathbb{G}_{\mathsf{getM}\mathbb{I}}(\mathsf{role}\,\{Q) \overset{\text{def}}{=} \{(\mu_{\mathsf{role}}, \mathsf{Once})\} \tag{Role}$$

The function $\mathbb{G}_{\mathsf{getM}\mathbb{I}}$ takes processes Q that follow return statements that end the current role. By Assumption 4.13, there cannot be an oracle definition outside a role $\{Q'$ in Q. So the function $\mathbb{G}_{\mathsf{getM}\mathbb{I}}$ 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: the function $\mathbb{G}$ generates OCaml code that encodes the semantics of oracle bodies given in Figures 3 and 4. After defining a variable, we store it in a file if needed, using $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}])$, defined by $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]) \overset{\text{def}}{=} (f := \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x))$ if $(x[\widetilde{i}], f) \in \mathit{Files}$ and $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]) \overset{\text{def}}{=} ()$ 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 (i.e., there is an oracle in the same role following it), we return the closures corresponding to the oracles defined after the return, as defined in (Return1). (The function $\mathbb{G}_{\mathsf{O}}$ is defined below, in Figure 17.) Otherwise, we update the set of available roles using the **return** expression introduced in Section 6, 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 $\mathbf{random}_\ell$ such that $\mathbf{random}_\ell\ l$ returns a random element of the list l, such that the probability of returning the j-th element of l is $\mathit{almostunif}(\{1, \dots, |l|\}, j)$. We assume that this function is programmed using the OCaml primitive **random**, and is present in the module for cryptographic primitives μ_{prim}.

An oracle $O(x_1, \dots, x_n) := P$ 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 (the second argument of $\mathbb{G}_{\mathsf{O}}$ is Once, in (Oracle1)), we use a token *token* to make sure that it can be called only once. This token can take the values **Callable** and **Invalid**. It is initially set to **Callable**, and it is set to **Invalid** 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

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

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

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

$$\frac{[(Q_1,\gamma_1),\dots,(Q_l,\gamma_l)] \stackrel{\text{def}}{=} \mathit{oracledeflist}(Q)}{\mathbb{G}(\mathsf{return}(N_1,\dots,N_k); Q) \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{O}}(Q_1,\gamma_1),\dots,\mathbb{G}_{\mathsf{O}}(Q_l,\gamma_l), \mathbb{G}_{\mathsf{M}}(N_1),\dots,\mathbb{G}_{\mathsf{M}}(N_k))} \qquad \text{(Return1)}$$

$$\mathbb{G}(\mathsf{return}(N_1,\dots,N_k)\}; Q) \stackrel{\text{def}}{=} (\textbf{return}(\mathbb{G}_{\mathsf{getM}\mathbb{I}}(Q), (\mathbb{G}_{\mathsf{M}}(N_1),\dots,\mathbb{G}_{\mathsf{M}}(N_k)))) \qquad \text{(Return2)}$$

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

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

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

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

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

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

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

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

```
$\mathbb{G}_{\mathsf{O}}(Q, \mathsf{Once}) \overset{\text{def}}{=}$ let $token$ = ref Callable in tagfunction$^O$ $pm_{\mathsf{Once}}(Q)$
where $pm_{\mathsf{Once}}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k) := P) \overset{\text{def}}{=}$
        $(\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k)) \rightarrow$
        if $(!token = \textbf{Callable})$ &&
            $(\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))$ && ... && $(\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k))$
        then $(token := \textbf{Invalid}; \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \dots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P))$
        else raise Bad_Call
                                                                                   (Oracle1)
```

```
$\mathbb{G}_{\mathsf{O}}(Q, \mathsf{Any}) \overset{\text{def}}{=}$ tagfunction$^O$ $pm_{\mathsf{Any}}(Q)$
where $pm_{\mathsf{Any}}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k) := P) \overset{\text{def}}{=}$
        $(\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k)) \rightarrow$
        if $(\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))$ && ... && $(\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k))$          (Oracle2)
        then $(\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \dots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P))$
        else raise Bad_Call
```

Figure 17: Translation of an oracle

interface is $interface(\mu_{\mathsf{role}}) \overset{\text{def}}{=} \{\mu_{\mathsf{role}}.init\}$ and its program is

```
$program(\mu_{\mathsf{role}}) \overset{\text{def}}{=}$
    let $\mu_{\mathsf{role}}.init$ = let $token$ = ref Callable in tagfunction$^{\mathsf{role}}$ $pm_{\mathsf{role}}$
where $pm_{\mathsf{role}} \overset{\text{def}}{=} () \rightarrow$
    if $(!token = \textbf{Callable})$ then
        $(token := \textbf{Invalid};$
        $\mathbb{G}_{\mathsf{read}}(x_1[\,])$ in ... in $\mathbb{G}_{\mathsf{read}}(x_m[\,])$ in
        $(\mathbb{G}_{\mathsf{O}}(Q_1, \gamma_1), \dots, \mathbb{G}_{\mathsf{O}}(Q_k, \gamma_k)))$
    else raise Bad_Call
```

where $[(Q_1, \gamma_1), \dots, (Q_k, \gamma_k)] = \mathit{oracledeflist}(Q(\mathsf{role}))$ and $x_1[\,], \dots, x_m[\,]$ 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 $\mathbb{G}_{\mathsf{read}}(x[\,]) \overset{\text{def}}{=}$ **let** $\mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{deser}}(T_x)\ (!f)$ if $(x[\,], f) \in \mathit{Files}$.

Example 7.1 Let us explain the translation of the role keygen described in Examples 4.1 and 4.10. This role contains the following oracle

$$\mathrm{Okeygen}() := rk \overset{R}{\leftarrow} keyseed; pk \leftarrow \mathrm{pkgen}(rk); sk \leftarrow \mathrm{skgen}(rk); \mathsf{return}(pk)$$

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

is:

```
let $\mu_{\mathsf{keygen}}.init$ = let $token$ = ref Callable in tagfunction$^{\mathsf{keygen}}$ $() \to$
    if (!$token$ = Callable) then
        ($token$ := Invalid;
         let $token$ = ref Callable in tagfunction$^{\mathrm{Okeygen}}$ $() \to$
             if (!$token$ = Callable) then
                 ($token$ := Invalid;
                  let $\mathbb{G}_{\mathsf{var}}(rk) = \mathbb{G}_{\mathsf{random}}(keyseed)$ () in
                  let $\mathbb{G}_{\mathsf{var}}(pk) = \mathbb{G}(\mathrm{pkgen})\ \mathbb{G}_{\mathsf{var}}(rk)$ in
                  $pkfile := \mathbb{G}_{\mathsf{ser}}(T_{pk})\ \mathbb{G}_{\mathsf{var}}(pk)$;
                  let $\mathbb{G}_{\mathsf{var}}(sk) = \mathbb{G}(\mathrm{skgen})\ \mathbb{G}_{\mathsf{var}}(rk)$ in
                  $skfile := \mathbb{G}_{\mathsf{ser}}(T_{sk})\ \mathbb{G}_{\mathsf{var}}(sk)$;
                  return($\{(\mu_{\mathsf{alice}}, \mathsf{Any}), (\mu_{\mathsf{bob}}, \mathsf{Any})\}, \mathbb{G}_{\mathsf{var}}(pk)$))
             else raise Bad_Call)
    else raise Bad_Call
```

This program defines the function $\mu_{\mathsf{keygen}}.init$, which expects () as argument and returns the function that implements oracle Okeygen. This function itself expects () as argument and returns the OCaml representation of the public key pk returned by Okeygen.

The function $\mu_{\mathsf{keygen}}.init$ can be called only once, which is guaranteed using a reference *token* to either **Callable** or **Invalid**. If *token* is already **Invalid**, then this function has already been called, so we raise the exception **Bad_Call**. Otherwise, we set *token* to **Invalid** and we continue by the translation of the CryptoVerif oracle Okeygen. The oracle Okeygen can also be called only once. So we define a new reference *token*, to guarantee this property, and define the function that implements Okeygen. When this function is called for the first time, it sets this second *token* to **Invalid**, creates a new key seed $\mathbb{G}_{\mathsf{var}}(rk)$, computes the keys *pk* and *sk* and stores them into files, modeled by the global store references *pkfile* and *skfile*. Finally, it returns the public key $\mathbb{G}_{\mathsf{var}}(pk)$. Since the oracle Okeygen ends the role keygen, and is followed by the roles alice and bob, we update the set of callable modules $\mathbb{MI}$ with the newly defined modules μ_{alice} and μ_{bob}, which can be called any number of times, using the **return** expression. When this function is called again, it raises the exception **Bad_Call**.

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

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

This code first initializes the role by calling $\mu_{\mathsf{keygen}}.init$ (), which returns a closure corresponding to the translation of Okeygen, and then calls this closure.

The generated modules $\mathbb{M}_{\mathsf{g}}$ (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 $\mathit{program}_0$. We consider that the programs of the protocol are launched by the adversary $\mathit{program}_0$ 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 $\mathit{program}(\mu_{\mathsf{prim}});;\ \mathit{program}(\mu_{\mathsf{role}_1});;\ \dots;;\ \mathit{program}(\mu_{\mathsf{role}_k});;\ \mathit{program}'$ where $\mathit{program}'$ contains no generated module, as required by the instrumented semantics of **addthread** (New toplevel add thread). We assume that $\mathit{program}_0$ uses the generated modules only inside **addthread**. Moreover, the network code is well-typed by Assumption A5. Well-typed OCaml with **random** is probabilistic Turing complete, so the adversary itself can be implemented by a well-typed OCaml program. Therefore, we can assume that $\mathit{program}_0$ is a well-typed OCaml program. (Our OCaml programs include **random** and exclude type-casting and other constructs that allow to bypass the type system, as defined in Section 5.) 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 A4, we suppose that only the generated modules access files that contain private CryptoVerif data (free variables of roles and tables). So we let $\mathit{Loc}_{\mathsf{priv}} \stackrel{\text{def}}{=} \{f \mid (x[\,], f) \in \mathit{Files} \text{ or } (\mathit{Tbl}, f) \in \mathit{Tables}\} \subseteq \mathit{Loc}_{\mathsf{g}}$ be the set of global locations reserved for private CryptoVerif data, and we have the following assumption:

Assumption 7.2 *The locations in $\mathit{Loc}_{\mathsf{priv}}$ occur only in the programs of generated modules; they do not occur elsewhere in $\mathit{program}_0$.*

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

$$\mathbb{C}_0(Q_0, \mathit{program}_0) \stackrel{\text{def}}{=} [\langle \emptyset, \mathit{program}_0, [\,], \emptyset\rangle], \mathit{globalstore}_0, 1, \mathbb{G}_{\mathsf{get}\mathbb{M}\mathbb{I}}(Q_0), [\,]$$

where $\mathbb{G}_{\mathsf{get}\mathbb{M}\mathbb{I}}(Q_0)$ is the set of modules available at the beginning of the execution and $\mathit{globalstore}_0 \stackrel{\text{def}}{=} \{l \mapsto \mathit{initval}_l \mid l \in \mathit{Loc}_{\mathsf{g}}\}$ is the initial value of the global store as defined in Section 5.2. Tables are represented by lists, and their initial value $\mathit{initval}_l$ 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 $\mathit{initval}_l$ is the empty string "". For other elements, the initial value $\mathit{initval}_l$ is the default value for the type of location l.

Example 7.3 Let us consider the following toy OCaml program $\mathit{program}_0$, which uses the translation from Example 7.1 of the process given in Example 4.10.

```
    let _ =
      addthread($program(\mu_{\mathsf{prim}})$;; $program(\mu_{\mathsf{keygen}})$;;
                    let _ = $pkg := \mu_{\mathsf{keygen}}.init$ () (); schedule(1);; );
      schedule(2);;
```

This example only creates a thread for key generation, then schedules it by **schedule**(2). This thread stores the public key returned by the oracle Okeygen in the global store location *pkg*, and returns control to the initial thread by **schedule**(1).

Following the annotations of Example 4.10, this example uses two private global store locations, *skfile* and *pkfile*, to store the private and public keys, so $Loc_{\mathsf{priv}} = \{skfile, pkfile\}$. It also uses the global store location *pkg*, so $Loc_{\mathsf{g}} = Loc_{\mathsf{priv}} \cup \{pkg\}$. Assuming keys are represented by strings, the initial global store is $globalstore_0 = \{skfile \mapsto \texttt{""}, pkfile \mapsto \texttt{""}, pkg \mapsto \texttt{""}\}$. The initial set of available modules is $\mathbb{MI}_0 = \mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q_0) = \{(\mu_{\mathsf{keygen}}, \mathsf{Once})\}$, and the initial configuration is $\mathbb{C}_0(Q_0, program_0) = [\langle\emptyset, program_0, [\,], \emptyset\rangle], globalstore_0, 1, \mathbb{MI}_0, [\,]$.

Detailing the reductions of this configuration would take too much space, but we still give some information on the configuration obtained after evaluating $\mu_{\mathsf{keygen}}.init$ (). We use this configuration in other examples below. This configuration is obtained after launching the thread for key generation, so it has 2 threads, the active thread is thread 2, and no event has been executed, hence it is $\mathbb{C}_1 \stackrel{\text{def}}{=} [th_1, th_2], globalstore_1, 2, \mathbb{MI}_1, events_1$ with $events_1 = [\,]$. The second thread uses the module μ_{keygen} given in Example 7.1. After evaluating $\mu_{\mathsf{keygen}}.init$ (), we obtain $th_2 \stackrel{\text{def}}{=} \langle env_2, pe_2, stack_2, store_2\rangle$, where

$$\begin{aligned}
&env_{\mathsf{prim}} \text{ is the environment after evaluating } program(\mu_{\mathsf{prim}})\,,\\
&env_2 \stackrel{\text{def}}{=} env_{\mathsf{prim}} \oplus \{\mu_{\mathsf{keygen}}.init \mapsto \mathbf{tagfunction}^{\mathsf{keygen},\tau_1}[env_{\mathsf{prim}} \cup \{token \mapsto l_1\},\\
&\qquad\qquad () \to \text{(lines 2 to 14 of Example 7.1)}]\}\,,\\
&pe_2 \stackrel{\text{def}}{=} \mathbf{tagfunction}^{\text{Okeygen},\tau_2}[env_2 \oplus \{token \mapsto l_2\},\\
&\qquad\qquad () \to \text{(lines 5 to 13 of Example 7.1)}]\ ()\,,\\
&stack_2 \stackrel{\text{def}}{=} [(env_2, pkg := [\cdot]); (env_2, [\cdot]; \mathbf{schedule}(1)); (env_2, \mathbf{let}\ \_ = [\cdot];;)]\,,\\
&store_2 \stackrel{\text{def}}{=} \{l_1 \mapsto \mathbf{Invalid}, l_2 \mapsto \mathbf{Callable}\}\,.
\end{aligned}$$

Thread 2 first initializes the module μ_{prim}, which creates the environment env_{prim}. Next, it initializes the module μ_{keygen}: it creates the store location l_1 for the token of $\mu_{\mathsf{keygen}}.init$ and defines $\mu_{\mathsf{keygen}}.init$, which leads to the environment env_2. Then it goes into evaluation contexts to evaluate $\mu_{\mathsf{keygen}}.init$ (), which leads to the stack $stack_2$. The evaluation of $\mu_{\mathsf{keygen}}.init$ () sets the token of $\mu_{\mathsf{keygen}}.init$, in location l_1, to **Invalid**, creates the location l_2 initialized to **Callable** for the token of oracle Okeygen, and replaces $\mu_{\mathsf{keygen}}.init$ () with the corresponding closure, which leads to the current expression pe_2.

The code executed until configuration $\mathbb{C}_1$ does not alter the global store, so $globalstore_1 = globalstore_0$. The execution of the **addthread** expression removes $(\mu_{\mathsf{keygen}}, \mathsf{Once})$ from the set of available modules, since it can be used only once. Hence $\mathbb{MI}_1 = \emptyset$.

8 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. Using our compiler, we generate modules $\mathbb{M}_{\mathsf{g}}$ that correspond to the 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 $\mathit{program}_0$ that uses the generated modules in $\mathbb{M}_{\mathsf{g}}$. As explained in Section 2, when CryptoVerif shows that Q_0 satisfies a certain security property, it shows that for any CryptoVerif adversary Q_{adv}, the probability that $Q_0 \mid Q_{\mathsf{adv}}$ 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 $\mathit{program}_0$ breaks the security property is bounded by the same bound. To prove this property, we build from the OCaml adversary $\mathit{program}_0$ a CryptoVerif adversary $Q_{\mathsf{adv}}(Q_0, \mathit{program}_0)$ that simulates $\mathit{program}_0$. We prove that $Q_{\mathsf{adv}}(Q_0, \mathit{program}_0) \mid Q_0$ and $\mathit{program}_0$ using $\mathbb{M}_{\mathsf{g}}$ 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 8.1, we state our assumptions on the cryptographic primitives, and show that the primitives behave correctly independently of the rest of the program. In Section 8.2, we prove that the OCaml translation of a CryptoVerif oracle behaves like the oracle. In Section 8.3, we define the CryptoVerif adversary that simulates the OCaml adversary $\mathit{program}_0$. Finally, in Section 8.4, 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.

8.1 Correctness of Cryptographic Primitives

Let us first formalize the assumptions we make about the implementation of cryptographic primitives. Let $\mathit{program}_{\mathsf{prim}} \overset{\mathrm{def}}{=} \mathit{program}(\mu_{\mathsf{prim}})$ be the program of the module that defines the primitives and $\mathit{interface}_{\mathsf{prim}} \overset{\mathrm{def}}{=} \mathit{interface}(\mu_{\mathsf{prim}})$ be its interface. The interface $\mathit{interface}_{\mathsf{prim}}$ consists of the function $\mathbf{random}_\ell$, 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 4.3. (The functions $\mathbb{G}_{\mathsf{ser}}(T)$ and $\mathbb{G}_{\mathsf{deser}}(T)$ are either both present or both absent in $\mathit{interface}_{\mathsf{prim}}$.) We rely on the following assumptions.

Assumption 8.1 *There are no* **schedule***,* **addthread***,* **return***, nor* **event** *operations and no global store locations in* $\mathit{program}_{\mathsf{prim}}$*.*

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 (Thread) reduction rule $th \longrightarrow_p th'$, so we can reduce it by considering as configuration only a thread th. We denote by $\mathbb{TT}$ traces over threads.

Let $th^s_0 \overset{\text{def}}{=} \langle \emptyset, \mathit{program}_{\mathsf{prim}};;, [\,], \emptyset\rangle$ be a thread configuration that evaluates only the implementation of the cryptographic primitives module.

Assumption 8.2 *There exists a unique complete thread trace* $\mathbb{TT}$ *beginning at the configuration* th^s_0 *and there exists* env_{prim} *such that the last configuration of the trace* $\mathbb{TT}$ *is:*

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

This assumption means that there are no uncaught 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 $\mathit{interface}_{\mathsf{prim}} \subseteq \mathit{Dom}(env_{\mathsf{prim}})$.

Assumption 8.3 *For each CryptoVerif type* T*, OCaml values of the corresponding type* $\mathbb{G}_{\mathsf{T}}(T)$ *do not contain closures nor store or global store locations.*

This assumption formalizes that data passed to or received from generated code is immutable, as mentioned in Assumption A6: such data does not contain locations.

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 $\mathbb{G}_{\mathsf{T}}(T)$ satisfying the predicate function $\mathbb{G}_{\mathsf{pred}}(T)$. Furthermore, the OCaml value **true** and the CryptoVerif value true are such that $\mathbb{G}_{\mathsf{val}\mathit{bool}}(\text{true}) = \textbf{true}$, and the same goes for false. We extend this function to events by $\mathbb{G}_{\mathsf{ev}}(ev(a_1, \dots, a_j)) = ev(\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_j}(a_j))$ if ev 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 A2: the implementation of the cryptographic primitives in $\mathit{interface}_{\mathsf{prim}}$ emulates the corresponding behavior of CryptoVerif, as explained below.

Assumption 8.4 (Correct primitives)

1. *For each CryptoVerif function* f *of type* $T_1 \times \cdots \times T_n \rightarrow T$*, for each CryptoVerif values* $a_1, \dots, a_n$ *of types* $T_1, \dots, T_n$*, there exist env and store such that*

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

2. *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*

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

3. *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 $\mathbb{G}_{\mathsf{T}}(T)$, there exist env and store such that*

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

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

4. *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 $ser(T, a)$, such that there exist env and store such that*

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

and there exist env and store such that

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

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

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbf{random}_\ell)\ v, [\,], \emptyset \rangle \rightarrow^*_{\sum_{j \in S} almostunif(\{1,\dots,|v|\},j)} \langle env, a, [\,], store \rangle$$

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

Item 1 states that the implementation $\mathbb{G}_{\mathsf{f}}(f)$ of the cryptographic primitive f emulates f: it returns a result that matches the result of f via the mapping $\mathbb{G}_{\mathsf{val}T}$ from CryptoVerif values to OCaml values. In particular, $\mathbb{G}_{\mathsf{f}}(f)$ does not raise exceptions when its arguments correspond to CryptoVerif values of the expected type. Since at the CryptoVerif level, f can be any function that satisfies the assumptions given in the CryptoVerif specification, Item 1 just means that the implementation of f satisfies the assumptions given in the CryptoVerif specification, as mentioned in Assumption A2. Item 2 means that the function $\mathbb{G}_{\mathsf{random}}(T)$ returns a uniformly distributed random element of T. Item 3 means that $\mathbb{G}_{\mathsf{pred}}(T)$ returns **true** when its argument corresponds to an element of type T, and **false** otherwise. Item 4 specifies the correctness of the serialization and deserialization functions, using an auxiliary function *ser* such that $ser(T, a)$ is the serialized representation of the CryptoVerif value a, of type T. Finally, Item 5 guarantees that $\mathbf{random}_\ell$ is programmed correctly: $\mathbf{random}_\ell\ v$ returns a random element of the list v, such that the probability of returning the j-th element of v is $\mathit{almostunif}(\{1, \dots, |v|\}, j)$. 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 contrast to the conference version [10], in this paper, we allow the cryptographic primitives to use the store for their internal computations (which often happens in practice); the store created by the primitives appears on the right-hand side of reductions in Assumption 8.4. However, we still assume that the cryptographic primitives are pure functions: their usage of the store should not have any visible side effect, so the primitives cannot communicate across calls or communicate data to the adversary or to the rest of the code using the store. This assumption is modeled in Assumption 8.4 by considering that the primitives are initially called in an empty store. Hence, they cannot access pre-existing locations (there are none), and since their return value does not contain locations, the store at the end of the call will be unreachable. We show below, in Proposition 8.5, that when the primitives are called with a non-empty initial store, the primitives still execute in the same way as with an empty initial store: the only difference is that the unmodified initial store is added to the current store. Therefore, the primitives still do not access the initial store and the part of the store created during the execution of the primitive becomes unreachable when the primitive returns.

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 8.4 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 8.4.

Proposition 8.5 (Correct behavior of the primitives) *Let us consider a thread* $\mathit{th} \overset{\text{def}}{=} \langle \mathit{env}, \mathit{env}_{\mathsf{prim}}(s)\ v, \mathit{stack}, \mathit{store}\rangle$.

- *If* $s = \mathbb{G}_{\mathsf{f}}(f)$, f *is a CryptoVerif function of type* $T_1 \times \cdots \times T_n \to T$, *and* $v = (\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_n}(a_n))$ *for some CryptoVerif values* $a_1, \dots, a_n$ *of types* $T_1, \dots, T_n$, *then there exist* env' *and* store' *such that*
$$\mathit{th} \to^* \langle \mathit{env}', \mathbb{G}_{\mathsf{val}T}(f(a_1, \dots, a_n)), \mathit{stack}, \mathit{store}'\rangle\,.$$
- *If* $s = \mathbb{G}_{\mathsf{random}}(T)$ *and* $v = ()$, *then for each CryptoVerif value* $a \in T$, *there exist* env' *and* store' *such that*
$$\mathit{th} \to^*_{1/|T|} \langle \mathit{env}', \mathbb{G}_{\mathsf{val}T}(a), \mathit{stack}, \mathit{store}'\rangle\,.$$
- *If* $s = \mathbb{G}_{\mathsf{pred}}(T)$, *then there exist* env' *and* store' *such that*
$$\mathit{th} \to^* \langle \mathit{env}', v', \mathit{stack}, \mathit{store}'\rangle$$
where $v' = \mathbf{true}$ *when* $\mathbb{G}^{-1}_{\mathsf{val}T}(v)$ *exists, and* $v' = \mathbf{false}$ *otherwise.*

- *If* $s = \mathbb{G}_{\mathsf{ser}}(T)$ *and* $v = \mathbb{G}_{\mathsf{val}T}(a)$*, then there exist* env' *and* $store'$ *such that*
$$th \to^* \langle env', ser(T, a), stack, store' \rangle\,.$$

- *If* $s = \mathbb{G}_{\mathsf{deser}}(T)$ *and* $v = ser(T, a)$*, then there exist* env' *and* $store'$ *such that*
$$th \to^* \langle env', \mathbb{G}_{\mathsf{val}T}(a), stack, store' \rangle\,.$$

- *If* $s = \mathbf{random}_\ell$ *and* v *is a non-empty list, then for each* $a \in v$*, there exist* env' *and* $store'$ *such that*
$$th \to^*_{\sum_{j \in S} almostunif(\{1,\dots,|v|\},j)} \langle env', a, stack, store' \rangle$$
where $S \overset{\text{def}}{=} \{1 \leq j \leq |v| \mid nth(v, j) = a\}$.

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

8.2 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 $fv(M)$, $fv(P)$, $fv(Q)$ be the sets of free variables of the CryptoVerif term M and processes P and Q, respectively. These sets are defined as usual, except that each variable comes with its indices: for example, the free variables of the term $x[\widetilde{i}]$ are $fv(x[\widetilde{i}]) \overset{\text{def}}{=} \{x[\widetilde{i}]\}$. We extend this definition to terms and processes in which the replication indices $\widetilde{i}$ have been instantiated to bitstrings: for example, $fv(x[\widetilde{a}]) = \{x[\widetilde{a}]\}$. We extend this definition to sets of processes by $fv(\mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}} fv(Q)$ and to stacks by $fv(\mathcal{S}) = \bigcup_{((x_1[\widetilde{a}],\dots,x_k[\widetilde{a}]),P_1,P_2) \in \mathcal{S}} fv(P_1) \setminus \{x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]\} \cup fv(P_2)$.

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 8.6 (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*
$$\mathbb{G}_{\mathsf{tblel}}(Tbl, (b_1, \dots, b_l)) \overset{\text{def}}{=} (ser(T_1, \mathbb{G}_{\mathsf{val}T_1}(b_1)), \dots, ser(T_l, \mathbb{G}_{\mathsf{val}T_l}(b_l)))\,.$$

Let $t = [a_1; \dots; a_k]$ *be the contents of the table* Tbl*: each* a_i *is an element of the table. Let us denote*
$$\mathbb{G}_{\mathsf{tbl}}(Tbl, t) \overset{\text{def}}{=} [\mathbb{G}_{\mathsf{tblel}}(Tbl, a_1); \dots; \mathbb{G}_{\mathsf{tblel}}(Tbl, a_k)]$$
the OCaml list corresponding to t.

Definition 8.7 (Minimal environment and global store)

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

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

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 8.8 (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 $E \cdot M \Downarrow a$, 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 $store' \supseteq store$.

In this lemma, we consider an OCaml thread that evaluates the translation $\mathbb{G}_{\mathsf{M}}(M)$ of the CryptoVerif term M. We assume that its environment contains the cryptographic primitives and the minimal environment for M, as defined in Definition 8.7. We also assume that, in CryptoVerif, M evaluates to a, and we show that correspondingly, in OCaml, $\mathbb{G}_{\mathsf{M}}(M)$ evaluates to $\mathbb{G}_{\mathsf{val}T}(a)$. The final store is an extension of the initial one, since primitives may create store locations internally. We prove this result by induction on the syntax of terms and by using Proposition 8.5 for the evaluation of cryptographic primitives.

Let us now introduce some notations that allow us to designate the various parts of OCaml semantic configurations.

Definition 8.9 (Helper functions) *For an OCaml configuration*

$$\mathbb{C} = [th_1, \dots, th_n], globalstore, tj', \mathbb{MI}, events$$

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

$$\mathbb{C}_{pe}(\mathbb{C}) \stackrel{\text{def}}{=} pe_{tj'}\,, \qquad \mathbb{C}_{th}(\mathbb{C}) \stackrel{\text{def}}{=} th_{tj'}\,,$$

$$\mathbb{C}_{globalstore}(\mathbb{C}) \stackrel{\text{def}}{=} globalstore\,, \qquad \mathbb{C}_{events}(\mathbb{C}) \stackrel{\text{def}}{=} events\,.$$

We also define

$$\begin{aligned} &\mathbb{C}[\mathsf{th} \mapsto th', \mathsf{globalstore} \mapsto globalstore', \mathsf{MI} \mapsto \mathbb{MI}', \mathsf{events} \mapsto events'] \stackrel{\text{def}}{=} \\ &\qquad [th_1, \dots, th_{tj'-1}, th', th_{tj'+1}, \dots, th_n], globalstore', tj', \mathbb{MI}', events'\,. \end{aligned}$$

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

The notation $\mathbb{C}_{pe}(\mathbb{C})$ denotes the current program or expression of $\mathbb{C}$, $\mathbb{C}_{th}(\mathbb{C})$ denotes its current thread, $\mathbb{C}_{globalstore}(\mathbb{C})$ its global store, and $\mathbb{C}_{events}(\mathbb{C})$ its list of events. The notation $\mathbb{C}[\mathsf{th} \mapsto th', \mathsf{globalstore} \mapsto globalstore', \mathsf{MI} \mapsto \mathbb{MI}', \mathsf{events} \mapsto events']$ allows us to modify some elements of the configuration $\mathbb{C}$.

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 7 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 8.10 (Inner reduction) *Let $\mathcal{C}$ be a CryptoVerif configuration. Suppose that the program part P of $\mathcal{C}$ is not in a return, end, call, or loop form. Suppose that we have n possible reductions beginning at this configuration:*

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

for $i \leq n$. Let $\mathbb{C}$ be an OCaml configuration such that

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

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

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

The proof of this lemma can be found in Appendix D. This lemma is proved by cases on the process P. We use Lemma 8.8 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. We use sets of traces on the OCaml side, because the OCaml implementation of primitives may make internal random choices, leading to several traces for the same arguments and the same result, which all correspond to the same CryptoVerif trace.

```
$Q_{\mathsf{adv}}(Q_0, \mathit{program}_0) = Q_{\mathsf{start}}(Q_0, \mathit{program}_0) \mid Q_{\mathsf{c}}(Q_0, \mathit{program}_0)$
$Q_{\mathsf{start}}(Q_0, \mathit{program}_0) = \mathrm{O}_{\mathsf{start}}() :=$
      $s_0 : T_{\mathbb{CS}} \leftarrow s_0(Q_0, \mathit{program}_0)$;
      let $r : T_{\mathbb{CS}} =$ loop $\mathrm{O}_{\mathsf{loop}}(s_0)$ in end else end
$Q_{\mathsf{c}}(Q_0, \mathit{program}_0) =$ foreach $i' \leq N_{\mathsf{rand+calls}}$ do
   $\mathrm{O}_{\mathsf{loop}}[i'](s : T_{\mathbb{CS}}) :=$
      let $(s' : T_{\mathbb{CS}}, o : T_{\mathsf{o}}, i : \mathit{bitstring}, \mathit{args} : \mathit{bitstring}) = \mathrm{simulate}_{\mathsf{ML}}(s)$ in
      if $o = \mathrm{o}_{\mathsf{S}}$ then
         return$s'$, stop
      else if $o = \mathrm{o}_1$ then
         let $(a_{1,1} : T_{1,1}, \dots, a_{1,m_1} : T_{1,m_1}) = \mathit{args}$ in
         let $(i_{1,1} : [1, N_{1,1}], \dots, i_{1,n_1} : [1, N_{1,n_1}]) = i$ in
         let $(r_{1,1} : T'_{1,1}, \dots, r_{1,m'_1} : T'_{1,m'_1}) = O_1[i_{1,1}, \dots, i_{1,n_1}](a_{1,1}, \dots, a_{1,m_1})$ in
            return$(\mathrm{simulate}_{\mathsf{ret}O_1}(s', (r_{1,1}, \dots, r_{1,m'_1})), \mathrm{continue})$
         else return$(\mathrm{simulate}_{\mathsf{end}O_1}(s'), \mathrm{continue})$
      elseif $o = \mathrm{o}_2$ then
         $\vdots$
      elseif $o = \mathrm{o}_{\mathsf{R}}$ then
         $b_{\mathsf{R}} \xleftarrow{R} \mathit{bool}$;
         return$(\mathrm{simulate}_{\mathsf{R}}(s', b_{\mathsf{R}}), \mathrm{continue})$
```

Figure 18: The program $Q_{\mathsf{adv}}(Q_0, \mathit{program}_0)$

8.3 Simulation of the OCaml Adversary

In this section, we show how to simulate in CryptoVerif any OCaml program $\mathit{program}_0$ that corresponds to an adversary interacting with the protocol implementation generated from the CryptoVerif process Q_0. Basically, we run the OCaml program $\mathit{program}_0$ inside the CryptoVerif function $\mathrm{simulate}_{\mathsf{ML}}$ (which is possible since these functions can represent any deterministic Turing machine). When $\mathit{program}_0$ needs to call an oracle of Q_0, the function returns and the call is made by CryptoVerif. When $\mathit{program}_0$ needs to generate a random number, this generation is performed by CryptoVerif.

In more detail, from the OCaml program $\mathit{program}_0$, we define a CryptoVerif adversary $Q_{\mathsf{adv}}(Q_0, \mathit{program}_0)$ given in Figure 18. 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 $\mathcal{C}_0(Q_0, \mathit{program}_0) = \mathcal{C}_{\mathsf{i}}(Q_0 \mid Q_{\mathsf{adv}}(Q_0, \mathit{program}_0))$. Informally, in Figure 18, the state s is a bitstring representation of the current OCaml semantic configuration. The oracle $\mathrm{O}_{\mathsf{start}}$ iterates the oracle $\mathrm{O}_{\mathsf{loop}}$ with initial state $s_0 = s_0(Q_0, \mathit{program}_0)$, which is a bitstring representation of the initial OCaml configuration in which $\mathit{program}_0$ is executed. Inside $\mathrm{O}_{\mathsf{loop}}(s)$, the function $\mathrm{simulate}_{\mathsf{ML}}(s)$

basically runs the OCaml program from state s, following the OCaml semantics with the following exceptions:

- When the OCaml program calls an oracle, $\mathsf{simulate}_{\mathsf{ML}}$ returns $(s', o, i, args)$ where s' is a bitstring representation of the new OCaml semantic configuration, o is a constant among $\mathrm{o}_1, \mathrm{o}_2, \dots$ that encodes which oracle is called, i is the tuple of indices with which the oracle is called, and $args$ is the tuple of arguments of the oracle. In this case, $\mathrm{O}_{\mathsf{loop}}$ calls the corresponding oracle O (lines 10–17). If the oracle call succeeds, it uses the function $\mathrm{simulate}_{\mathsf{ret}O}$, which replaces the oracle call with the result $r_{i,1}, \dots, r_{i,m'_i}$ of the oracle in the OCaml configuration s' (see Definition 8.13 below). If the oracle call fails, the call raises the exception **Match_failure** in OCaml; the function $\mathrm{simulate}_{\mathsf{end}O}$ then replaces the oracle call with this exception in the OCaml configuration s' (see Definition 8.13 below). The execution of the program then continues with the new configuration in the next iteration.

- When the OCaml program chooses a random bit, $\mathrm{simulate}_{\mathsf{ML}}$ returns $(s', \mathrm{o}_{\mathsf{R}}, (), ())$ where s' is again a bitstring representation of the current OCaml semantic configuration. In this case, $\mathrm{O}_{\mathsf{loop}}$ chooses a random bit (lines 18–20) and uses the function $\mathrm{simulate}_{\mathsf{R}}$ (see Definition 8.14 below) to integrate that random bit into the OCaml configuration s'. The execution of the program continues with the new configuration in the next iteration.

- When the OCaml program terminates, $\mathrm{simulate}_{\mathsf{ML}}$ returns $(s', \mathrm{o}_{\mathsf{S}}, (), ())$, and the CryptoVerif adversary also terminates. (The second element returned by $\mathrm{O}_{\mathsf{loop}}$ is stop, which stops the iteration.)

The rest of this section is devoted to the formal definition of all elements used in Figure 18.

We assume that the OCaml program $program_0$ runs in bounded time, so makes a bounded number of oracle calls. By Assumption 4.15, 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 $\mathbf{tagfunction}^{O,\tau}[env, pm]$ corresponding to oracle O. When a role role is under replication, we let N_{role} be the maximum number of executions of $\mathbf{addthread}(program)$ 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 8.11 *The number of calls to the closure with tag O, τ in a trace $\mathbb{CT}$, denoted $N_{\mathsf{calls}}(O, \tau, \mathbb{CT})$, is the number of configurations $\mathbb{C}$ such that $\mathbb{C}_{pe}(\mathbb{C}) =$ $\mathbf{tagfunction}^{O,\tau}[env, pm]\ v$ in $\mathbb{CT}$ excluding its last configuration.*

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

The number of random number generations in a trace $\mathbb{CT}$*, denoted* $N_{\mathsf{rand}}(\mathbb{CT})$*, is the number of configurations* $\mathbb{C}$ *such that* $\mathbb{C}_{pe}(\mathbb{C}) = \mathbf{random}\ ()$ *in* $\mathbb{CT}$ *excluding its last configuration.*

We define

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

where $\mathbb{CT}$ *ranges over traces that begin with the configuration* $\mathbb{C}_0(Q_0, \mathit{program}_0)$*.*

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 $\mathit{program}_0$ of the number of instantiations of role divided by the number of calls to O made 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.

In Figure 18, we use a let construct with pattern matching, which can be defined as follows. We define the function $\mathrm{tuple}_{T_1,\ldots,T_j} : T_1 \times \cdots \times T_j \rightarrow \mathit{bitstring}$ 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 $\pi_{k,T_1,\ldots,T_j} : \mathit{bitstring} \rightarrow T_k$ with $k \leq j$ (which may return any value when their argument is not a tuple with j elements). The construct $\mathsf{let}\ (x_1 : T_1, \ldots, x_j : T_j) = M\ \mathsf{in}\ P$ is an abbreviation for:

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

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

Let $O_1, \dots, O_n$ be the oracle names in Q_0. We define n constants $\mathrm{o}_1, \dots, \mathrm{o}_n$ which are used to designate the oracles $O_1, \dots, O_n$ respectively, $\mathsf{o_R}$ which corresponds to a random choice, and $\mathsf{o_S}$ which corresponds to the end of the OCaml program. We define the CryptoVerif type $T_{\mathsf{o}} \overset{\text{def}}{=} \{\mathsf{o_R}, \mathsf{o_S}, \mathrm{o}_1, \dots, \mathrm{o}_n\}$, which contains all these bitstring constants.

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

$$\mathbb{CS} = \underbrace{([\mathit{th}_1, \dots, \mathit{th}_n], \mathit{globalstore}, i)}_{\mathbb{C}}, \mathbb{RI}, \mathbb{I}\,.$$

The function repr is injective. We denote its inverse by repr^{-1}. We also define a CryptoVerif type $T_{\mathbb{CS}}$ that consists of all bitstrings in the image of repr, that is, all bitstrings that correspond to simulator configurations $\mathbb{CS}$. We also use the notations of Definition 8.9 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 $\mathbb{RI}$ and $\mathbb{I}$ represent the sets of callable roles and oracles, by storing the smallest index a that is not used yet.

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

- of the form $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$. Intuitively, this element represents all roles $\mathsf{role}[a'', \widetilde{a'}]$ for $a'' \geq a$, which we represent by the interval $[a, +\infty[$. When $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$ is in $\mathbb{RI}$, 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 $\mathbb{RI}$ never contains simultaneously $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$ and $\mathsf{role}[\widetilde{a''}]$ for the same role and any $a, \widetilde{a'}, \widetilde{a''}$, 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 $\mathbb{I}$ represents the set of callable oracles with their replication indices. Elements of $\mathbb{I}$ 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 $\mathbb{I}$ never contains simultaneously $O\big[[a,+\infty[,\widetilde{a'}\big]$ and $O[\widetilde{a''}]$ for the same O and any $a,\widetilde{a'},\widetilde{a''}$, 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 $\mathbb{I} - O[\widetilde{a}]$ on sets of oracles.

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

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

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

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

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

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

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

Let us define the function *oraclelist*, which is similar to *oracledeflist* 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:

$$\mathit{oraclelist}(0) \stackrel{\text{def}}{=} [\,] \tag{Nil}$$

$$\mathit{oraclelist}(Q_1 \mid Q_2) \stackrel{\text{def}}{=} \mathit{oraclelist}(Q_1) \mathbin{@} \mathit{oraclelist}(Q_2) \tag{Par}$$

$$\begin{aligned}\mathit{oraclelist}(\mathsf{foreach}\ i' \leq n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} {} & \big[O_1[\_,\widetilde{i}],\dots,O_l[\_,\widetilde{i}]\big] \text{ when}\\ & \mathit{oraclelist}(Q) = \big[O_1[i',\widetilde{i}],\dots,O_l[i',\widetilde{i}]\big]\end{aligned} \tag{Repl}$$

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

$$\mathit{oraclelist}(O[\widetilde{i}](x_1[\widetilde{i}],\dots,x_k[\widetilde{i}]) := P) \stackrel{\text{def}}{=} \big[O[\widetilde{i}]\big] \tag{Oracle}$$

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 *oracledeflist*, 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 4.14, 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), $\mathit{oraclelist}(Q')$

produces the list of callable oracles in Q where we replace the replication index i' with _.

By Property 4.5, 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 $T_{i,1}, \dots, T_{i,m_i}$, and returns m'_i bitstrings of types $T'_{i,1}, \dots, T'_{i,m'_i}$. We can also define $\mathit{returnoracles}(O[\widetilde{i}]) \stackrel{\text{def}}{=} \mathit{oraclelist}(Q)$ 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 4.5, the structure of the processes Q after any return statement of a given oracle O is always the same, so the list $\mathit{oraclelist}(Q)$ will be the same for each of these Q. The function *returnoracles* can take an oracle with its replication indices partly instantiated to values: $\mathit{returnoracles}(O[\widetilde{a}]) \stackrel{\text{def}}{=} \mathit{returnoracles}(O[\widetilde{i}])\{\widetilde{a}/\widetilde{i}\}$.

Let us recall that we 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 8.12 (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,* $\mathit{oraclelist}(Q(\mathsf{role}))$*.*

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

$$\begin{aligned}\mathit{add}(\mathbb{I}, \mathbb{RI}) \stackrel{\text{def}}{=} \mathbb{I} \cup \{O[\widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathbb{RI}, O[\widetilde{a}] \in \mathit{oraclelist}(Q(\mathsf{role})[\widetilde{a}])\} \cup \\ \{O\big[[1, +\infty[, \widetilde{a}\big] \mid \mathsf{role}[\widetilde{a}] \in \mathbb{RI}, O[\_, \widetilde{a}] \in \mathit{oraclelist}(Q(\mathsf{role})[\widetilde{a}])\}\end{aligned}$$

The syntax of the language of the simulator is almost the same as the language we described in Section 5, with the addition of tagged functions introduced in Section 6. We add the functional values $\mathbf{call}(O[\widetilde{a}])$ and $\mathbf{call}(O[\_, \widetilde{a}])$ that replace our generated closures for the oracle O. The value $\mathbf{call}(O[\widetilde{a}])$ is used when O is not directly under replication; $\mathbf{call}(O[\_, \widetilde{a}])$ is used when O is directly under replication.

We present the semantics followed by our simulator in Figure 19. 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 **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 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

$$\frac{\begin{array}{l} O[\widetilde{a}] \notin \mathbb{I} \\ \text{or } O \text{ does not have } k \text{ arguments} \\ \text{or } O \text{ has } k \text{ arguments of type } T_1, \dots, T_k \text{ and } \exists i, \nexists a_i, v_i = \mathbb{G}_{\mathsf{val}T_i}(a_i) \end{array}}{\mathit{env}, \mathbf{call}(O[\widetilde{a}])\ (v_1, \dots, v_k), \mathit{stack} \longrightarrow \mathit{env}, \mathbf{raise\ Bad\_Call}, \mathit{stack}}$$

(FailedCall1)

$$\frac{\begin{array}{l} \forall a', O\big[[a', +\infty[, \widetilde{a}\big] \notin \mathbb{I} \\ \text{or } O \text{ does not have } k \text{ arguments} \\ \text{or } O \text{ has } k \text{ arguments of type } T_1, \dots, T_k \text{ and } \exists i, \nexists a_i, v_i = \mathbb{G}_{\mathsf{val}T_i}(a_i) \end{array}}{\mathit{env}, \mathbf{call}(O[\_, \widetilde{a}])\ (v_1, \dots, v_k), \mathit{stack} \longrightarrow \mathit{env}, \mathbf{raise\ Bad\_Call}, \mathit{stack}}$$

(FailedCall2)

$$\frac{\begin{array}{c} \mathbb{C} \to \mathbb{C}' \text{ using the rules of Figures 7–14, (FailedCall1), and (FailedCall2)} \\ \text{but not (Random) and (Toplevel add thread)} \end{array}}{\mathbb{C}, \mathbb{RI}, \mathbb{I} \to \mathbb{C}', \mathbb{RI}, \mathbb{I}}$$

(Simulator toplevel)

$$\frac{\begin{array}{l} \mathit{program}^a = \mathit{program}_{\mathsf{prim}};; \mathit{program}(\mu_{\mathsf{role}_1});;\ \dots;; \mathit{program}(\mu_{\mathsf{role}_l});; \mathit{program}' \\ \mathit{program}' \text{ does not contain } \mathit{program}(\mu_{\mathsf{prim}}) \text{ nor any } \mathit{program}(\mu) \text{ for } \mu \in \mathbb{M}_{\mathsf{g}} \\ \{\mu_{\mathsf{role}_1}, \dots, \mu_{\mathsf{role}_l}\} \subseteq \mathbb{M}_{\mathsf{g}} \\ \widetilde{a_1} = \mathit{smallest}(\mathbb{RI}, \mathsf{role}_1), \dots, \widetilde{a_l} = \mathit{smallest}(\mathbb{RI}, \mathsf{role}_l) \\ \mathbb{RI}'' = \{\mathsf{role}_1[\widetilde{a_1}], \dots, \mathsf{role}_l[\widetilde{a_l}]\} \qquad \mathbb{RI}' = \mathbb{RI} - \mathbb{RI}'' \qquad \mathbb{I}' = \mathit{add}(\mathbb{I}, \mathbb{RI}'') \\ \mathit{program}^b = \mathit{program}_{\mathsf{prim}};; \mathit{program}'(\mathsf{role}_1[\widetilde{a_1}]);;\ \dots;; \mathit{program}'(\mathsf{role}_l[\widetilde{a_l}]);; \\ \qquad\qquad \mathit{program}' \\ \text{or} \\ \mathit{program}^a \text{ does not contain } \mathit{program}(\mu_{\mathsf{prim}}) \text{ nor any } \mathit{program}(\mu) \text{ for } \mu \in \mathbb{M}_{\mathsf{g}} \\ \mathbb{RI}'' = \emptyset \qquad \mathbb{RI}' = \mathbb{RI} \qquad \mathbb{I}' = \mathbb{I} \qquad \mathit{program}^b = \mathit{program}^a \end{array}}{\begin{array}{l} [\mathit{th}_1, \dots, \mathit{th}_{tj-1}, \langle \mathit{env}, \mathbf{addthread}(\mathit{program}^a), \mathit{stack}, \mathit{store}\rangle, \mathit{th}_{tj+1}, \dots, \mathit{th}_n], \\ \qquad \mathit{globalstore}, \mathit{tj}, \mathbb{RI}, \mathbb{I} \longrightarrow \\ [\mathit{th}_1, \dots, \mathit{th}_{tj-1}, \langle \mathit{env}, (), \mathit{stack}, \mathit{store}\rangle, \mathit{th}_{tj+1}, \dots, \mathit{th}_n, \langle \emptyset, \mathit{program}^b, [\,], \emptyset\rangle], \\ \qquad \mathit{globalstore}, \mathit{tj}, \mathbb{RI}', \mathbb{I}' \end{array}}$$

(Simulator add thread)

Figure 19: Semantics followed by the simulator

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 $\mathit{program}(\mu_{\mathsf{role}})$ into references to the corresponding role $\mathit{program}'(\mathsf{role}[\widetilde{a}])$ where $\widetilde{a}$ are the replication indices we chose for this particular reference and

```
$\mathit{program}'(\mathsf{role}[\widetilde{a}]) \overset{\text{def}}{=}$
        $\textbf{let } \mu_{\mathsf{role}}.\mathit{init} = \textbf{let } \mathit{token} = \textbf{ref Callable in tagfunction}^{\mathsf{role}}\ \mathit{pm}'_{\mathsf{role}[\widetilde{a}]}$
    where $\mathit{pm}'_{\mathsf{role}[\widetilde{a}]} \overset{\text{def}}{=} () \rightarrow$
        $\textbf{if } (!\mathit{token} = \textbf{Callable})$
        $\textbf{then } (\mathit{token} := \textbf{Invalid}; (\textbf{call}(O_1[\widetilde{a_1}]), \ldots, \textbf{call}(O_k[\widetilde{a_k}])))$
        $\textbf{else raise Bad\_Call}$
```

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

The CryptoVerif function $\mathrm{simulate}_{\mathsf{ML}} : T_{\mathbb{CS}} \rightarrow \mathit{bitstring}$ follows the simulator semantics defined in Figure 19: formally, we define $\mathrm{simulate}_{\mathsf{ML}}(\mathit{repr}(\mathbb{CS})) \overset{\text{def}}{=} \mathit{simreturn}(\mathbb{CS}')$ where $\mathbb{CS}'$ is the configuration such that either $\mathbb{CS}$ reduces into $\mathbb{CS}'$ in at most N_{steps} reductions and $\mathbb{CS}'$ does not reduce, or $\mathbb{CS}$ reduces into $\mathbb{CS}'$ in exactly N_{steps} reductions, by the semantics of Figure 19, and $\mathit{simreturn}(\mathbb{CS}')$ is defined below. (We need to bound the number of reductions to make sure that $\mathrm{simulate}_{\mathsf{ML}}$ 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 $\mathbb{C}_{pe}(\mathbb{CS}') = \textbf{call}(O[\widetilde{a}])\ (v_1, \ldots, v_l)$, 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
$$\mathit{simreturn}(\mathbb{CS}') \overset{\text{def}}{=} (\mathit{repr}(\mathbb{CS}'), o, \widetilde{a}, (\mathbb{G}^{-1}_{\mathsf{val}T_1}(v_1), \ldots, \mathbb{G}^{-1}_{\mathsf{val}T_l}(v_l)))\,.$$
- If $\mathbb{C}_{pe}(\mathbb{CS}') = \textbf{call}(O[\_, \widetilde{a}])\ (v_1, \ldots, v_l)$, 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 $\mathbb{I}$ where $\mathbb{CS}' = \mathbb{C}, \mathbb{RI}, \mathbb{I}$. We define
$$\mathit{simreturn}(\mathbb{CS}') \overset{\text{def}}{=} (\mathit{repr}(\mathbb{CS}'), o, (a', \widetilde{a}), (\mathbb{G}^{-1}_{\mathsf{val}T_1}(v_1), \ldots, \mathbb{G}^{-1}_{\mathsf{val}T_l}(v_l)))\,.$$
- If $\mathbb{C}_{pe}(\mathbb{CS}') = \textbf{random}\ ()$, we define
$$\mathit{simreturn}(\mathbb{CS}') \overset{\text{def}}{=} (\mathit{repr}(\mathbb{CS}'), \mathsf{o_R}, (), ())\,.$$
- Otherwise, we define
$$\mathit{simreturn}(\mathbb{CS}') \overset{\text{def}}{=} (\mathit{repr}(\mathbb{CS}'), \mathsf{o_S}, (), ())\,.$$

The function $\mathrm{simulate}_{\mathsf{ML}}$ can be implemented by a deterministic Turing machine (since the random choices are handled outside $\mathrm{simulate}_{\mathsf{ML}}$), so it can be used as a CryptoVerif function.

When $\mathrm{simulate}_{\mathsf{ML}}$ returns $(\mathit{repr}(\mathbb{CS}'), o, \widetilde{a}, (a_1, \dots, a_l))$, the CryptoVerif process $Q_c(Q_0, \mathit{program}_0)$ performs the corresponding oracle call $O[\widetilde{a}](a_1, \dots, a_l)$ (lines 10–17 of Figure 18). Similarly, when $\mathrm{simulate}_{\mathsf{ML}}$ returns $(\mathit{repr}(\mathbb{CS}'), \mathsf{o_R}, (), ())$, the process $Q_c(Q_0, \mathit{program}_0)$ performs a random choice (lines 18–20), and when $\mathrm{simulate}_{\mathsf{ML}}$ returns $(\mathit{repr}(\mathbb{CS}'), \mathsf{o_S}, (), ())$, the process $Q_c(Q_0, \mathit{program}_0)$ terminates (lines 8–9; the corresponding OCaml program also terminates).

The functions $\mathrm{simulate}_{\mathsf{ret}O}$ and $\mathrm{simulate}_{\mathsf{end}O}$ replace, in the simulator configuration, the **call** expression with the result returned by the oracle, and raise the **Match_failure** exception, respectively. The function $\mathrm{simulate}_{\mathsf{ret}O}$ handles the situation in which an oracle returns a result by return; the function $\mathrm{simulate}_{\mathsf{end}O}$ handles the situation in which the oracle terminates with end. Formally, these functions are defined as follows.

Definition 8.13 (Simulation of oracle return) *Let us consider a simulator configuration $\mathbb{CS} = \mathbb{C}, \mathbb{RI}, \mathbb{I}$, with*

$$\mathbb{C}_{pe}(\mathbb{CS}) = \mathbf{call}(O[\widetilde{a}])\ (v_1, \dots, v_l) \textit{ or } \mathbf{call}(O[\_\,, \widetilde{a'}])\ (v_1, \dots, v_l)\,.$$

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

We define the CryptoVerif function $\mathrm{simulate}_{\mathsf{ret}O} : T_{\mathbb{CS}} \times \mathit{bitstring} \to T_{\mathbb{CS}}$ *as follows.*

1. *If the returns in oracle O end the current role, then by Property 4.11, there is only one* return *statement in O; let Q be the oracle definition following this statement, and let*

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

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

$$\mathrm{simulate}_{\mathsf{ret}O}(\mathit{repr}(\mathbb{C}, \mathbb{RI}, \mathbb{I}), (r_1, \dots, r_n)) \stackrel{\text{def}}{=} \mathit{repr}(\mathbb{C}', \mathbb{RI} \cup \mathbb{RI}', \mathbb{I}')$$

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

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

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

   We define:

$$\mathrm{simulate}_{\mathsf{ret}O}(\mathit{repr}(\mathbb{C}, \mathbb{RI}, \mathbb{I}), (r_1, \dots, r_n)) \stackrel{\text{def}}{=} \mathit{repr}(\mathbb{C}', \mathbb{RI}, \mathbb{I}'')$$

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

3. *In all other cases (that is, $\mathbb{CS}$ is not of the form mentioned above or a is not a tuple of n bitstrings of types $T_1, \dots, T_n$),* $\mathrm{simulate}_{\mathsf{ret}O}(\mathit{repr}(\mathbb{CS}), a)$ *can take any value, since these cases are in fact not used.*

Finally, we define the CryptoVerif function $\mathrm{simulate}_{\mathsf{end}O} : T_{\mathbb{CS}} \to T_{\mathbb{CS}}$ *by:*

$$\mathrm{simulate}_{\mathsf{end}O}(\mathit{repr}(\mathbb{C}, \mathbb{RI}, \mathbb{I})) \overset{\text{def}}{=} \mathit{repr}(\mathbb{C}'', \mathbb{RI}, \mathbb{I}')$$

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

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

Definition 8.14 (Random simulation) *We define the CryptoVerif function* $\mathrm{simulate}_{\mathsf{R}} : T_{\mathbb{CS}} \times \mathit{bool} \to T_{\mathbb{CS}}$ *by*

$$\mathrm{simulate}_{\mathsf{R}}(\mathit{repr}(\mathbb{C}, \mathbb{RI}, \mathbb{I}), b) \overset{\text{def}}{=} \mathit{repr}(\mathbb{C}'(b), \mathbb{RI}, \mathbb{I})$$

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

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

$$s_0(Q_0, \mathit{program}_0) \overset{\text{def}}{=} \mathit{repr}(([\langle \emptyset, \mathit{program}_0, [\,], \emptyset \rangle], \mathit{globalstore}_0, 1), \mathbb{RI}_0, \emptyset)$$

Let us introduce notations for subprocesses of Figure 18, used in the next example and in Definition 8.32.

Definition 8.15 (Processes) *We use the following notations:*

P_{loop} *is the process from line 7 to line 20 in Figure 18.*

$Q_{\mathsf{loop}} \overset{\text{def}}{=} \mathrm{O}_{\mathsf{loop}}[i'](s : T_{\mathbb{CS}}) := P_{\mathsf{loop}}\,.$

$$\begin{aligned}P_{\mathsf{return\text{-}loop}}(\alpha) \overset{\text{def}}{=}\ & \mathsf{if}\ b_{\alpha,r}[\,]\ \mathsf{then} \\ &\qquad \mathsf{let}\ r[\,] : T_{\mathbb{CS}} = \mathsf{loop}\ O_{loop}[\alpha+1](r'_{\alpha,r}[\,])\ \mathsf{in\ end\ else\ end} \\ &\quad \mathsf{else}\ r[\,] \leftarrow r'_{\alpha,r}[\,]; \mathsf{end}\,.\end{aligned}$$

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

These notations are useful to represent the CryptoVerif configuration when CryptoVerif calls $\text{simulate}_{\mathsf{ML}}$, at line 7 of Figure 18: in iteration α of oracle $\mathrm{O}_{\mathsf{loop}}$, the current process is $P_{\mathsf{loop}}\{\alpha/i'\}$, the available $\mathrm{O}_{\mathsf{loop}}$ oracles are $Q_{\mathsf{loop}}\{a/i'\}$ for $\alpha < a \leq N_{\mathsf{rand+calls}}$, and the CryptoVerif stack is $\mathcal{S}_{\mathsf{loop}}(\alpha)$.

Example 8.16 Let us consider again the OCaml program $\mathit{program}_0$ of Example 7.3 and the process Q_0 of Example 4.10. The initial state of the simulator is then $s_0(Q_0, \mathit{program}_0) = \mathit{repr}(\mathbb{CS}_0)$ with

$$\mathbb{CS}_0 \stackrel{\text{def}}{=} ([\langle\emptyset, \mathit{program}_0, [\,], \emptyset\rangle], \mathit{globalstore}_0, 1), \mathbb{RI}_0, \mathbb{I}_0$$

where $\mathbb{RI}_0 \stackrel{\text{def}}{=} \{\mathsf{keygen}[\,]\}$, $\mathbb{I}_0 \stackrel{\text{def}}{=} \emptyset$, and $\mathit{globalstore}_0$ is defined in Example 7.3.

We execute the simulator of Figure 18 with that value of $s_0(Q_0, \mathit{program}_0)$; in this example, the oracles $O_1, O_2, \dots$ are Okeygen, OA, and OB. CryptoVerif calls oracle $\mathrm{O}_{\mathsf{start}}$, which iterates $\mathrm{O}_{\mathsf{loop}}$. It first calls $\mathrm{O}_{\mathsf{loop}}[1](s_0(Q_0, \mathit{program}_0))$, which calls $\text{simulate}_{\mathsf{ML}}(s_0(Q_0, \mathit{program}_0))$ (Figure 18, line 7). This function starts running the simulator semantics. In the execution of the **addthread** expression, $\mathsf{role}_1 = \mathsf{keygen}$, $\widetilde{a}$ is empty, $\mathbb{RI}'' = \{\mathsf{keygen}[\,]\}$, so $\mathsf{keygen}[\,]$ is removed from $\mathbb{RI}$, which becomes $\mathbb{RI}_1 = \emptyset$, and the corresponding oracle Okeygen[ ] is added to $\mathbb{I}$, which becomes $\mathbb{I}_1 = \{\text{Okeygen}[\,]\}$: this oracle can now be called. In the added thread, $\mathit{program}(\mu_{\mathsf{keygen}})$ is replaced with

```
$\mathit{program}'(\mathsf{keygen}[\,]) \stackrel{\text{def}}{=}$
    $\textbf{let } \mu_{\mathsf{keygen}}.\mathit{init} = \textbf{let } \mathit{token} = \textbf{ref Callable in tagfunction}^{\mathsf{keygen}}\ () \to$
    $\textbf{if } (!\mathit{token} = \textbf{Callable})$
    $\textbf{then } (\mathit{token} := \textbf{Invalid}; \textbf{call}(\text{Okeygen}[\,]))$
    $\textbf{else raise Bad\_Call}$
```

After the evaluation of $\mu_{\mathsf{keygen}}.\mathit{init}\ ()$, the simulator configuration is $\mathbb{CS}_1 \stackrel{\text{def}}{=} ([\mathit{th}^{\mathsf{s}}_1, \mathit{th}^{\mathsf{s}}_2], \mathit{globalstore}^{\mathsf{s}}_1, 2), \mathbb{RI}_1, \mathbb{I}_1$ where

$$\begin{aligned}
&\mathit{th}^{\mathsf{s}}_2 \stackrel{\text{def}}{=} \langle \mathit{env}^{\mathsf{s}}_2, \mathit{pe}^{\mathsf{s}}_2, \mathit{stack}^{\mathsf{s}}_2, \mathit{store}^{\mathsf{s}}_2 \rangle\,,\\
&\mathit{env}^{\mathsf{s}}_2 \stackrel{\text{def}}{=} \mathit{env}_{\mathsf{prim}} \oplus \{\mu_{\mathsf{keygen}}.\mathit{init} \mapsto\\
&\qquad\qquad \textbf{tagfunction}^{\mathsf{keygen},\tau_1}[\mathit{env}_{\mathsf{prim}} \cup \{\mathit{token} \mapsto l_1\}, \mathit{pm}'_{\mathsf{keygen}[\,]}]\}\,,\\
&\mathit{pe}^{\mathsf{s}}_2 \stackrel{\text{def}}{=} \textbf{call}(\text{Okeygen}[\,])\ ()\,,\\
&\mathit{stack}^{\mathsf{s}}_2 \stackrel{\text{def}}{=} [(\mathit{env}^{\mathsf{s}}_2, \mathit{pkg} := [\cdot]); (\mathit{env}^{\mathsf{s}}_2, [\cdot]; \textbf{schedule}(1)); (\mathit{env}^{\mathsf{s}}_2, \textbf{let}\ \_ = [\cdot];;)]\,,\\
&\mathit{store}^{\mathsf{s}}_2 \stackrel{\text{def}}{=} \{l_1 \mapsto \textbf{Invalid}\}\,.
\end{aligned}$$

The execution of this thread in the simulator is fairly similar to the one in OCaml, discussed in Example 7.3. It first initializes the module μ_{prim}, which creates the environment $\mathit{env}_{\mathsf{prim}}$. Next, it initializes the module μ_{keygen}: it creates the store location l_1 for the token of $\mu_{\mathsf{keygen}}.\mathit{init}$ and defines $\mu_{\mathsf{keygen}}.\mathit{init}$, which leads to the environment $\mathit{env}^{\mathsf{s}}_2$. Then it goes into evaluation contexts to evaluate $\mu_{\mathsf{keygen}}.\mathit{init}\ ()\ ()$, which leads to the stack $\mathit{stack}^{\mathsf{s}}_2$. The evaluation of $\mu_{\mathsf{keygen}}.\mathit{init}\ ()$

sets the token of $\mu_{\mathsf{keygen}}.\mathit{init}$, in location l_1, to **Invalid** and replaces $\mu_{\mathsf{keygen}}.\mathit{init}\ ()$ with the corresponding **call** value, which leads to the current expression pe_2^{s}. In contrast to the OCaml execution, no token is created for oracle Okeygen, so location l_2 does not appear. The global store remains unchanged: $\mathit{globalstore}_1^{\mathsf{s}} = \mathit{globalstore}_0$.

At this configuration, the function simulate$_{\mathsf{ML}}$ stops and returns to CryptoVerif to evaluate the call to oracle Okeygen[ ]. The CryptoVerif configuration at this point is $\mathcal{C}_1 = E_1, P_{\mathsf{loop}}\{1/i'\}, \mathcal{T}_1, \mathcal{Q}_1, \mathcal{S}_{\mathsf{loop}}(1), \mathcal{E}_1$ where $\mathcal{T}_1 = \emptyset$ since this example does not use tables; $\mathcal{Q}_1 = \{Q_{\mathsf{loop}}\{a/i'\} \mid 1 < a \leq N_{\mathsf{rand+calls}}\} \cup \{\text{Okeygen}[\,]() := \ldots (\text{as in } Q_0)\}$ since O$_{\mathsf{loop}}$ has been called with index 1 and is still available for larger indices, Okeygen has not been called yet, so it is available, OA and OB will become available only after the return from Okeygen; $\mathcal{E}_1 = [\,]$ since no events have been executed so far.

8.4 Correspondence between the CryptoVerif and OCaml Systems

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

Similarly to the definition of $\Pr[\mathcal{C} :^{(\mathsf{CV})} D]$ in Section 4.2, we define the probability of breaking the security property associated to D in OCaml: $\Pr[\mathbb{C} :^{(\mathsf{ML})} D]$ is the probability of the set of complete OCaml traces starting at $\mathbb{C}$ and such that the list of events *events* in their last configuration satisfies $D(\mathbb{G}_{\mathsf{ev}}^{-1}(\mathit{events})) =$ true. Our goal is to prove that, for all protocols Q_0, OCaml adversaries $\mathit{program}_0$, and distinguishers D, we have

$$\Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) :^{(\mathsf{CV})} D] = \Pr[\mathbb{C}_0(Q_0, \mathit{program}_0) :^{(\mathsf{ML})} D]\,.$$

As explained in Section 2, this result shows the correctness of our compiler.

To that order, we first introduce an intermediate semantics for CryptoVerif that decomposes the evaluation of the function simulate$_{\mathsf{ML}}$ into several small steps. We easily relate this semantics to the semantics of CryptoVerif. Next, in Section 8.4.2, 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 8.4.3, we use these results to prove our main theorem.

8.4.1 Intermediate Semantics

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

Definition 8.17 *Let us define the reduction relation $\rightsquigarrow$ such that:*

$$\frac{E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \to_p \mathcal{C}' \quad P \textit{ is not of the form } x[a] \leftarrow \mathrm{simulate}_{\mathsf{ML}}(s[a]); P' \textit{ for any } x, a, P'}{E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \rightsquigarrow_p \mathcal{C}'} \quad \text{(CryptoVerif)}$$

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

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

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

When encountering a configuration $\mathcal{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$ such that P is of the form $x[a] \leftarrow \mathrm{simulate}_{\mathsf{ML}}(s[a]); P'$ and $E(s[a]) = \mathit{repr}(\mathbb{CS})$, we reduce $\mathcal{C}$ into an extended configuration $\mathcal{C}, N_{\mathsf{steps}}, \mathbb{CS}$ by (Enter Simulator). We reduce $\mathbb{CS}$ 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 $\to$, to prove that all events have the same probability in these two semantics. These results are proved in Appendix E.

Lemma 8.18 *Let $\mathcal{C}$ be a CryptoVerif configuration.*

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

We denote by $\Pr[\mathcal{C}^{\mathsf{cs}} :^{(\rightsquigarrow)} D]$ the probability of the set of complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathcal{C}^{\mathsf{cs}}$ 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 8.19 $\Pr[\mathcal{C} :^{(\rightsquigarrow)} D] = \Pr[\mathcal{C} :^{(\mathsf{CV})} D]$.

Proof sketch We partition the set of complete traces using $\to$ and beginning at $\mathcal{C}$ into two: the traces $\mathcal{CTS}_{\text{true}}$ that verify D and the traces $\mathcal{CTS}_{\text{false}}$ that do not verify D. By using Lemma 8.18, we convert these sets into two sets of traces using $\rightsquigarrow$, $\mathcal{CTS}^{\mathsf{cs}}_{\text{true}}$ and $\mathcal{CTS}^{\mathsf{cs}}_{\text{false}}$. Traces in $\mathcal{CTS}^{\mathsf{cs}}_{\text{true}}$ verify D, and traces in $\mathcal{CTS}^{\mathsf{cs}}_{\text{false}}$ do not verify D, and $\Pr[\mathcal{CTS}_b] = \Pr[\mathcal{CTS}^{\mathsf{cs}}_b]$ for $b \in \{\text{true}, \text{false}\}$. These two sets form a partition of the set of complete traces using $\rightsquigarrow$. $\square$

8.4.2 Relation between the Intermediate Semantics and the OCaml Semantics

In this section, we first define a relation between the intermediate semantics and the OCaml semantics. Then, we prove that this relation holds, which implies that $\Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) :^{(\rightsquigarrow)} D] = \Pr[\mathbb{C}_0(Q_0, \mathit{program}_0) :^{(\mathsf{ML})} D]$.

Since the definition of the relation is fairly complex, we proceed in several steps. We first define an invariant on the simulator configurations, which intuitively means that each oracle is in a single status (possibly available in the future, available for immediate calls, already called) and that oracles available in different threads are distinct. To formalize this invariant, we first define the sets of oracles represented by $\mathbb{I}$ and $\mathbb{RI}$.

Definition 8.20 (Concretization of $\mathbb{I}$ and $\mathbb{RI}$) *Let us define the sets of oracles $\mathcal{O}^\infty(\mathbb{I})$ and $\mathcal{O}^\infty(\mathbb{RI})$ represented by $\mathbb{I}$ and $\mathbb{RI}$ respectively:*

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

The definition of $\mathcal{O}^\infty(\mathbb{I})$ and $\mathcal{O}^\infty(\mathbb{RI})$ ignores the replication bounds and allows the indices of oracles to go to infinity. Using unbounded indices is helpful in Definition 8.22 below. By Assumption 4.14, 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 $\mathcal{O}^\infty(\mathbb{RI})$ cannot contain oracles under replication.

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 8.21 (Oracle sets) *Let $\mathcal{O}_{\mathbf{call}}(th)$ 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 $\mathcal{O}_{\mathbf{call}\text{-}\mathsf{repl}}(th)$ be the set of oracles $O[a, \widetilde{a}]$ such that O is under replication, $a > N_O$, and $\mathbf{call}(O[\_\,, \widetilde{a}])$ 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 $\mathbf{tagfunction}^{\mathsf{role},\tau}[\mathit{env}, \mathit{pm}'_{\mathsf{role}[\widetilde{a}]}]$ is present in the thread th, and such that env(token) is bound in its store to **Callable***.*

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

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

Let $\mathbb{CS} = \mathbb{C}, \mathbb{RI}, \mathbb{I}$. Let willbeavailable$(\mathbb{CS})$ be the set of oracles that can eventually become available. This set is defined as follows. We denote the callable *set of oracles:*

$$\mathit{callable}(\mathbb{CS}) \overset{\text{def}}{=} \mathcal{O}^\infty(\mathbb{I}) \cup \mathcal{O}^\infty(\mathbb{RI}) \cup \mathcal{O}^\infty(\mathcal{R}_{\textsf{init-closure}}(\mathbb{CS}) \cup \mathcal{R}_{\textsf{init-function}}(\mathbb{CS}))$$

We let $\mathit{oracleset}(Q)$ *(resp.* $\mathit{oracleset}(P)$*) be the set of oracles that may be defined by the process* Q *(resp.* P*), defined as follows:*

$$\begin{aligned}
\mathit{oracleset}(0) &\overset{\text{def}}{=} \emptyset \\
\mathit{oracleset}(Q_1 \mid Q_2) &\overset{\text{def}}{=} \mathit{oracleset}(Q_1) \cup \mathit{oracleset}(Q_2) \\
\mathit{oracleset}(\textsf{foreach } i' \leq n \textsf{ do } Q) &\overset{\text{def}}{=} \bigcup_{b=1}^{n} \mathit{oracleset}(Q\{b/i'\}) \\
\mathit{oracleset}(\textsf{role }\{Q) &\overset{\text{def}}{=} \mathit{oracleset}(Q) \\
\mathit{oracleset}(O[\widetilde{a}](x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]) := P) &\overset{\text{def}}{=} \{O[\widetilde{a}]\} \cup \mathit{oracleset}(P) \\
\mathit{oracleset}(P) &\overset{\text{def}}{=} \mathit{oracleset}(Q) \textit{ where } Q \textit{ is an oracle} \\
&\qquad \textit{definition located after a } \textsf{return} \textit{ in } P, \\
&\qquad \textit{or } \emptyset \textit{ if there is no } \textsf{return} \textit{ in } P.
\end{aligned}$$

By Property 4.5, the result is independent of the chosen return *statement in the last formula.*

We let $\mathit{returnoracles}'(O[\widetilde{a}]) \overset{\text{def}}{=} \mathit{oracleset}(P\{\widetilde{a}/\widetilde{i}\})$ *where oracle* O *is defined by* $O[\widetilde{i}](x_1[\widetilde{i}], \dots, x_k[\widetilde{i}]) := P$ *in* Q_0*. Finally, we define* $\mathit{willbeavailable}(\mathbb{CS}) \overset{\text{def}}{=} \bigcup_{O[\widetilde{a}] \in \mathit{callable}(\mathbb{CS})} \mathit{returnoracles}'(O[\widetilde{a}])$*.*

The definition of $\mathcal{O}_{\textbf{call-}\textsf{repl}}(th)$ 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 $\mathcal{O}_{\textbf{call-}\textsf{repl}}(th)$ is always included in $\mathcal{O}^\infty(\mathbb{I})$: 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 Item O2 of Definition 8.22 below.

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

Finally, we can define the desired invariant on simulator configurations:

Definition 8.22 (Oracles have distinct status) *Let* $\mathbb{CS} = ([th_1, \dots, th_n], \mathit{globalstore}, tj), \mathbb{RI}, \mathbb{I}$ *be a simulator configuration. We say that the oracles of* $\mathbb{CS}$ have distinct status *when:*

O1. The sets $\mathcal{O}^\infty(\mathbb{I}) \cup \mathcal{O}_{\textbf{call}}(\mathbb{CS})$*,* $\mathcal{O}^\infty(\mathbb{RI})$*, and* $\mathit{willbeavailable}(\mathbb{CS})$ *are pairwise disjoint.*

O2. The $4n$ *sets of oracles* $\mathcal{O}_{\textbf{call}}(th_i)$*,* $\mathcal{O}_{\textbf{call-}\textsf{repl}}(th_i)$*,* $\mathcal{O}^\infty(\mathcal{R}_{\textsf{init-function}}(th_i))$*, and* $\mathcal{O}^\infty(\mathcal{R}_{\textsf{init-closure}}(th_i))$ *for* $i \leq n$ *are pairwise disjoint, and are all included in* $\mathcal{O}^\infty(\mathbb{I}) \cup \mathcal{O}_{\textbf{call}}(\mathbb{CS})$*.*

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 $\mathcal{O}^\infty(\mathbb{RI})$; otherwise, it is in $\mathit{willbeavailable}(\mathbb{CS})$.

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

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 $\mathcal{O}_{\mathbf{call}}(th)$.

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

The case of an oracle under replication is fairly similar, using $\mathcal{O}_{\mathbf{call}\text{-}\mathsf{repl}}(th)$ instead of $\mathcal{O}_{\mathbf{call}}(th)$. Definition 8.22 ensures that an oracle cannot be simultaneously in two different sets. (We let indices go to infinity in $\mathcal{O}^\infty$ to make sure that we cannot have simultaneously $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathbb{I}$ with $a' > N_O$ and $O[b, \widetilde{a}] \in \mathit{willbeavailable}(\mathbb{CS})$ for all b. Indeed, if we bounded the indices to N_O, no oracle would correspond to $O\big[[a', +\infty[, \widetilde{a}\big]$ when $a' > N_O$, so this situation would not be prevented by Item O1. It is prevented using $\mathcal{O}^\infty$.)

Example 8.23 Let us show that the oracles of the initial simulator configuration $\mathbb{CS}_0$ of Example 8.16 have distinct status. We have $\mathcal{O}^\infty(\mathbb{I}_0) = \emptyset$, $\mathcal{O}_{\mathbf{call}}(\mathbb{CS}_0) = \emptyset$, $\mathcal{O}^\infty(\mathbb{RI}_0) = \{\mathrm{Okeygen}[\,]\}$, and $\mathit{willbeavailable}(\mathbb{CS}_0) = \{\mathrm{OA}[i] \mid 1 \leq i \leq N_1\} \cup \{\mathrm{OB}[i] \mid 1 \leq i \leq N_2\}$: the oracle Okeygen can be called immediately, just by starting the role keygen, the oracles OA and OB will be available later. Hence Item O1 holds. All sets of Item O2 are empty, so that item holds as well.

Let us also show that the oracles of the simulator configuration $\mathbb{CS}_1$ of Example 8.16 have distinct status. We have $\mathcal{O}^\infty(\mathbb{I}_1) = \{\mathrm{Okeygen}[\,]\}$, $\mathcal{O}_{\mathbf{call}}(\mathbb{CS}) = \{\mathrm{Okeygen}[\,]\}$ since the only **call** outside tagged functions and closures is $\mathbf{call}(\mathrm{Okeygen}[\,])$ in th^{s}_2, $\mathcal{O}^\infty(\mathbb{RI}_1) = \emptyset$, and $\mathit{willbeavailable}(\mathbb{CS}_1) = \mathit{willbeavailable}(\mathbb{CS}_0)$: a **call** to oracle Okeygen[ ] occurs in the thread th^{s}_2, and that call is allowed as shown by $\mathbb{I}_1$; the oracles OA and OB will be available later. Hence Item O1 holds. The only non-empty set in Item O2 is $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_2) = \{\mathrm{Okeygen}[\,]\}$, so that item holds as well. (There is a closure $\mathbf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ in th^{s}_2, but its token is **Invalid** because it has already been called.)

As a second step in our definition of the relation between the intermediate semantics and the OCaml semantics, we define an invariant of the intermediate semantics that shows how the sets $\mathbb{I}$ and $\mathbb{RI}$ of the simulator configuration represent the contents of the set of callable oracle definitions $\mathcal{Q}$.

Definition 8.24 (Relation between $\mathbb{I}$, $\mathbb{RI}$ and $\mathcal{Q}$) *Let us define the sets of oracles $\mathcal{O}(\mathbb{I})$ and $\mathcal{O}(\mathbb{RI})$ represented by $\mathbb{I}$ and $\mathbb{RI}$ respectively:*

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

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

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

In contrast to the sets we defined in Definition 8.20, the indices of oracles in $\mathcal{Q}$ are bounded by the replication bounds. So we redefine sets of oracles $\mathcal{O}(\mathbb{RI})$ and $\mathcal{O}(\mathbb{I})$ that correspond to $\mathbb{RI}$ and $\mathbb{I}$, but with indices bounded by N_O and N_{role} as appropriate. The sets $\mathcal{O}(\mathbb{RI})$ and $\mathcal{O}(\mathbb{I})$ are included in $\mathcal{O}^\infty(\mathbb{RI})$ and $\mathcal{O}^\infty(\mathbb{I})$, respectively. The set of processes $\mathcal{Q}$ corresponds to $\mathbb{RI}, \mathbb{I}$ when it contains exactly one definition for each oracle in $\mathcal{O}(\mathbb{I}) \cup \mathcal{O}(\mathbb{RI})$. 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 8.24.

Example 8.25 Let us consider the simulator configuration $\mathbb{CS}_1$ and the CryptoVerif configuration $\mathcal{C}_1$ of Example 8.16. Let $\mathcal{Q}_0 = \{\text{Okeygen}[\,]() := \dots(\text{as in the process } Q_0 \text{ of Example 4.10})\}$. Since $\mathcal{O}(\mathbb{I}_1) = \{\text{Okeygen}[\,]\}$ and $\mathcal{O}(\mathbb{RI}_1) = \emptyset$, we have $\mathcal{Q}_0 \leftrightarrow \mathbb{RI}_1, \mathbb{I}_1$ and $\mathcal{Q}_1 = \{Q_{\mathsf{loop}}\{a/i'\} \mid 1 < a \leq N_{\mathsf{rand+calls}}\} \cup \mathcal{Q}_0$. Hence, the sets $\mathbb{RI}_1$ and $\mathbb{I}_1$ correctly represent the callable oracle definitions $\mathcal{Q}_0$ that come from the protocol under consideration. The set of all callable oracle definitions $\mathcal{Q}_1$ additionally contains oracle definitions Q_{loop} that come from the simulator.

As a third step, we relate OCaml and simulator threads. To define this relation, we start from a simulator thread. We first replace the simulator role initialization with the OCaml one using the function *replaceinitpm* (Definition 8.26

below). Next, we replace **call** functional values with the corresponding closures (defined in Definition 8.27) using the function *replacecalls* (Definition 8.28). Finally, we add the part of the store that contains the tokens (Definition 8.29) and possibly an unreachable part of the store created during calls to cryptographic primitives, and we obtain the corresponding OCaml thread. The full relation between OCaml and simulator threads is defined in Definition 8.30.

Definition 8.26 (Replace initialization) *The function replaceinitpm replaces in its argument the pattern matchings corresponding to role initialization of the simulator with the OCaml module initialization: to be more precise, $replaceinitpm(th)$ replaces each occurrence of* $\mathbf{tagfunction}^{\mathsf{role}}\ pm'_{\mathsf{role}[\widetilde{a}]}$ *in th with* $\mathbf{tagfunction}^{\mathsf{role}}\ pm_{\mu_{\mathsf{role}}}$ *and each occurrence of* $\mathbf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *in th with* $\mathbf{tagfunction}^{\mathsf{role},\tau}[env, pm_{\mu_{\mathsf{role}}}]$.

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 8.27 (Correct closure) *Assume that* $\mathcal{Q} \leftrightarrow \mathbb{RI}, \mathbb{I}$ *for some* $\mathbb{RI}$, E *is a CryptoVerif environment,* l_{tok} *is a function that maps each oracle* $O[\widetilde{a}]$ *to the location of its token, and* τ_{O} *is a function that 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* $O[\widetilde{a}] \in \mathbb{I}$*:*

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

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

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

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

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

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

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

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

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

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

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

The case $O[\widetilde{a}] \notin \mathbb{I}$ 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 $\mathbb{I}$ 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 $\mathbb{I}$. 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 $\mathbf{call}(O[\_, \widetilde{a''}])$ 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 $\mathbf{call}(O[\_, \widetilde{a''}])$ can be mapped to closures that translate any process.

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

Definition 8.28 (Replace call) *Let* $\mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}$ *be as in Definition 8.27.*

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

The function *replacecalls* replaces in its argument each call $\mathbf{call}(O[\widetilde{a}])$ 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 8.29 (Token part of the store) *Let* $\mathbb{I}$ *and* l_{tok} *be as in Definition 8.27. Let* $\mathcal{O}$ *be a set of oracles with indices of the form* $O[\widetilde{a}]$*.*

$$\begin{aligned}\mathit{gettokens}(\mathbb{I}, \mathcal{O}, l_{\mathsf{tok}}) \stackrel{\text{def}}{=} &\{l_{\mathsf{tok}}(O[\widetilde{a}]) \mapsto \textbf{Callable} \mid O[\widetilde{a}] \in \mathcal{O} \cap \mathbb{I}\}\\ &\cup \{l_{\mathsf{tok}}(O[\widetilde{a}]) \mapsto \textbf{Invalid} \mid O[\widetilde{a}] \in \mathcal{O} \setminus \mathbb{I}\}\end{aligned}$$

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

In the following definitions, we use the exponent s for the elements of the simulator configuration and the exponent o for the elements of the OCaml configuration.

Definition 8.30 (Relation between simulator and OCaml threads) *Let* $th^{\mathsf{s}} = \langle env^{\mathsf{s}}, pe^{\mathsf{s}}, stack^{\mathsf{s}}, store^{\mathsf{s}}\rangle$ *be a simulator thread, and* $th^{\mathsf{o}} = \langle env^{\mathsf{o}}, pe^{\mathsf{o}}, stack^{\mathsf{o}}, store^{\mathsf{o}}\rangle$ *be an OCaml thread. Let* $\mathbb{I}, E, \mathcal{Q}, \tau_{\mathsf{O}}$ *be as in Definition 8.27. We say that* th^{o} matches th^{s} knowing $\mathbb{I}, E, \mathcal{Q}, \tau_{\mathsf{O}}$ *when one of the following two cases occurs:*

T1. $th^{\mathsf{o}} = replaceinitpm(th^{\mathsf{s}})$ *and* $th^{\mathsf{s}} = \langle \emptyset, program_{\mathsf{prim}};; program'(\mathsf{role}_1[\widetilde{a_1}]);; \dots;; program'(\mathsf{role}_l[\widetilde{a_l}]);; program', [\,], \emptyset\rangle$.

There is no closure, no tagged function $\mathbf{tagfunction}^t\ pm$*, no* **event***, and no* **return** *in* $program'$*, except in* $program(\mu_{\mathsf{role}})$ *in arguments of* **addthread**.

T2. The following properties hold:

(a) There exist $store'$ *and an injective function* l_{tok} *that associates to each* $O[\widetilde{a}]$ *in* $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}})$ *a store location that does not occur in* th^{s} *such that*

$$\begin{aligned}&\langle env^{\mathsf{o}}, pe^{\mathsf{o}}, stack^{\mathsf{o}}, store'\rangle\\ &\quad \in replacecalls(replaceinitpm(th^{\mathsf{s}}), \mathbb{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,,\\ &store' \cup gettokens(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}), l_{\mathsf{tok}}) \subseteq store^{\mathsf{o}}\,.\end{aligned}$$

(b) There exists an injective function $l_{\mathsf{init\text{-}tok}}$ *that associates to each role* $\mathsf{role}[\widetilde{a}]$ *such that a closure* $\mathbf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *occurs in the thread* th^{s} *for some* env *and* τ*, a store location such that for all closures* $\mathbf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *present in* th^{s}*, 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* $Dom(store^{\mathsf{s}})$ *and in* $env(token)$ *where* env *is the environment of a tagged closure* $\mathbf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *in* th^{s}.

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

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

This definition relates the threads of the simulator and of OCaml. A thread can be in one of the following two states. If it satisfies Item T1, 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 T2. In this case, Item T2(a) 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 $\mathbb{I}$ by the function *gettokens*, and we require that they are present in the OCaml store with their correct value. Item T2(b) 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 T2(b) also ensures that all locations used for the tokens of role initialization are not accessible elsewhere. Item T2(c) ensures that every tagged closure present in the simulator is a correct closure for the initialization of a role. Item T2(d) is an invariant of the simulator that ensures that the adversary does not have access to our OCaml instrumentation features.

Example 8.31 We use the notations $\mathbb{I}_1, E_1$ of Example 8.16, $\mathcal{Q}_0$ of Example 8.25, and let τ_{O} be any function. We verify that the thread th_2 of Example 7.3 matches the thread th^{s}_2 of Example 8.16 knowing $\mathbb{I}_1, E_1, \mathcal{Q}_0, \tau_{\mathsf{O}}$, because they satisfy Property T2. The function *replaceinitpm* replaces $\mathit{env}^{\mathsf{s}}_2(\mu_{\mathsf{keygen}}.\mathit{init})$ with $\mathit{env}_2(\mu_{\mathsf{keygen}}.\mathit{init})$. Let $l_{\mathsf{tok}} = \{\mathrm{Okeygen}[\,] \mapsto l_2\}$. We have $\mathbb{I}_1 = \{\mathrm{Okeygen}[\,]\}$, so

$$\begin{aligned}&\mathit{correctclosure}(\mathrm{Okeygen}[\,], \mathbb{I}_1, E_1, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \\ &\quad\{\mathbf{tagfunction}^{\mathrm{Okeygen},\tau}[\mathit{env}, \mathit{pm}_{\mathsf{Once}}(\mathcal{Q}_0(\mathrm{Okeygen}[\,]))] \mid \\ &\quad\ \mathit{env} \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_1, \mathcal{Q}_0(\mathrm{Okeygen}[\,])), \mathit{env}(\mathit{token}) = l_2\}\,.\end{aligned}$$

The process $\mathcal{Q}_0(\mathrm{Okeygen}[\,])$ is the definition of $\mathrm{Okeygen}[\,]$ in Example 4.10. It has no free variables, so $\mathit{env}(E_1, \mathcal{Q}_0(\mathrm{Okeygen}[\,])) = \emptyset$. Therefore, we have $\mathbf{tagfunction}^{\mathrm{Okeygen},\tau_2}[\mathit{env}_2 \oplus \{\mathit{token} \mapsto l_2\}, () \to$ (lines 5 to 13 of Example 7.1)$] \in \mathit{correctclosure}(\mathrm{Okeygen}[\,], \mathbb{I}_1, E_1, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, that is, the OCaml closure that corresponds to $\mathrm{Okeygen}[\,]$ is correct, so *replacecalls* replaces $\mathit{pe}^{\mathsf{s}}_2$ with pe_2, hence $\langle \mathit{env}_2, \mathit{pe}_2, \mathit{stack}_2, \mathit{store}^{\mathsf{s}}_2\rangle \in \mathit{replacecalls}(\mathit{replaceinitpm}(\mathit{th}^{\mathsf{s}}_2), \mathbb{I}_1, E_1, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Moreover, $\mathcal{O}_{\mathbf{call}}(\mathit{th}^{\mathsf{s}}_2) = \{\mathrm{Okeygen}[\,]\}$, so $\mathit{gettokens}(\mathbb{I}_1, \mathcal{O}_{\mathbf{call}}(\mathit{th}^{\mathsf{s}}_2), l_{\mathsf{tok}}) = \{l_2 \mapsto \mathbf{Callable}\}$, so $\mathit{store}^{\mathsf{s}}_2 \cup \mathit{gettokens}(\mathbb{I}_1, \mathcal{O}_{\mathbf{call}}(\mathit{th}^{\mathsf{s}}_2), l_{\mathsf{tok}}) = \mathit{store}_2$: the part of the store corresponding to tokens of oracles, here the token of $\mathrm{Okeygen}[\,]$, is computed by *gettokens*; it is included in the OCaml store but not in the simulator store. Therefore, Property T2(a) holds.

The only closure of the form $\mathbf{tagfunction}^{\mathsf{role},\tau}[\mathit{env}, \mathit{pm}'_{\mathsf{role}[\widetilde{a}]}]$ in $\mathit{th}^{\mathsf{s}}_2$ is $\mathit{env}^{\mathsf{s}}_2(\mu_{\mathsf{keygen}}.\mathit{init}) = \mathbf{tagfunction}^{\mathsf{keygen},\tau_1}[\mathit{env}_{\mathsf{prim}} \cup \{\mathit{token} \mapsto l_1\}, \mathit{pm}'_{\mathsf{keygen}[\,]}]$, so we define $l_{\mathsf{init\text{-}tok}} = \{\mathsf{keygen}[\,] \mapsto l_1\}$ and easily verify Property T2(b). Property T2(c) also concerns the same tagged closure, and is easily verified with $\widetilde{a}$ empty. There is no tagged function $\mathbf{tagfunction}^t\ \mathit{pm}$, no **event**, and no **return** in $\mathit{th}^{\mathsf{s}}_2$, so Property T2(d) holds, which concludes the verification of Property T2.

A similar verification can be done for th_1 and $\mathit{th}^{\mathsf{s}}_1$; we leave it to the reader.

Finally, we can define our relation between the intermediate and the OCaml semantics.

Definition 8.32 (Relation between extended CryptoVerif configurations and OCaml traces) *Let $\mathcal{C}^{\mathsf{cs}}$ be an extended CryptoVerif configuration and $\mathbb{CT}$ be an OCaml trace that starts with the initial configuration $\mathbb{C}_0(Q_0, \mathit{program}_0)$ defined in Section 7. We say that $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ when there exists an injective function τ_{O} that maps oracles $O[\_, \widetilde{a}]$ such that $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathbb{I}$ for some a' to tags τ, such that the following properties are all true:*

I1. $\mathcal{C}^{\mathsf{cs}} = E, P_{\mathsf{loop}}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E}, \mathit{steps}, \mathbb{CS}$.
$\mathbb{CS} = ([\mathit{th}^{\mathsf{s}}_1, \dots, \mathit{th}^{\mathsf{s}}_n], \mathit{globalstore}^{\mathsf{s}}, \mathit{tj}), \mathbb{RI}, \mathbb{I}$.

I2. $\mathbb{C}$ *is the last configuration of* $\mathbb{CT}$.
$\mathbb{C} = [\mathit{th}^{\mathsf{o}}_1, \dots, \mathit{th}^{\mathsf{o}}_n], \mathit{globalstore}^{\mathsf{o}}, \mathit{tj}, \mathbb{MI}, \mathit{events}$.

I3. $\mathcal{Q} = \{Q_{\mathsf{loop}}\{a/i'\} \mid \alpha < a \leq N_{\mathsf{rand+calls}}\} \cup \mathcal{Q}_0$ *and* $\mathcal{Q}_0 \leftrightarrow \mathbb{RI}, \mathbb{I}$.

I4. $\mathit{fv}(P_{\mathsf{loop}}\{\alpha/i'\}) \cup \mathit{fv}(\mathcal{Q}) \cup \mathit{fv}(\mathcal{S}_{\mathsf{loop}}(\alpha)) \subseteq \mathit{Dom}(E)$.

I5. For $i \leq n$, *all store locations in* Loc_ℓ *present in* $\mathit{th}^{\mathsf{s}}_i$ *are in* $\mathit{Dom}(\mathit{store}^{\mathsf{s}}_i)$, *where* $\mathit{th}^{\mathsf{s}}_i = \langle \mathit{env}^{\mathsf{s}}_i, \mathit{pe}^{\mathsf{s}}_i, \mathit{stack}^{\mathsf{s}}_i, \mathit{store}^{\mathsf{s}}_i \rangle$.

I6. For $i \leq n$, $\mathit{th}^{\mathsf{o}}_i$ *matches* $\mathit{th}^{\mathsf{s}}_i$ *knowing* $\mathbb{I}, E, \mathcal{Q}, \tau_{\mathsf{O}}$ *(Definition 8.30).*

I7. For all locations $l \in \mathit{Loc}_{\mathsf{priv}}$, l *does not occur in* $\mathit{th}^{\mathsf{s}}_1, \dots, \mathit{th}^{\mathsf{s}}_n$ *except in* $\mathit{program}(\mu_{\mathsf{role}})$ *in arguments of* **addthread**.

I8. $\forall l \in \mathit{Loc}_{\mathsf{priv}}, \mathit{globalstore}^{\mathsf{s}}(l) = \mathit{initval}_l$.

I9. $\mathit{globalstore}(E, \mathcal{T}) \subseteq \mathit{globalstore}^{\mathsf{o}}$.

I10. $\forall l \notin \mathit{Loc}_{\mathsf{priv}}, \mathit{globalstore}^{\mathsf{s}}(l) = \mathit{globalstore}^{\mathsf{o}}(l)$.

I11. $\mathbb{MI} = \{(\mu_{\mathsf{role}}, \mathsf{Once}) \mid \mathsf{role}[\widetilde{a}] \in \mathbb{RI}\} \cup \{(\mu_{\mathsf{role}}, \mathsf{Any}) \mid \mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathbb{RI}\}$.

I12. $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$.

I13. The oracles of $\mathbb{CS}$ *have distinct status (Definition 8.22).*

I14. $|\mathbb{CT}| + \mathit{steps} \geq N_{\mathsf{steps}}$.

I15. $\alpha \leq N_{\mathsf{rand}}(\mathbb{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathbb{CT}) + 1$.

I16. If $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathbb{I}$, *then* $a' \leq N_{\mathsf{calls}}(O, \tau_{\mathsf{O}}(O[\_, \widetilde{a}]), \mathbb{CT}) + 1$.

I17. If $\mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathbb{RI}$, *then* $a' \leq N_{\mathsf{exec}}(\mathsf{role}, \mathbb{CT}) + 1$.

The relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is our main tool to relate the CryptoVerif and OCaml systems. This relation holds only when the CryptoVerif adversary is evaluating the function $\text{simulate}_{\mathsf{ML}}$ (line 7 of Figure 18), as shown by the form of the extended CryptoVerif configuration $\mathcal{C}^{\mathsf{cs}}$ in Item I1. (The value α is the current value of the index i', that is, the number of iterations in the loop.) Items I1 and I2 also ensure that there is the same number of threads in the simulator configuration $\mathbb{CS}$ and in the OCaml configuration $\mathbb{C}$.

Item I3 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 $\mathbb{I}$, it is also present in $\mathcal{Q}$, and the oracle call in the CryptoVerif adversary (line 13 of Figure 18) can proceed. Item I4 is an invariant of the CryptoVerif semantics: the environment contains bindings for every free variable present in the current configuration. Item I5 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 I6 relates the threads of the simulator and of the OCaml semantics, following Definition 8.30.

Items I7 to I10 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 I10). The private part (files and tables) is empty in the simulator, since this part is handled by CryptoVerif itself (Item I8) and cannot be accessed by the adversary (Item I7). We require that the private part of the OCaml global store corresponds to the CryptoVerif configuration (Item I9).

Item I11 relates the OCaml multiset of callable modules $\mathbb{MI}$ and the simulator set of callable roles $\mathbb{RI}$. Item I12 relates the OCaml and CryptoVerif events. Item I13 guarantees that the oracles have distinct status, following Definition 8.22. This property allow us to prove that the injections l_{tok} and $l_{\mathsf{init\text{-}tok}}$ of Items T2(a) and T2(b) of Definition 8.30 are kept. (These injections appear in Item I6.)

Items I14 to I17 ensure that we never reach the limits on the number of simulator steps N_{steps} (Item I14), the number of calls to the oracles (Item I15 for the oracle $\mathrm{O}_{\mathsf{loop}}$ and Item I16 for the other oracles), and the number of calls to roles (Item I17), 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 $O\big[[a',+\infty[,\widetilde{a}\big] \in \mathbb{I}$, may be less than the number of calls to that oracle in the OCaml trace, $N_{\mathsf{calls}}(O, \tau_{\mathsf{O}}(O[\_,\widetilde{a}]), \mathbb{CT})$, because failed calls are not counted on the CryptoVerif side.

Example 8.33 We verify the relation $\mathcal{C}_1^{\mathsf{cs}} \equiv \mathbb{CT}_1$ after evaluating $\mu_{\mathsf{keygen}}.\mathit{init}\ ()$ in Example 7.3. The intermediate semantic configuration $\mathcal{C}_1^{\mathsf{cs}}$ is $\mathcal{C}_1^{\mathsf{cs}} = \mathcal{C}_1, \mathit{steps}, \mathbb{CS}_1$ where $\mathcal{C}_1$ and $\mathbb{CS}_1$ are defined in Example 8.16 and steps is N_{steps} minus the number of steps executed in the simulator. The OCaml trace $\mathbb{CT}_1$ ends at the configuration $\mathbb{C}_1$ defined in Example 7.3. We use the notations of these examples.

Properties I1 and I2 are obvious from the form of the configurations, with

$\alpha = 1$, $n = 2$, and $tj = 2$. Properties I3, I6, and I13 have been verified in Examples 8.25, 8.31, and 8.23 respectively. Property I4 can be verified by computing the value of E_1; we leave this detail to the reader. There is no store location in th^{s}_1 and the only store location of th^{s}_2 is l_1, which is in $Dom(store^{\mathsf{s}}_1)$, so Property I5 holds.

The locations *pkfile* and *skfile* do not occur in th^{s}_1 nor th^{s}_2, so Property I7 holds. (They occur in $program(\mu_{\mathsf{keygen}})$ in the argument of **addthread** in the initial program $program_0$, but they disappear when **addthread** is executed.) For $l \in \{skfile, pkfile\}$, $globalstore^{\mathsf{s}}_1(l) = inital_l$ so Property I8 holds. We have $globalstore(E_1, \mathcal{T}_1) = \emptyset$ because neither $sk[\,]$ nor $pk[\,]$ are defined in E_1, so Property I9 holds, and $globalstore^{\mathsf{s}}_1(pkg) = globalstore_1(pkg)$, so Property I10 holds. (The verification of the correspondence between the global stores is not very interesting in this configuration. It is more interesting at the end of the execution of $program_0$. At this point, $globalstore^{\mathsf{o}} = \{skfile \mapsto v_{sk}, pkfile \mapsto v_{pk}, pkg \mapsto v_{pk}\}$, since *skfile* and *pkfile* are written by $\mu_{\mathsf{keygen}}.init$ () () and *pkg* is written by $program_0$, while $globalstore^{\mathsf{s}} = \{skfile \mapsto "", pkfile \mapsto "", pkg \mapsto v_{pk}\}$ since the simulator calls Okeygen via CryptoVerif, which does not write into files. The minimal global store $globalstore(E, \mathcal{T})$ contains values for *skfile* and *pkfile* since $sk[\,]$ and $pk[\,]$ are defined in the CryptoVerif environment E after calling Okeygen and they should be stored in the files *skfile* and *pkfile* respectively. These values are indeed in $globalstore^{\mathsf{o}}$, so Property I9 holds. However, $globalstore^{\mathsf{s}}(l)$ still contains the initial values for $l \in \{skfile, pkfile\}$, so Property I8 holds. The same value for *pkg* appears in $globalstore^{\mathsf{s}}$ and $globalstore^{\mathsf{o}}$, so Property I10 holds.)

We have $\mathbb{MI}_1 = \emptyset$ and $\mathbb{RI}_1 = \emptyset$, so Property I11 holds; $events_1 = [\,]$ and $\mathcal{E}_1 = [\,]$, so Property I12 holds. Property I14 can be verified by counting the number of steps in OCaml and in the simulator. We omit this tedious but not difficult point here. Property I15 holds because $\alpha = 1$; there are no random number generations nor oracle calls in $\mathbb{CT}_1$. Properties I16 and I17 hold because neither $\mathbb{I}_1 = \{\text{Okeygen}[\,]\}$ nor $\mathbb{RI}_1 = \emptyset$ contain oracles of the considered form.

The next two lemmas show that the relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is preserved during execution. Lemma 8.34 shows that it holds at the beginning, as soon as the simulator reaches line 7 of Figure 18.

Lemma 8.34 *There exists a trace* $\mathcal{C}_0(Q_0, program_0) \rightsquigarrow^* \mathcal{C}^{\mathsf{cs}}$ *where* $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}_0$ *and* $\mathbb{CT}_0 = \mathbb{C}_0(Q_0, program_0)$.

Lemma 8.35 shows that the relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is preserved. More precisely, the relation does not hold at all steps (in particular because it holds only when the CryptoVerif adversary is executing $\text{simulate}_{\mathsf{ML}}$), 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 8.35 *Let* $\mathcal{C}^{\mathsf{cs}}$ *be such that there exists a trace* $\mathbb{CT}$ *satisfying* $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$.

- *Either there exist n configurations* $\mathcal{C}^{\mathsf{cs}}_1, \dots, \mathcal{C}^{\mathsf{cs}}_n$ *and n traces* $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow^+_{p_1} \mathcal{C}^{\mathsf{cs}}_1$, $\dots, \mathcal{C}^{\mathsf{cs}} \rightsquigarrow^+_{p_n} \mathcal{C}^{\mathsf{cs}}_n$ *such that none of these traces is a prefix of another,*

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

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

We prove these lemmas in Appendix F. Let us present a proof sketch of Lemma 8.35.

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

Case 1: the current thread of $\mathbb{CS}$ verifies Item T1 of Definition 8.30, we run the initialization of the module. The programs of the current threads of $\mathbb{CS}$ and $\mathbb{C}$ are the same except that the occurrences of *program*$'(\mathsf{role}[\widetilde{a}])$ present in $\mathbb{CS}$ are transformed into *program*(μ_{role}). We show that after having reduced the initialization of the primitives and the initialization of the roles on both sides, the current threads verify Item T2. The oracles in $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}function}}(th^{\mathsf{s}}))$ that correspond to the roles implemented in this initialization are moved to $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(th^{\mathsf{s}}))$. We prove that the relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is preserved.

Case 2: the current thread of $\mathbb{CS}$ verifies Item T2. We distinguish cases on the form of the simulator configuration $\mathbb{CS}$.

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

- If the current expression of $\mathbb{CS}$ is $\mathbf{call}(O_j[\widetilde{a}])\ v$, then the result of simulate$_{\mathsf{ML}}$ is such that $o = \mathrm{o}_j$, so the CryptoVerif adversary of Figure 18 calls the oracle O_j at line 13 in the branch $o = \mathrm{o}_j$, ends one iteration of $\mathrm{O}_{\mathsf{loop}}$, and starts the next iteration until it reaches line 7. We use Lemma 8.10 and we exploit the definition of simulate$_{\mathsf{ret}O_j}$ and simulate$_{\mathsf{end}O_j}$ 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 $\mathcal{O}^{\infty}(\mathbb{I})$, and from $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}})$ if all occurrences of $\mathbf{call}(O_j[\widetilde{a}])$ have disappeared. The newly available oracles, added to sets $\mathcal{O}^{\infty}(\mathbb{RI})$ or $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}})$ and $\mathcal{O}_{\mathbf{call\text{-}repl}}(th^{\mathsf{s}})$, are removed from the set *willbeavailable*$(\mathbb{CS})$. We prove that the relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is preserved.

- If the current expression of $\mathbb{CS}$ is **random** $()$, then the result of simulate$_{\mathsf{ML}}$ is such that $o = \mathrm{o}_{\mathsf{R}}$, so the CryptoVerif adversary of Figure 18 samples a random boolean at line 19, ends one iteration of $\mathrm{O}_{\mathsf{loop}}$, and starts the next iteration until it reaches line 7. The current expression of $\mathbb{CS}$ 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 evaluating **random** (), and the relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is preserved.

- Otherwise, the configuration $\mathbb{CS}$ cannot reduce, and the corresponding configuration $\mathbb{C}$ cannot reduce either. The result of simulate$_{\mathsf{ML}}$ is such that $o = o_{\mathsf{S}}$, so the CryptoVerif adversary of Figure 18 ends the current iteration of $\mathsf{O}_{\mathsf{loop}}$ 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 $\mathbb{CS}$ is **addthread**(*program*), a new thread is created on both sides. If *program* is a protocol program, then this new thread satisfies Item T1 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 $\mathbb{RI}$ and the corresponding oracles are added to $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}function}}(th))$ and to $\mathbb{I}$. Otherwise, the new thread satisfies Item T2. We prove that the relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is preserved.

If the current expression of $\mathbb{CS}$ is $\mathbf{call}(O_j[\widetilde{a}])\ v$ and the configuration $\mathbb{CS}$ 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 $\mathbb{I}$ in the simulator by Item T2(a)). We prove that the relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is preserved.

If the current expression of $\mathbb{CS}$ is $\mathbf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]\ ()$, then the initialization function of role role is executed. This role is removed from $\mathcal{R}_{\mathsf{init\text{-}closure}}(th^{\mathsf{s}})$, and the corresponding oracles are added to $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}})$ and to $\mathcal{O}_{\mathbf{call\text{-}repl}}(th^{\mathsf{s}})$. We prove that the relation $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$ is preserved.

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

From Lemmas 8.34 and 8.35, we can prove the following proposition, by extending the traces using Lemma 8.35 until we get complete traces.

Proposition 8.36 *Let $\mathcal{CT}_1, \dots, \mathcal{CT}_n$ be complete CryptoVerif traces starting at $\mathcal{C}_0(Q_0, program_0)$.*

Then there exist disjoint sets of complete OCaml traces $\mathbb{CTS}_1, \dots, \mathbb{CTS}_n$ all starting at $\mathbb{C}_0(Q_0, program_0)$ such that for all $i \leq n$, $\Pr[\mathcal{CT}_i] = \Pr[\mathbb{CTS}_i]$, and if $\mathcal{C}$ is the last configuration of $\mathcal{CT}_i$ and $\mathbb{C}$ is the last configuration of a trace in $\mathbb{CTS}_i$, then the event list $\mathcal{E}$ of $\mathcal{C}$ and the event list events of $\mathbb{C}$ 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 8.37 $\Pr[\mathcal{C}_0(Q_0, program_0) :^{(\rightsquigarrow)} D] = \Pr[\mathbb{C}_0(Q_0, program_0) :^{(\mathsf{ML})} D]$.

8.4.3 Security Result

By combining Propositions 8.19 and 8.37, we obtain the following theorem:

Theorem 8.38 (Security result)

$$\Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) :^{(\mathsf{CV})} D] = \Pr[\mathbb{C}_0(Q_0, \mathit{program}_0) :^{(\mathsf{ML})} D]\,.$$

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 $Q_{\mathsf{adv}}(Q_0, \mathit{program}_0)$ 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,

$$\Pr[\mathcal{C}_{\mathsf{i}}(Q_0 \mid Q) :^{(\mathsf{CV})} D] \leq p\,.$$

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

$$\Pr[\mathbb{C}_0(Q_0, \mathit{program}) :^{(\mathsf{ML})} D] = \Pr[\mathcal{C}_0(Q_0, \mathit{program}) :^{(\mathsf{CV})} D] \leq p$$

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 8.11, 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 $\mathbb{G}_{\mathsf{T}}(T)$ is fixed by the annotations of the CryptoVerif specification. The size of T can be set to the size of $\mathbb{G}_{\mathsf{T}}(T)$. 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 $\mathsf{simulate}_{\mathsf{ML}}$ executes at most as many reduction steps as the OCaml adversary. However, the CryptoVerif adversary shown in Figure 18 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.

Example 8.39 For the protocol Q_0 of Example 4.1, using Theorem 8.38 and the probability bound computed by CryptoVerif in Example 4.9, we obtain that our generated implementation satisfies

$$\Pr[\mathbb{C}_0(Q_0, \mathit{program}) :^{(\mathsf{ML})} D_c] \leq \mathsf{Succ}^{\mathsf{uf-cma}}_{\mathrm{sign}}(t + (N_2 - 1)t_{\mathrm{check}}, N_1)$$

where t is the execution time of the adversary *program*, t_{check} is the maximum execution time of a call to the implementation of check, N_1 is the maximum number of calls to oracle OA, N_2 is the maximum number of calls to oracle OB, and $\mathsf{Succ}^{\mathsf{uf-cma}}_{\mathrm{sign}}(t', n')$ is the probability of forging a signature in time t' with at most n' calls to the signature oracle.

As detailed in [9], 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 8.38, our generated implementation of this protocol satisfies the same properties, provided assumptions A1 to A7 hold.

9 Conclusion

We have proved that our compiler preserves security. Therefore, by using CryptoVerif, we can prove the desired security properties on the protocol specification, and then by using our compiler, we get a runnable implementation of the protocol, which satisfies the same security properties as the specification. Making such a proof is also useful because it clarifies the assumptions needed to ensure that the implementation is secure (Assumptions A1 to A7 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.

Our approach could obviously be used to generate implementations in languages other than OCaml. It should not be difficult to adapt our compiler to another language. The structure of the proof should also remain the same, but obviously the details will need to be adapted to the semantics of each programming language. In a target language such as C, closures that we use to represent oracles could be represented by records containing a function pointer. Since C does not guarantee memory safety, an additional analysis of the network code should be performed to make sure that it does not access private data of our generated code. To simplify the analysis, one may require that the generated code and the network code belong to a clean subset of C. One might also go all the way to the generation of certified machine code, by using a certified compiler, as in [3].

Extending the specification language of CryptoVerif, for instance with loops and mutable data structures, would be helpful to implement real, complex protocols. The main difficulty in this task does not lie in the generation of implementations, but in the extension of the prover CryptoVerif itself. Formalizing our manual proof using a proof assistant (e.g. Coq) would also be interesting future work. We believe that our detailed proof will be a good starting point for that. It would also be interesting to extend our approach to support side channel attacks, such as timing attacks and power consumption attacks. Protection against such attacks is important in practical protocols.

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). The authors thank the reviewers for their helpful comments on a previous version of this paper.

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. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, pages 1217–1230, Berlin, Germany, Nov. 2013. ACM.

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

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

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

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

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

[9] 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.

[10] 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.

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

[12] R. Corin and F. A. Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In Ú. Erlingsson, R. Wieringa, and N. Zannone, editors, *Engineering Secure Software and Systems (ESSoS'11)*, volume 6542 of *LNCS*, pages 58–72, Madrid, Spain, Feb. 2011. Springer.

[13] 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.

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

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

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

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

[18] S. Owens, G. Peskine, and P. Sewell. A formal specification for OCaml: the core language. Available at `http://www.cl.cam.ac.uk/~so294/ocaml/caml_typedef.pdf`, 2008.

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

[20] 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

A Proof of Lemma 4.7

Proof (of Lemma 4.7) 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 $\mathit{Dom}(S) \subseteq \mathit{Dom}(S')$ and $\forall x \in \mathit{Dom}(S), S(x) \leq S'(x)$.

We define the multiset of available oracles inductively as follows:

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

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

Let us first show this property for the initial configuration. We show by an easy induction on Q that $\biguplus_{Q' \in \mathit{oracledefset}(Q)} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q)$. Therefore, by definition of $\mathcal{C}_{\mathsf{i}}$, $\mathit{Oracles}(\mathcal{C}_{\mathsf{i}}(Q_0)) = \biguplus_{Q' \in \mathit{oracledefset}(Q_0)} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q_0)$. 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 4.6, 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 foreach $i \leq n$ do Q, by Property 4.6, 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 $O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \dots, x_k[\widetilde{a}] : T_k) := P$, the definition of O is not in a branch of if or get different from P, so by Property 4.6, 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, $\mathit{Oracles}(\mathcal{C})$ decreases by reduction: if $\mathcal{C} \to_p \mathcal{C}'$, then we have $\mathit{Oracles}(\mathcal{C}') \subseteq \mathit{Oracles}(\mathcal{C})$. Indeed, the rules (New), (Let), (Insert), (Event), (Loop1) leave $\mathit{Oracles}(\mathcal{C})$ unchanged. In the case of (Loop1), we use

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

The rules (If1), (If2), (Get1), (Get2), (Loop2), (Return), (End) decrease the multiset $\mathit{Oracles}(\mathcal{C})$ 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 $\biguplus_{Q' \in \mathit{oracledefset}(Q'')} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q'')$. The rule (Call) removes the called oracle $O[\widetilde{a'}]$ from $\mathit{Oracles}(\mathcal{C})$.

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

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

B Proof of Proposition 6.5

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

Proof (of Lemma 6.3) 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. $\square$

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 $\mathbb{CI}$ can reduce, then

$$\sum_{\{\mathbb{CI}' \mid \mathbb{CI} \rightarrow_{p(\mathbb{CI}')} \mathbb{CI}'\}} p(\mathbb{CI}') = 1\,.$$

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

Proof (of Proposition 6.5) Let $n_{\mathsf{ev,ret}}(\mathbb{CI})$ be the number of occurrences of **event** or **return** in the instrumented configuration $\mathbb{CI}$. Let us first prove the following property:

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

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

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

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

Let us first suppose that $\mathbb{CI}$ reduces by (Globalstore1) and (Toplevel) from a reduction $\mathit{th} \longrightarrow_p \mathit{th}'$ 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 $\mathit{th} = \langle \mathit{env}, C_{\mathsf{m}}[e], \mathit{stack}, \mathit{store}\rangle \longrightarrow \mathit{th}' = \langle \mathit{env}, e, (\mathit{env}, C_{\mathsf{m}}) :: \mathit{stack}, \mathit{store}\rangle$ where e is not a value and C_{m} is a minimal expression or program evaluation context. Let us distinguish cases on the form of C_{m}.
  - If $C_{\mathsf{m}} = \mathbf{return}(\mathbb{MI}, [\cdot])$, then $\mathit{noinstr}_{\mathit{th}}(\mathit{th}) = \mathit{noinstr}_{\mathit{th}}(\langle \mathit{env}, e, \mathit{stack}, \mathit{store}\rangle) = \mathit{noinstr}_{\mathit{th}}(\mathit{th}')$ by Definition 6.4, so by expanding this property to the complete configuration, and noting that the reduction removes one **return**, Property P1 holds.
  - If $C_{\mathsf{m}} = \mathbf{event}\ \mathit{ev}(e_1, \dots, e_{i-1}, [\cdot], v_{i+1}, \dots, v_n)$, then we have by Definition 6.4,

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

    and we have $\mathit{noinstr}_{\mathit{th}}(\mathit{th}) \longrightarrow \mathit{noinstr}_{\mathit{th}}(\mathit{th}')$, so Property P2 holds.

- If C_{m} is neither a return nor an event context, then the reduction $\mathit{th} \longrightarrow_p \mathit{th}'$ implies $\mathit{noinstr}_{\mathit{th}}(\mathit{th}) \longrightarrow_p \mathit{noinstr}_{\mathit{th}}(\mathit{th}')$, so Property P2 holds.

- The reduction comes from rules (Context out) or (Let ctx out): we have $\mathit{th} = \langle \mathit{env}, v, (\mathit{env}', C_{\mathsf{m}}) :: \mathit{stack}, \mathit{store}\rangle \longrightarrow \mathit{th}' = \langle \mathit{env}', C_{\mathsf{m}}[v], \mathit{stack}, \mathit{store}\rangle$ where C_{m} is a minimal expression or program evaluation context. Let us distinguish cases on the form of C_{m}.

  - If $C_{\mathsf{m}} = \mathbf{return}(\mathbb{MI}, [\cdot])$, then we have by Definitions 6.4 and 6.2,

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

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

  - If $C_{\mathsf{m}} = \mathbf{event}\ \mathit{ev}(e_1, \dots, e_{i-1}, [\cdot], v_{i+1}, \dots, v_n)$, we have by Definition 6.4,

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

    so Property P2 holds.

  - If C_{m} 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 $\mathbf{return}(\mathbb{MI}, [\cdot])$; Property P2 holds otherwise.

- Property P2 holds in the other cases.

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

If $\mathbb{CI} \to \mathbb{CI}_1$ by (Toplevel event), then the program of the current thread is **event** $\mathit{ev}(v_1, \dots, v_n)$ in $\mathbb{CI}$ and the only differences between $\mathbb{CI}$ and $\mathbb{CI}_1$ are that the program of the current thread is $(v_1, \dots, v_n)$ and the list of executed events is updated in $\mathbb{CI}_1$. Therefore, $\mathit{noinstr}_{\mathbb{CI}}(\mathbb{CI}) = \mathit{noinstr}_{\mathbb{CI}}(\mathbb{CI}_1)$, 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 6.1.

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 $\mathbb{C} \approx_v \mathit{noinstr}_{\mathbb{CI}}(\mathbb{CI})$ and $\mathbb{CI}$ cannot reduce, then $\mathbb{C}$ cannot reduce.

We need to prove that $\mathit{noinstr}_{\mathbb{CI}}(\mathbb{CI})$ cannot reduce. We can then use Lemma 6.3 to conclude. We distinguish cases depending on the program or expression in the current thread of $\mathbb{CI}$. If this program or expression was of the form $C_{\mathsf{m}}[e]$ for some program or expression minimal evaluation context C_{m}, or $\mathbf{return}(\mathbb{MI}, v)$, or **event** $ev(v_1, \dots, v_n)$, the configuration $\mathbb{CI}$ could reduce. In all other cases, $\mathit{noinstr}_{\mathbb{CI}}$ 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 $n_{\mathsf{ev,ret}}(\mathbb{CI})$. Let us suppose that $\mathbb{C} \approx_v \mathit{noinstr}_{\mathbb{CI}}(\mathbb{CI})$ and $\mathbb{C}_1, \dots, \mathbb{C}_n$ are pairwise distinct configurations such that for all $i \leq n$, we have $\mathbb{C} \rightarrow_{p_i} \mathbb{C}_i$ with $\sum_{i \leq n} p_i = 1$.

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

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

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

C Proof of Proposition 8.5

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

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

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

1. *all store locations* $l \in Loc_\ell$ *that occur in the thread* th *are bound in the store:* $l \in Dom(store)$,
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 $th \to_p th'$, 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 $th \to_p th'$ so that they do not occur in th'', we have $plug(th, th'') \to_p plug(th', th'')$ 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 $th \overset{\text{def}}{=} \langle env, pe, stack, store\rangle$ and $th' \overset{\text{def}}{=} \langle env', pe', stack', store'\rangle$. Let us prove that, if $th \to_p th'$, then for every $th'' \overset{\text{def}}{=} \langle env'', pe'', stack'', store''\rangle$ such that $Dom(store) \cap Dom(store'') = \emptyset$, we have the reduction $plug(th, th'') = \langle env, pe, stack \mathbin{@} stack'', store \cup store''\rangle \to_p plug(th', th'') = \langle env', pe', stack' \mathbin{@} stack'', store' \cup store''\rangle$ with $Dom(store') \cap Dom(store'') = \emptyset$. 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 $\mathbf{ref}\ v = l$, then by (Store alloc), $l \notin Dom(store)$. By reviewing the uses of L of the form $\mathbf{ref}\ v = l$ in the reduction rules, we can deduce that $pe = \mathbf{ref}\ v$ and $pe' = l$. By Property 1 of Definition C.2, the location l does not occur in th. Let us take a location $l' \in Loc_\ell$ that is not in $Dom(store) \cup Dom(store'')$; we rename l into l'. As $l' \notin Dom(store)$, 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 $L' \overset{\text{def}}{=} (\mathbf{ref}\ v = l')$. By Property (P1), $env, pe, stack \mathbin{@} stack'' \xrightarrow{L'} env', l', stack' \mathbin{@} stack''$ and, since $l' \notin Dom(store) \cup Dom(store'')$, we have $store \cup store'' \xrightarrow{L'} store[l' \mapsto v] \cup store''$. We conclude that $plug(th, th'') = \langle env, pe, stack \mathbin{@} stack'', store \cup store''\rangle \to plug(th', th'') = \langle env', l', stack' \mathbin{@} stack'', store[l' \mapsto v] \cup store''\rangle$ by rule (Thread). $\square$

Lemma C.4 *Let th be a well-formed thread. If $th \to_p th'$, 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. $\square$

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 Loc_ℓ and by Assumption 8.1, it contains no **schedule**, **addthread**, **return**, or **event** operations and no global store locations.

By Assumption 8.2, $th_0^s \to^* th = \langle env_{\mathsf{prim}}, \varepsilon, [\,], \emptyset\rangle$, 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 8.3, 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. $\square$

Proof (of Proposition 8.5) By Assumption 8.4, we have reductions of the form

$$th_1 \overset{\text{def}}{=} \langle \emptyset, env_{\mathsf{prim}}(s)\ v, [\,], \emptyset\rangle \to_p^* th_1' \overset{\text{def}}{=} \langle env', v', [\,], store_1'\rangle\,.$$

Let $th_2 \overset{\text{def}}{=} \langle env, (), stack, store\rangle$. 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 $store' \overset{\text{def}}{=} store_1' \cup store$, we obtain exactly the desired reductions $th \to_p^* \langle env', v', stack, store'\rangle$, and $store' \supseteq store$. $\square$

D Proof of Lemmas 8.8 and 8.10

Let us first prove Lemma 8.8.

Proof (of Lemma 8.8) We prove this result by induction on the syntax of terms.

- Case $M = x[\widetilde{a'}]$: Since $E \cdot M \Downarrow a$ 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

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

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

$$\begin{aligned}
\mathit{th} &= \langle \mathit{env}, \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_k)), \mathit{stack}, \mathit{store}\rangle \\
&\to \langle \mathit{env}, (\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_k)), \mathit{stack}', \mathit{store}\rangle \\
&\quad \text{where } \mathit{stack}' \stackrel{\text{def}}{=} (\mathit{env}, \mathbb{G}_{\mathsf{f}}(f)\ [\cdot]) :: \mathit{stack} \\
\to \mathit{th}_1 &\stackrel{\text{def}}{=} \langle \mathit{env}, \mathbb{G}_{\mathsf{M}}(M_k), \mathit{stack}'', \mathit{store}\rangle \\
&\quad \text{where } \mathit{stack}'' \stackrel{\text{def}}{=} (\mathit{env}, (\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_{k-1}), [\cdot])) :: \mathit{stack}' \\
\to^* \mathit{th}_2 &\stackrel{\text{def}}{=} \langle \mathit{env}', \mathbb{G}_{\mathsf{val}T_k}(a_k), \mathit{stack}'', \mathit{store}'\rangle \\
&\qquad\qquad \text{by induction hypothesis applied to } M_k \\
&\to \langle \mathit{env}, (\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_{k-1}), \mathbb{G}_{\mathsf{val}T_k}(a_k)), \mathit{stack}', \mathit{store}'\rangle \\
\to^* \mathit{th}_3 &\stackrel{\text{def}}{=} \langle \mathit{env}, (\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), \mathit{stack}', \mathit{store}''\rangle \\
&\qquad\qquad \text{by an easy induction} \\
&\to \langle \mathit{env}, \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), \mathit{stack}, \mathit{store}''\rangle \\
&\to \langle \mathit{env}, \mathbb{G}_{\mathsf{f}}(f), \mathit{stack}''', \mathit{store}''\rangle \\
&\quad \text{where } \mathit{stack}''' \stackrel{\text{def}}{=} (\mathit{env}, [\cdot]\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_k}(a_k))) :: \mathit{stack} \\
&\to \langle \mathit{env}, \mathit{env}_{\mathsf{prim}}(\mathbb{G}_{\mathsf{f}}(f)), \mathit{stack}''', \mathit{store}''\rangle \qquad \text{since } \mathit{env}_{\mathsf{prim}} \subseteq \mathit{env} \\
&\to \langle \mathit{env}, \mathit{env}_{\mathsf{prim}}(\mathbb{G}_{\mathsf{f}}(f))\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), \mathit{stack}, \mathit{store}''\rangle \\
\to^* \mathit{th}' &\stackrel{\text{def}}{=} \langle \mathit{env}'', \mathbb{G}_{\mathsf{val}T}(a), \mathit{stack}, \mathit{store}'''\rangle \qquad \text{by Proposition 8.5}
\end{aligned}$$

By Proposition 8.5 and induction hypothesis, we have $\mathit{store}''' \supseteq \mathit{store}'' \supseteq \mathit{store}' \supseteq \mathit{store}$. □

Let us now prove lemmas useful to prove Lemma 8.10.

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

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

Proof If $(x[\widetilde{a}], f) \in \mathit{Files}$ 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

$$\begin{aligned}
&\mathbb{C} \to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x)), \mathit{stack}', \mathit{store}\rangle]\\
&\qquad \text{where } \mathit{stack}' \overset{\text{def}}{=} (\mathit{env}, f := [\cdot]) :: \mathit{stack}\\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathit{env}_{\mathsf{prim}}(\mathbb{G}_{\mathsf{ser}}(T_x))\ \mathbb{G}_{\mathsf{val}T_x}(a), \mathit{stack}', \mathit{store}\rangle]\\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}', \mathit{ser}(T_x, a), \mathit{stack}', \mathit{store}'\rangle] \qquad \text{by Proposition 8.5}\\
&\quad \to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, f := \mathit{ser}(T_x, a), \mathit{stack}, \mathit{store}'\rangle]\\
&\to \mathbb{C}' = \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, (), \mathit{stack}, \mathit{store}'\rangle, \mathsf{globalstore} \mapsto \mathit{globalstore}']
\end{aligned}$$

where

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

The modified location f is in $\mathit{Loc}_{\mathsf{priv}}$, so for all $l \notin \mathit{Loc}_{\mathsf{priv}}$, $\mathit{globalstore}'(l) = \mathbb{C}_{\mathit{globalstore}}(\mathbb{C})(l)$. We have $\mathit{store}' \supseteq \mathit{store}$ by Proposition 8.5.

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

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

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

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 $(b_1, \dots, b_l)$ of this table is*

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

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

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

Definition D.3 (Function *filter*) *Let E be a CryptoVerif environment, M a CryptoVerif boolean term, $(x_1, \dots, x_k)$ 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 $\mathit{filter}(E, M, (x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]), t)$ be the list of tuples $(a_1, \dots, a_k)$ in t such that the term M is true when the variables $x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]$ are bound to $a_1, \dots, a_k$ in the environment E, respectively:*

$$\begin{aligned}
&\mathit{filter}(E, M, (x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]), t) \overset{\text{def}}{=}\\
&\qquad [(a_1, \dots, a_k) \in t \mid E[x_1[\widetilde{a}] \mapsto a_1, \dots, x_k[\widetilde{a}] \mapsto a_k] \cdot M \Downarrow \text{true}]
\end{aligned}$$

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

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

Lemma D.4 *Suppose that*

$$\begin{aligned}
th &= \langle env, fold\ c\ [\,]\ (\mathbb{G}_{\mathsf{tbl}}(Tbl, t)), stack, store\rangle\\
c &= \mathbf{function}[env',\\
&\qquad a \to \mathbf{function}\ x \to \mathbf{try}\ (c'\ x) :: a\ \mathbf{with\ Match\_failure} \to a]\\
c' &= \mathbb{G}_{\mathsf{test}}((x_1, \dots, x_k), M)\\
env' &\supseteq env_{\mathsf{prim}} \cup \{\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T_x}(b) \mid x[\widetilde{a'}] \in fv(M) \setminus \{x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]\},\\
&\qquad\qquad E(x[\widetilde{a'}]) = b\}\\
env(fold) &= env'(fold) = \mathbf{letrec}[env_0, \{fold \mapsto \mathbb{G}_{\mathsf{fold}}\}\ \mathbf{in}\ fold]
\end{aligned}$$

where t is a list of CryptoVerif values of type $T_{x_1} \times \cdots \times T_{x_k}$ and all occurrences of $x_1, \dots, x_k$ in M have indices $\widetilde{a}$.

Then $th \to^* th'$ such that

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

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

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

For any $E, M, (x_1[\widetilde{a}], \dots, x_k[\widetilde{a}])$, we have $filter(E, M, (x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]), [\,]) = [\,]$. 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 $b = (a_1, \dots, a_k)$ and $y = (d_1, \dots, d_k)$, where each d_i corresponds to a_i. Let $(d'_1, \dots, d'_k) = \mathbb{G}_{\mathsf{val}T_1,\dots,T_k}(b)$, where Tbl is a table of type $T_1 \times \cdots \times T_k$.

$$\begin{aligned}
th &= \langle env, fold\ c\ [\,]\ (y :: l'), stack, store\rangle\\
&\to^* \langle env_1, (\mathbf{match}\ c\ \mathbf{with}\ \mathbb{G}_{\mathsf{fold}})\ [\,]\ (y :: l'), stack, store\rangle\\
&\qquad \text{where } env_1 \overset{\text{def}}{=} env_0[fold \mapsto env'(fold)]\\
&\to^* \langle env_2, \mathbf{match}\ y :: l'\ \mathbf{with}\ [\,] \mapsto a \mid x :: l \to f\ (fold\ f\ a\ l)\ x, stack, store\rangle\\
&\qquad \text{where } env_2 \overset{\text{def}}{=} env_1[f \mapsto c, a \mapsto [\,]]\\
&\to^* \langle env_3, f\ (fold\ f\ a\ l)\ x, stack, store\rangle\\
&\qquad \text{where } env_3 \overset{\text{def}}{=} env_2[x \mapsto y, l \mapsto l']
\end{aligned}$$

$\rightarrow^* \langle \mathit{env}_3, \mathit{fold}\ f\ a\ l, \mathit{stack}_1, \mathit{store}\rangle$ where $\mathit{stack}_1 \overset{\text{def}}{=} (\mathit{env}_3, f\ [\cdot]\ y) :: \mathit{stack}$ (arguments are evaluated from right to left)

$\rightarrow^* \langle \mathit{env}_3, \mathit{fold}\ c\ [\,]\ l', \mathit{stack}_1, \mathit{store}\rangle$

$\rightarrow^* \langle \mathit{env}_4, \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t''), \mathit{stack}_1, \mathit{store}_1\rangle$ by induction hypothesis

where $t'' \overset{\text{def}}{=} \mathit{filter}(E, M, (x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]), t')$

$\rightarrow \langle \mathit{env}_3, f\ (\mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t''))\ y, \mathit{stack}, \mathit{store}_1\rangle$

$\rightarrow^* \langle \mathit{env}_3, c\ (\mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t''))\ y, \mathit{stack}, \mathit{store}_1\rangle$

$\rightarrow^* \langle \mathit{env}_5, \textbf{try}\ (c'\ x) :: a\ \textbf{with Match\_failure} \rightarrow a, \mathit{stack}, \mathit{store}_1\rangle$

where $\mathit{env}_5 \overset{\text{def}}{=} \mathit{env}'[a \mapsto \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t''), x \mapsto y]$

$\rightarrow \langle \mathit{env}_5, (c'\ x) :: a, \mathit{stack}_2, \mathit{store}_1\rangle$

where $\mathit{stack}_2 \overset{\text{def}}{=} (\mathit{env}_5, \textbf{try}\ [\cdot]\ \textbf{with Match\_failure} \rightarrow a) :: \mathit{stack}$

$\rightarrow \langle \mathit{env}_5, c'\ x, \mathit{stack}_3, \mathit{store}_1\rangle$ where $\mathit{stack}_3 \overset{\text{def}}{=} (\mathit{env}_5, [\cdot] :: a) :: \mathit{stack}_2$

$\rightarrow^* \langle \mathit{env}_5[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto d_1, \dots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto d_k],$
$\textbf{let}\ \mathbb{G}_{\mathsf{var}}(x_1) = \mathbb{G}_{\mathsf{deser}}(T_{x_1})\ \mathbb{G}_{\mathsf{var}}(x_1)\ \textbf{in} \dots$
$\textbf{let}\ \mathbb{G}_{\mathsf{var}}(x_k) = \mathbb{G}_{\mathsf{deser}}(T_{x_k})\ \mathbb{G}_{\mathsf{var}}(x_k)\ \textbf{in}$
$\textbf{if}\ (\mathbb{G}_{\mathsf{M}}(M))\ \textbf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k))\ \textbf{else raise Match\_failure},$
$\mathit{stack}_3, \mathit{store}_1\rangle$

$\rightarrow^* \langle \mathit{env}_6,$
$\textbf{if}\ (\mathbb{G}_{\mathsf{M}}(M))\ \textbf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k))\ \textbf{else raise Match\_failure},$
$\mathit{stack}_3, \mathit{store}_2\rangle$ where $\mathit{env}_6 \overset{\text{def}}{=} \mathit{env}_5[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto d'_1, \dots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto d'_k]$
by Proposition 8.5 applied k times

$\rightarrow \mathit{th}_1 \overset{\text{def}}{=} \langle \mathit{env}_6, \mathbb{G}_{\mathsf{M}}(M), \mathit{stack}_4, \mathit{store}_2\rangle$

where $\mathit{stack}_4 \overset{\text{def}}{=} (\mathit{env}_6, \textbf{if}\ [\cdot]\ \textbf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k))$
$\textbf{else raise Match\_failure}) :: \mathit{stack}_3$

The environment env_6 contains $\mathit{env}_{\mathsf{prim}}$ and $\mathit{env}(E[x_1[\widetilde{a}] \mapsto a_1, \dots, x_k[\widetilde{a}] \mapsto a_k], M)$. Let r be the CryptoVerif value such that $E[x_1[\widetilde{a}] \mapsto a_1, \dots, x_k[\widetilde{a}] \mapsto a_k] \cdot M \Downarrow r$. So by Lemma 8.8,

$$\mathit{th}_1 \rightarrow^* \mathit{th}_2 \overset{\text{def}}{=} \langle \mathit{env}_7, \mathbb{G}_{\mathsf{val}\,\mathit{bool}}(r), \mathit{stack}_4, \mathit{store}_3\rangle$$

and by Lemma 8.8, Proposition 8.5, and the induction hypothesis, we have $\mathit{store}_3 \supseteq \mathit{store}_2 \supseteq \mathit{store}_1 \supseteq \mathit{store}$.

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

$\mathit{th}_2 = \langle \mathit{env}_7, \textbf{true}, \mathit{stack}_4, \mathit{store}_3\rangle$

$\rightarrow^* \langle \mathit{env}_6, \textbf{if true then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k))\ \textbf{else raise Match\_failure},$
$\mathit{stack}_3, \mathit{store}_3\rangle$

$$\begin{aligned}
&\to \langle \mathit{env}_6, (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k)), \mathit{stack}_3, \mathit{store}_3\rangle\\
&\to^* \langle \mathit{env}_6, (d'_1, \dots, d'_k), \mathit{stack}_3, \mathit{store}_3\rangle\\
&\to \langle \mathit{env}_5, (d'_1, \dots, d'_k) :: a, \mathit{stack}_2, \mathit{store}_3\rangle\\
&\to^* \langle \mathit{env}_5, \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t'''), \mathit{stack}_2, \mathit{store}_3\rangle\\
&\quad \text{since } \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t''') = (d'_1, \dots, d'_k) :: (\mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t''))\\
&\to^* \langle \mathit{env}_5, \textbf{try } \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t''') \textbf{ with Match\_failure} \to a, \mathit{stack}, \mathit{store}_3\rangle\\
&\to \mathit{th}' \overset{\text{def}}{=} \langle \mathit{env}_5, \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t'''), \mathit{stack}, \mathit{store}_3\rangle
\end{aligned}$$

So the lemma is correct in this case.

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

$$\begin{aligned}
\mathit{th}_2 &= \langle \mathit{env}_7, \textbf{false}, \mathit{stack}_4, \mathit{store}_3\rangle\\
&\to^* \langle \mathit{env}_6, \textbf{if false then } (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k)) \textbf{ else raise Match\_failure},\\
&\qquad \mathit{stack}_3, \mathit{store}_3\rangle\\
&\to \langle \mathit{env}_6, \textbf{raise Match\_failure}, \mathit{stack}_3, \mathit{store}_3\rangle\\
&\to^* \langle \mathit{env}_5, \textbf{try raise Match\_failure with Match\_failure} \to a,\\
&\qquad \mathit{stack}, \mathit{store}_3\rangle\\
&\to^* \langle \mathit{env}_5, a, \mathit{stack}, \mathit{store}_3\rangle\\
&\to \mathit{th}' \overset{\text{def}}{=} \langle \mathit{env}_5, \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, t''), \mathit{stack}, \mathit{store}_3\rangle
\end{aligned}$$

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

Proof (of Lemma 8.10) 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:

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

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

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

  $$\begin{aligned}
  \mathbb{C} &= \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \textbf{let } \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{random}}(T)\ () \textbf{ in } (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P')),\\
  &\qquad \mathit{stack}, \mathit{store}\rangle]\\
  &\to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathbb{G}_{\mathsf{random}}(T)\ (), \mathit{stack}', \mathit{store}\rangle]\\
  &\qquad \text{where } \mathit{stack}' \overset{\text{def}}{=} (\mathit{env}, \textbf{let } \mathbb{G}_{\mathsf{var}}(x) = [\cdot] \textbf{ in } (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P')))\\
  &\qquad\qquad :: \mathit{stack}\\
  &\to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathit{env}_{\mathsf{prim}}(\mathbb{G}_{\mathsf{random}}(T))\ (), \mathit{stack}', \mathit{store}\rangle]\\
  &\to^*_{1/|T|} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}', \mathbb{G}_{\mathsf{val}T}(b), \mathit{stack}', \mathit{store}'\rangle] \qquad \text{by Proposition 8.5}
  \end{aligned}$$

$$\to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'), \mathit{stack}, \mathit{store}'\rangle]$$
$$\text{where } \mathit{env}'' \overset{\text{def}}{=} \mathit{env}[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)]$$
$$\to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]), ([\cdot]; \mathbb{G}(P')) :: \mathit{stack}, \mathit{store}'\rangle]$$
$$\to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', (); \mathbb{G}(P'), \mathit{stack}, \mathit{store}''\rangle, \mathsf{globalstore} \mapsto \mathit{globalstore}'] \quad \text{by Lemma D.1}$$
$$\to \mathbb{C}' = \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', \mathbb{G}(P'), \mathit{stack}, \mathit{store}''\rangle, \mathsf{globalstore} \mapsto \mathit{globalstore}']$$

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

Let $E' \overset{\text{def}}{=} E[x[\widetilde{a}] \mapsto b]$. We have $\mathit{env}(E', P') = \mathit{env}(E, P)[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(a)]$, so $\mathit{env}'' \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E', P')$. By Lemma D.1, we have $\mathit{globalstore}' \supseteq \mathit{globalstore}(E', \mathcal{T})$ and $\mathit{globalstore}'(l) = \mathbb{C}_{\mathit{globalstore}}(\mathbb{C})(l)$ for all $l \notin \mathit{Loc}_{\mathsf{priv}}$. By Lemma D.1 and Proposition 8.5, we have $\mathit{store}'' \supseteq \mathit{store}' \supseteq \mathit{store}$. Events are unchanged on both sides. So this construct satisfies the lemma.

- The assignment construct:

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

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

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

  On the OCaml side, we have:

$$\mathbb{C} = \mathbb{C}[\mathsf{th} \mapsto \mathit{th}] \text{ where } \mathit{th} \overset{\text{def}}{=}$$
$$\langle \mathit{env}, \textbf{let } \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{M}}(M) \textbf{ in } (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P')), \mathit{stack}, \mathit{store}\rangle$$
$$\to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathbb{G}_{\mathsf{M}}(M), \mathit{stack}', \mathit{store}\rangle]$$
$$\text{where } \mathit{stack}' \overset{\text{def}}{=} (\mathit{env}, \textbf{let } \mathbb{G}_{\mathsf{var}}(x) = [\cdot] \textbf{ in } (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'))) :: \mathit{stack}$$
$$\to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathbb{G}_{\mathsf{val}T}(b), \mathit{stack}', \mathit{store}'\rangle] \quad \text{by Lemma 8.8}$$
$$\to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'), \mathit{stack}, \mathit{store}'\rangle]$$
$$\text{where } \mathit{env}' \overset{\text{def}}{=} \mathit{env}[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)]$$
$$\to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}', (); \mathbb{G}(P'), \mathit{stack}, \mathit{store}''\rangle, \mathsf{globalstore} \mapsto \mathit{globalstore}'] \quad \text{by Lemma D.1}$$
$$\to \mathbb{C}' \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}', \mathbb{G}(P'), \mathit{stack}, \mathit{store}''\rangle, \mathsf{globalstore} \mapsto \mathit{globalstore}']$$

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

  Let $E' \overset{\text{def}}{=} E[x[\widetilde{a}] \mapsto b]$. We have $\mathit{env}(E', P') = \mathit{env}(E, P)[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)]$, so $\mathit{env}' \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E', P')$. By Lemma D.1, we have

$\mathit{globalstore}' \supseteq \mathit{globalstore}(E', \mathcal{T})$ and $\mathit{globalstore}'(l) = \mathbb{C}_{\mathit{globalstore}}(\mathbb{C})(l)$ for all $l \notin \mathit{Loc}_{\mathsf{priv}}$. By Lemmas D.1 and 8.8, we have $\mathit{store}'' \supseteq \mathit{store}' \supseteq \mathit{store}$. Events are unchanged on both sides. So this construct satisfies the lemma.

- The conditional construct:

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

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

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

$$\begin{aligned}
\mathit{th} &= \langle \mathit{env}, \textbf{if}\ \mathbb{G}_{\mathsf{M}}(M)\ \textbf{then}\ \mathbb{G}(P_1)\ \textbf{else}\ \mathbb{G}(P_2), \mathit{stack}, \mathit{store}\rangle \\
&\to \langle \mathit{env}, \mathbb{G}_{\mathsf{M}}(M), \mathit{stack}', \mathit{store}\rangle \\
&\qquad \text{where } \mathit{stack}' \stackrel{\text{def}}{=} (\mathit{env}, \textbf{if}\ [\cdot]\ \textbf{then}\ \mathbb{G}(P_1)\ \textbf{else}\ \mathbb{G}(P_2)) :: \mathit{stack} \\
&\to^* \langle \mathit{env}, \textbf{true}, \mathit{stack}', \mathit{store}'\rangle \qquad \text{by Lemma 8.8} \\
&\to \langle \mathit{env}, \textbf{if true then}\ \mathbb{G}(P_1)\ \textbf{else}\ \mathbb{G}(P_2), \mathit{stack}, \mathit{store}'\rangle \\
\to \mathit{th}' &\stackrel{\text{def}}{=} \langle \mathit{env}, \mathbb{G}(P_1), \mathit{stack}, \mathit{store}'\rangle
\end{aligned}$$

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

  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 8.8, we have $\mathit{store}' \supseteq \mathit{store}$, so this construct satisfies the lemma.

- The insert construct:

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

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

  where $\mathcal{T}' \stackrel{\text{def}}{=} \mathcal{T}[\mathit{Tbl} \mapsto (a_1, \dots, a_k) :: \mathcal{T}(\mathit{Tbl})]$. 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 $(\mathit{Tbl}, f) \in \mathit{Tables}$. Let $\mathit{globalstore} = \mathbb{C}_{\mathit{globalstore}}(\mathbb{C})$. Since $\mathit{globalstore} \supseteq \mathit{globalstore}(E, \mathcal{T})$, we have

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

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

$$\begin{aligned}
&\mathbb{C} = \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, f := e :: (!f); \mathbb{G}(P'), \mathit{stack}, \mathit{store}\rangle]\\
&\qquad \text{where } e \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{ser}}(T_1)\ \mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{ser}}(T_k)\ \mathbb{G}_{\mathsf{M}}(M_k))\\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, e :: t, \mathit{stack}', \mathit{store}\rangle]\\
&\qquad \text{where } \mathit{stack}' \stackrel{\text{def}}{=} (\mathit{env}, f := [\cdot]) :: (\mathit{env}, [\cdot]; \mathbb{G}(P')) :: \mathit{stack}\\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathbb{G}_{\mathsf{tblel}}(\mathit{Tbl}, (a_1, \dots, a_k)) :: t, \mathit{stack}', \mathit{store}'\rangle]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by Lemma 8.8 and Proposition 8.5}\\
&\quad \to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, f := t', (\mathit{env}, [\cdot]; \mathbb{G}(P')) :: \mathit{stack}, \mathit{store}'\rangle]\\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, (); \mathbb{G}(P'), \mathit{stack}, \mathit{store}'\rangle, \mathsf{globalstore} \mapsto \mathit{globalstore}']\\
&\to \mathbb{C}' \stackrel{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathbb{G}(P'), \mathit{stack}, \mathit{store}'\rangle, \mathsf{globalstore} \mapsto \mathit{globalstore}']
\end{aligned}$$

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

The global store is modified so that $\mathit{globalstore}' \supseteq \mathit{globalstore}(E, \mathcal{T}')$ and $\mathit{globalstore}'(l) = \mathbb{C}_{\mathit{globalstore}}(\mathbb{C})(l)$ for all $l \notin \mathit{Loc}_{\mathsf{priv}}$, and the environments and events are unchanged on both sides. Moreover, by Proposition 8.5 and Lemma 8.8, we have $\mathit{store}' \supseteq \mathit{store}$, so this construct satisfies the lemma.

- The get construct:

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

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

  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 $l' \stackrel{\text{def}}{=} \mathit{filter}(E, M, (x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]), \mathcal{T}(\mathit{Tbl})) = [b_1, \dots, b_m]$. This list contains every element of $\mathcal{T}(\mathit{Tbl})$ such that M is true.

  If l' is empty, then:

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

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

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

  with $p_b \stackrel{\text{def}}{=} \sum_{\{j \in \{1,\dots,m\} \mid b_j = b\}} \mathit{almostunif}(\{1, \dots, m\}, j)$ and $E_b \stackrel{\text{def}}{=} E[x_1[\widetilde{a}] \mapsto a_1, \dots, x_k[\widetilde{a}] \mapsto a_k]$.

On the OCaml side, let us denote

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

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

$$\begin{aligned}
&\mathbb{C} = \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \textbf{let } l = e_2 \textbf{ in } e, \mathit{stack}, \mathit{store}\rangle]\\
&\qquad \text{where } e_2 \overset{\text{def}}{=} \mathit{read\_table}(f, \mathbb{G}_{\mathsf{test}}((x_1, \dots, x_k), M))\\
&\qquad\qquad = \textbf{let rec } \mathit{fold} = \textbf{function } \mathbb{G}_{\mathsf{fold}} \textbf{ in}\\
&\qquad\qquad\qquad \mathit{fold}\ (\textbf{function } a \to\ x \to\ e_1)\ [\,]\ !f\\
&\quad \to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, e_2, \mathit{stack}', \mathit{store}\rangle]\\
&\qquad \text{where } \mathit{stack}'' \overset{\text{def}}{=} (\mathit{env}, \textbf{let } l = [\cdot] \textbf{ in } e) :: \mathit{stack}\\
&\quad \to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}', \mathit{fold}\ (\textbf{function } a \to\ x \to\ e_1)\ [\,]\ !f, \mathit{stack}', \mathit{store}\rangle]\\
&\qquad \text{where } \mathit{env}' \overset{\text{def}}{=} \mathit{env}[\mathit{fold} \mapsto \textbf{letrec}[\mathit{env}, \{\mathit{fold} \mapsto \mathbb{G}_{\mathsf{fold}}\} \textbf{ in } \mathit{fold}]]\\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}', e_3, \mathit{stack}', \mathit{store}\rangle]\\
&\qquad \text{where } e_3 \overset{\text{def}}{=}\\
&\qquad\quad \mathit{fold}\ \textbf{function}[\mathit{env}', a \to\ \textbf{function } x \to\ e_1]\ [\,]\ \mathbb{G}_{\mathsf{tbl}}(\mathit{Tbl}, \mathcal{T}(\mathit{Tbl}))\\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, l'), \mathit{stack}', \mathit{store}'\rangle] \qquad \text{by Lemma D.4}\\
&\qquad \text{since } l' = \mathit{filter}(E, M, (x_1[\widetilde{a}], \dots, x_k[\widetilde{a}]), \mathcal{T}(\mathit{Tbl}))\\
&\quad \to \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \textbf{let } l = \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, l') \textbf{ in } e, \mathit{stack}, \mathit{store}'\rangle]\\
&\to \mathbb{C}_1 \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', e, \mathit{stack}, \mathit{store}'\rangle]\\
&\qquad \text{where } \mathit{env}'' \overset{\text{def}}{=} \mathit{env}[l \mapsto \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, l')]
\end{aligned}$$

Now, if l' is empty, then $\mathit{env}''(l) = [\,]$, so

$$\mathbb{C}_1 \to^* \mathbb{C}' \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', \mathbb{G}(P''), \mathit{stack}, \mathit{store}'\rangle]$$

The sequence of reductions $\mathbb{C} \to^* \mathbb{C}_1 \to^* \mathbb{C}'$ describes the set of traces $\mathbb{CTS}_1$ that corresponds to the unique CryptoVerif reduction that can happen when l' is empty. We have $\Pr[\mathbb{CTS}_1] = 1$.

The CryptoVerif environment E is unchanged and the OCaml environment env'' is an extension of env, so we have $\mathit{env}'' \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E, P'')$. The CryptoVerif tables, the global store, and the events on both sides are unchanged. By Lemma D.4, we have $\mathit{store}' \supseteq \mathit{store}$. So, in this case, the get construct satisfies the lemma.

If l' is not empty, then let $b = (a_1, \dots, a_k)$ be any element of l', and let $v = \mathbb{G}_{\mathsf{val}T_1,\dots,T_k}(\mathit{Tbl}, b) = (\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_k}(a_k))$. We have $\mathit{env}''(l) =$

$\mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, l')$. Let $\mathit{env}''(l) = [v_1; \dots; v_m]$. The set $S \overset{\text{def}}{=} \{j \in \{1, \dots, m\} \mid v = v_j\}$ is equal to the set $\{j \in \{1, \dots, m\} \mid b = b_j\}$, because the function $b \mapsto \mathbb{G}_{\mathsf{val}T_1,\dots,T_k}(\mathit{Tbl}, b)$ is injective. Hence, we have $p_b = \sum_{j \in S} \mathit{almostunif}(\{1, \dots, m\}, j)$.

$$\begin{aligned}
&\mathbb{C}_1 \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', e_4, \mathit{stack}, \mathit{store}' \rangle] \\
&\qquad \text{where } e_4 \overset{\text{def}}{=} \textbf{let } (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k)) = \textbf{random}_\ell\ l \textbf{ in} \\
&\qquad\qquad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \dots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P')) \\
&\quad \to^*_{p_b} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}'', e_5, \mathit{stack}, \mathit{store}'' \rangle] \qquad \text{by Proposition 8.5} \\
&\qquad \text{where } e_5 \overset{\text{def}}{=} \textbf{let } (\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k)) = v \textbf{ in} \\
&\qquad\qquad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \dots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P')) \\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}''', \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \dots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P'), \mathit{stack}, \mathit{store}'' \rangle] \\
&\qquad \text{where } \mathit{env}''' \overset{\text{def}}{=} \\
&\qquad\quad \mathit{env}''[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_k}(a_k)] \\
&\to^* \mathbb{C}'_b \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}''', \mathbb{G}(P'), \mathit{stack}, \mathit{store}''' \rangle, \mathsf{globalstore} \mapsto \mathit{globalstore}'] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{by Lemma D.1}
\end{aligned}$$

The sequence of reductions $\mathbb{C} \to^* \mathbb{C}_1 \to^* \mathbb{C}'_b$ describes the set of traces $\mathbb{CTS}_b$ that corresponds to the CryptoVerif reduction in which the element $b = (a_1, \dots, a_k)$ of l' is chosen. We have $\Pr[\mathbb{CTS}_b] = p_b$.

By Lemma D.1, $\mathit{globalstore}' \supseteq \mathit{globalstore}(E_b, \mathcal{T})$, and $\mathit{globalstore}'$ and $\mathbb{C}_{\mathit{globalstore}}(\mathbb{C})$ are equal on all locations not in $\mathit{Loc}_{\mathsf{priv}}$, since $E_b = E[x_1[\widetilde{a}] \mapsto a_1, \dots, x_k[\widetilde{a}] \mapsto a_k]$. Since the OCaml environment is $\mathit{env}''' = \mathit{env}[l \mapsto \dots, \mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_k}(a_k)]$, we have $\mathit{env}''' \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_b, P')$. The events are unchanged on both sides. By Lemma D.1, Proposition 8.5, and Lemma D.4, we have $\mathit{store}''' \supseteq \mathit{store}'' \supseteq \mathit{store}' \supseteq \mathit{store}$. So, in this case, the get construct also satisfies the lemma.

- The event construct:

  On the CryptoVerif side, let us suppose that $E \cdot M_j \Downarrow a_j$ for all $j \leq l$. We have the following reduction:

  $$E, \mathsf{event}\ ev(M_1, \dots, M_l); P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \to E, P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}',$$

  where $\mathcal{E}' \overset{\text{def}}{=} ev(a_1, \dots, a_l) :: \mathcal{E}$. Let us denote $T_1 \times \dots \times T_l$ the type of the event ev.

  On the OCaml side, let us denote

  $$\mathit{events}' \overset{\text{def}}{=} \mathbb{G}_{\mathsf{ev}}(\mathcal{E}') = ev(\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_l}(a_l)) :: \mathit{events}.$$

We have

$$\begin{aligned}
\mathbb{C} &= \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, e; \mathbb{G}(P'), \mathit{stack}, \mathit{store}\rangle] \\
&\qquad \text{where } e \stackrel{\text{def}}{=} \textbf{event } \mathit{ev}(\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_l)) \\
&\quad \rightarrow \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, e, \mathit{stack}', \mathit{store}, \rangle] \\
&\qquad \text{where } \mathit{stack}' \stackrel{\text{def}}{=} (\mathit{env}, [\cdot]; \mathbb{G}(P')) :: \mathit{stack} \\
&\quad \rightarrow^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, e', \mathit{stack}', \mathit{store}'\rangle] \qquad\qquad \text{by Lemma 8.8} \\
&\qquad \text{where } e' \stackrel{\text{def}}{=} \textbf{event } \mathit{ev}(\mathbb{G}_{\mathsf{val}T_1}(a_1), \dots, \mathbb{G}_{\mathsf{val}T_l}(a_l)) \\
&\quad \rightarrow \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, (), \mathit{stack}', \mathit{store}'\rangle, \mathsf{events} \mapsto \mathit{events}'] \\
\rightarrow^* \mathbb{C}' &\stackrel{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}, \mathbb{G}(P'), \mathit{stack}, \mathit{store}'\rangle, \mathsf{events} \mapsto \mathit{events}']
\end{aligned}$$

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

The CryptoVerif environment E and tables $\mathcal{T}$ and the OCaml environment *env* and global store *globalstore* are unchanged. We have $\mathit{events}' = \mathbb{G}_{\mathsf{ev}}(\mathcal{E}')$. By Lemma 8.8, we have $\mathit{store}' \supseteq \mathit{store}$, so this construct satisfies the lemma. □

E Proof of Lemma 8.18 and Proposition 8.19

Proof (of Lemma 8.18) Let us first show by induction on *steps* that, if $\mathbb{CS} \rightarrow^* \mathbb{CS}'$ in at most *steps* steps and $\mathbb{CS}'$ does not reduce, or $\mathbb{CS} \rightarrow^* \mathbb{CS}'$ in exactly *steps* steps, $\mathcal{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$, and $P = x[a] \leftarrow \text{simulate}_{\mathsf{ML}}(s[a]); P'$, then $\mathcal{C}, \mathit{steps}, \mathbb{CS} \rightsquigarrow^* E[x[a] \mapsto \mathit{simreturn}(\mathbb{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$.

- If $\mathit{steps} = 0$ or $\mathbb{CS}$ does not reduce, then $\mathbb{CS}' = \mathbb{CS}$ and $\mathcal{C}, \mathit{steps}, \mathbb{CS} \rightsquigarrow E[x[a] \mapsto \mathit{simreturn}(\mathbb{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$ by (Leave Simulator).
- If $\mathit{steps} > 0$ and $\mathbb{CS} \rightarrow \mathbb{CS}_1$, then $\mathbb{CS}_1 \rightarrow^* \mathbb{CS}'$ in at most $\mathit{steps}-1$ steps and $\mathbb{CS}'$ does not reduce, or $\mathbb{CS}_1 \rightarrow^* \mathbb{CS}'$ in exactly $\mathit{steps}-1$ steps, so by induction hypothesis, $\mathcal{C}, \mathit{steps} - 1, \mathbb{CS}_1 \rightsquigarrow^* E[x[a] \mapsto \mathit{simreturn}(\mathbb{CS})], P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$. By (Simulator), $\mathcal{C}, \mathit{steps}, \mathbb{CS} \rightsquigarrow \mathcal{C}, \mathit{steps} - 1, \mathbb{CS}_1$, so $\mathcal{C}, \mathit{steps}, \mathbb{CS} \rightsquigarrow^* E[x[a] \mapsto \mathit{simreturn}(\mathbb{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$.

Let us now prove that, if $\mathcal{C} \rightarrow_p \mathcal{C}'$, then there is a trace $\mathcal{C} \rightsquigarrow_p^* \mathcal{C}'$ and all intermediate configurations in this trace (if any) are of the form $\mathcal{C}, \mathit{steps}, \mathbb{CS}$. Let $\mathcal{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$.

- If P is not of the form $x[a] \leftarrow \text{simulate}_{\mathsf{ML}}(s[a]); P'$ for any x, a, P', then $\mathcal{C} \rightsquigarrow_p \mathcal{C}'$ by (CryptoVerif).
- If $P = x[a] \leftarrow \text{simulate}_{\mathsf{ML}}(s[a]); P'$ for some x, a, P', then by the semantics of CryptoVerif, $s[a] \in \mathit{Dom}(E)$, $E(s[a])$ is of type $T_{\mathbb{CS}}$, and $\mathcal{C}' = E[x[a] \mapsto \text{simulate}_{\mathsf{ML}}(E(s[a]))], P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$. Since $E(s[a])$ is of

type $T_{\mathbb{CS}}$, there exists a configuration $\mathbb{CS}$ such that $E(s[a]) = \mathit{repr}(\mathbb{CS})$. By reduction rule (Enter Simulator), $\mathcal{C} \rightsquigarrow \mathcal{C}_1^{\mathsf{cs}} = \mathcal{C}, N_{\mathsf{steps}}, \mathbb{CS}$. Moreover, by definition of $\mathrm{simulate}_{\mathsf{ML}}$, $\mathbb{CS} \rightarrow^* \mathbb{CS}'$ in at most N_{steps} steps and $\mathbb{CS}'$ does not reduce, or $\mathbb{CS} \rightarrow^* \mathbb{CS}'$ in exactly N_{steps} steps, and $\mathrm{simulate}_{\mathsf{ML}}(\mathit{repr}(\mathbb{CS})) = \mathit{simreturn}(\mathbb{CS}')$. By the result shown above, $\mathcal{C} \rightsquigarrow \mathcal{C}_1^{\mathsf{cs}} \rightsquigarrow^* E[x[a] \mapsto \mathit{simreturn}(\mathbb{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} = \mathcal{C}'$.

Finally, let us show that, if $\mathcal{C}$ does not reduce by $\rightarrow$, then it does not reduce by $\rightsquigarrow$ either. Let $\mathcal{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}$.

- If P is not of the form $x[a] \leftarrow \mathrm{simulate}_{\mathsf{ML}}(s[a]); P'$ for any x, a, P', then the only rule applicable to reduce $\mathcal{C}$ by $\rightsquigarrow$ is (CryptoVerif), and it cannot be applied because $\mathcal{C}$ does not reduce by $\rightarrow$. Hence $\mathcal{C}$ does not reduce by $\rightsquigarrow$.

- If $P = x[a] \leftarrow \mathrm{simulate}_{\mathsf{ML}}(s[a]); P'$ for some x, a, P', then either $s[a] \notin \mathit{Dom}(E)$ or $E(s[a]) \notin T_{\mathbb{CS}}$. The only rule applicable to reduce $\mathcal{C}$ by $\rightsquigarrow$ is (Enter Simulator), and it does not apply when $s[a] \notin \mathit{Dom}(E)$ or $E(s[a]) \notin T_{\mathbb{CS}}$. Hence $\mathcal{C}$ does not reduce by $\rightsquigarrow$. (We could also show that, because the CryptoVerif configurations are well-typed, $\mathcal{C}$ always reduces when $P = x[a] \leftarrow \mathrm{simulate}_{\mathsf{ML}}(s[a]); P'$.) $\square$

Proof (of Proposition 8.19) For $b \in \{\mathrm{true}, \mathrm{false}\}$, let $\mathcal{CTS}_b$ be the set of complete CryptoVerif traces using $\rightarrow$ starting at $\mathcal{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = b$. By the first property of Lemma 8.18, we can map each trace $\mathcal{CT} \in \mathcal{CTS}_b$ into a trace $\mathcal{CT}^{\mathsf{cs}}$ using $\rightsquigarrow$ and starting at $\mathcal{C}$, such that the configurations of the form $\mathcal{C}$ of $\mathcal{CT}^{\mathsf{cs}}$ are exactly the same as in $\mathcal{CT}$ and $\Pr[\mathcal{CT}^{\mathsf{cs}}] = \Pr[\mathcal{CT}]$.

Let $\mathcal{CTS}_b^{\mathsf{cs}}$ be the set of these traces $\mathcal{CT}^{\mathsf{cs}}$. Let us show that $\mathcal{CTS}_b^{\mathsf{cs}}$ is the set of complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathcal{C}$ 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 $\mathcal{CT}^{\mathsf{cs}}$ is the same as in $\mathcal{CT}$, so it satisfies $D(\mathcal{E}) = b$. By the second property of Lemma 8.18, since $\mathcal{CT}$ is complete, $\mathcal{CT}^{\mathsf{cs}}$ is also complete. Since the mapping from $\mathcal{CT}$ to $\mathcal{CT}^{\mathsf{cs}}$ is injective, we have $\Pr[\mathcal{CTS}_b^{\mathsf{cs}}] = \Pr[\mathcal{CTS}_b]$.

Moreover, if a configuration $\mathcal{C}^{\mathsf{cs}}$ reduces by $\rightsquigarrow$ into another configuration, then the sum of the probabilities of all the possible reductions from $\mathcal{C}^{\mathsf{cs}}$ is 1:

$$\sum_{\{\mathcal{C}^{\mathsf{cs}\prime} \mid \mathcal{C}^{\mathsf{cs}} \rightsquigarrow_{p(\mathcal{C}^{\mathsf{cs}\prime})} \mathcal{C}^{\mathsf{cs}\prime}\}} p(\mathcal{C}^{\mathsf{cs}\prime}) = 1\,.$$

Indeed, the rules that define $\rightsquigarrow$ are mutually exclusive. If $\mathcal{C}^{\mathsf{cs}}$ 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 $\mathcal{C}$ is 1, so $\Pr[\mathcal{CTS}_{\mathrm{true}}] + \Pr[\mathcal{CTS}_{\mathrm{false}}] = 1$. So $\Pr[\mathcal{CTS}_{\mathrm{true}}^{\mathsf{cs}}] +$

$\Pr[\mathcal{CTS}^{\mathsf{cs}}_{\mathrm{false}}] = 1$. 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 $\mathcal{C}$ is 1, so all these traces are in $\mathcal{CTS}^{\mathsf{cs}}_{\mathrm{true}}$ or $\mathcal{CTS}^{\mathsf{cs}}_{\mathrm{false}}$. Hence all complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathcal{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = b$ are in $\mathcal{CTS}^{\mathsf{cs}}_{b}$.

So $\Pr[\mathcal{C} :^{(\rightsquigarrow)} D] = \Pr[\mathcal{CTS}^{\mathsf{cs}}_{\mathrm{true}}] = \Pr[\mathcal{CTS}_{\mathrm{true}}] = \Pr[\mathcal{C} :^{(\mathsf{CV})} D]$. □

F Proof of Lemmas 8.34 and 8.35

Proof (of Lemma 8.34) Let $\mathcal{C}^{\mathsf{cs}}_0 \stackrel{\text{def}}{=} \mathcal{C}^0(Q_0, \mathit{program}_0)$.

$$\begin{aligned}
\mathcal{C}^{\mathsf{cs}}_0 = {} & \emptyset, \mathsf{let}\ x[\,] : \mathit{bitstring} = \mathrm{O}_{\mathsf{start}}()\ \mathsf{in}\ \mathsf{return}(x[\,])\ \mathsf{else}\ \mathsf{end}, \mathcal{T}_0, \mathcal{Q}_0, \emptyset, [\,] \\
& \quad \text{where } \mathcal{Q}_0 \stackrel{\text{def}}{=} \{Q_{\mathsf{start}}(Q_0, \mathit{program}_0)\} \cup \bigcup_{a \le N_{\mathsf{rand+calls}}} \{Q_{\mathsf{loop}}\{a/i'\}\} \\
& \qquad\qquad \cup\ \mathit{oracledefset}(Q_0) \\
& \rightsquigarrow \emptyset, P_1, \mathcal{T}_0, \mathcal{Q}_1, \mathcal{S}_1, [\,] \\
& \quad \text{where } \mathcal{Q}_1 \stackrel{\text{def}}{=} \mathcal{Q}_0 \setminus \{Q_{\mathsf{start}}(Q_0, \mathit{program}_0)\}, \\
& \qquad P_1 \stackrel{\text{def}}{=} s_0[\,] \leftarrow s_0(Q_0, \mathit{program}_0); P_2, \\
& \qquad P_2 \stackrel{\text{def}}{=} \mathsf{let}\ r[\,] : T_{\mathbb{CS}} = \mathsf{loop}\ \mathrm{O}_{\mathsf{loop}}[1](s_0)\ \mathsf{in}\ \mathsf{end}\ \mathsf{else}\ \mathsf{end}, \\
& \qquad \mathcal{S}_1 \stackrel{\text{def}}{=} [x[\,], \mathsf{return}(x[\,]), \mathsf{end}] \\
& \rightsquigarrow E_1, P_2, \mathcal{T}_0, \mathcal{Q}_1, \mathcal{S}_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{S}_1, [\,] \\
& \quad \text{where } P_3 \stackrel{\text{def}}{=} \mathsf{let}\ (r'_{1,r}[\,] : T_{\mathbb{CS}}, b_{1,r}[\,] : \mathit{bool}) = \mathrm{O}_{\mathsf{loop}}[1](s_0) \\
& \qquad\qquad \mathsf{in}\ P_{\mathsf{return\text{-}loop}}(1)\ \mathsf{else}\ \mathsf{end} \\
& \rightsquigarrow E_2, P_{\mathsf{loop}}\{1/i'\}, \mathcal{T}_0, \mathcal{Q}_2, \mathcal{S}_{\mathsf{loop}}(1), [\,] \\
& \quad \text{where } E_2 \stackrel{\text{def}}{=} E_1[s[1] \mapsto s_0(Q_0, \mathit{program}_0)], \\
& \qquad \mathcal{Q}_2 \stackrel{\text{def}}{=} \mathcal{Q}_1 \setminus \{Q_{\mathsf{loop}}\{1/i'\}\}, \\
\rightsquigarrow \mathcal{C}^{\mathsf{cs}}_1 \stackrel{\text{def}}{=} {} & E_2, P_{\mathsf{loop}}\{1/i'\}, \mathcal{T}_0, \mathcal{Q}_2, \mathcal{S}_{\mathsf{loop}}(1), [\,], N_{\mathsf{steps}}, \mathbb{CS}_0 \\
& \quad \text{where } \mathbb{CS}_0 \stackrel{\text{def}}{=} ([\langle \emptyset, \mathit{program}_0, [\,], \emptyset \rangle], \mathit{globalstore}_0, 1), \mathbb{RI}_0, \emptyset
\end{aligned}$$

We have

$$\mathbb{C}_0(Q_0, \mathit{program}_0) = [\langle \emptyset, \mathit{program}_0, [\,], \emptyset \rangle], \mathit{globalstore}_0, 1, \mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q_0), [\,]\,.$$

Let $\mathbb{CT}$ be the trace that consists only of the configuration $\mathbb{C}_0(Q_0, \mathit{program}_0)$. Let us prove that $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}$. Properties I1, I2, and I4 hold. The set $\mathcal{O}(\mathbb{RI}_0)$ contains all the oracles that can be called at the beginning, and

$$\mathcal{Q}_2 = \bigcup_{2 \le a \le N_{\mathsf{rand+calls}}} \{Q_{\mathsf{loop}}\{a/i'\}\} \cup \mathit{oracledefset}(Q_0)\,,$$

so Property I3 holds. As mentioned in Section 5.2.6, the initial program $\mathit{program}_0$ does not contain locations in Loc_ℓ, so Property I5 holds. As also mentioned in Section 5.2.6, $\mathit{program}_0$ contains no closure, and as mentioned in Section 7, $\mathit{program}_0$ contains no tagged function, no return, and no event except in parts $\mathit{program}(\mu_{\mathsf{role}})$ inside **addthread**. So Property T2 holds, which proves Property I6. By Assumption 7.2, Property I7 holds. The global store $\mathit{globalstore}_0$ maps each $l \in \mathit{Loc}_{\mathsf{g}}$ to its initial value $\mathit{initval}_l$ and $\mathit{globalstore}(E_2, \mathcal{T}_0)$ maps each $f \in \mathit{Loc}_{\mathsf{priv}}$ to its initial value $\mathit{initval}_f$ (the empty string "" when $(x[\,], f) \in \mathit{Files}$ and the empty list $[\,]$ when $(\mathit{Tbl}, f) \in \mathit{Tables}$), so Properties I8, I9, and I10 hold. The module set $\mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q_0)$ and the role set $\mathbb{RI}_0$ correspond by definition of $\mathbb{RI}_0$, so Property I11 holds. The event lists are empty on both sides, so Property I12 holds. The sets $\mathcal{O}^\infty(\mathbb{I})$ with $\mathbb{I} \overset{\text{def}}{=} \emptyset$ and $\mathcal{O}_{\mathbf{call}}(\mathbb{CS}_0)$ are both empty, so Property O1 holds. The sets $\mathcal{O}_{\mathbf{call}\text{-}\mathsf{repl}}(th_1^{\mathsf{s}})$, $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(th_1^{\mathsf{s}}))$, and $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(th_1^{\mathsf{s}}))$ where th_1^{s} is the current thread of $\mathbb{CS}_0$ are also empty, so Property O2 holds, which shows Property I13. We have $0 + N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property I14 holds. We have $\alpha = 1$, so Property I15 holds. The set $\mathbb{I}$ is empty, so Property I16 holds. For all $\mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathbb{RI}_0$, we have $a' = 1$, so Property I17 holds. Therefore, $\mathcal{C}_1^{\mathsf{cs}} \equiv \mathbb{CT}$. □

The following two lemmas serve to prove Property I4 of the invariant.

Lemma F.1 *If $E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E} \rightarrow_p E', P', \mathcal{Q}', \mathcal{T}', \mathcal{S}', \mathcal{E}'$, and $\mathit{fv}(P) \cup \mathit{fv}(\mathcal{Q}) \cup \mathit{fv}(\mathcal{S}) \subseteq \mathit{Dom}(E)$, then $\mathit{fv}(P') \cup \mathit{fv}(\mathcal{Q}') \cup \mathit{fv}(\mathcal{S}') \subseteq \mathit{Dom}(E')$.*

Proof This result is easily proved by cases on the applied reduction rule. □

We denote by $\mathcal{C}^{\mathsf{cs}} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathit{rest}$ an extended CryptoVerif configuration in which rest is either nothing or $\mathit{steps}, \mathbb{CS}$.

Lemma F.2 *If $E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathit{rest} \rightsquigarrow_p E', P', \mathcal{Q}', \mathcal{T}', \mathcal{S}', \mathcal{E}', \mathit{rest}'$ and $\mathit{fv}(P) \cup \mathit{fv}(\mathcal{Q}) \cup \mathit{fv}(\mathcal{S}) \subseteq \mathit{Dom}(E)$, then $\mathit{fv}(P') \cup \mathit{fv}(\mathcal{Q}') \cup \mathit{fv}(\mathcal{S}') \subseteq \mathit{Dom}(E')$.*

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 $\mathit{fv}(\mathcal{Q}_0) \subseteq \mathit{Dom}(E)$, $\mathcal{Q}_0 \leftrightarrow \mathbb{RI}, \mathbb{I}$, $\mathcal{Q}_0' \leftrightarrow \mathbb{RI}', \mathbb{I}'$, 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. *$E' \supseteq E$, $\mathbb{I}' = \mathbb{I} - \{O[\widetilde{a}]\}$, $\mathcal{Q}_0' \supseteq \mathcal{Q}_0 \setminus \{\mathcal{Q}_0(O[\widetilde{a}])\}$, $\tau_{\mathsf{O}}' = \tau_{\mathsf{O}}$, and l_{tok}' is a restriction of l_{tok} such that, if $R = O'[\widetilde{a}']$, then $R \in \mathit{Dom}(l_{\mathsf{tok}}')$.*

2. *$E' \supseteq E$, $\mathbb{I}' \supseteq \mathbb{I}$, $\mathcal{Q}_0' \supseteq \mathcal{Q}_0$, $l_{\mathsf{tok}}' \supseteq l_{\mathsf{tok}}$, $\tau_{\mathsf{O}}' \supseteq \tau_{\mathsf{O}}$, if $R = O'[\widetilde{a'}]$, then $O'[\widetilde{a'}] \notin \mathbb{I}' \setminus \mathbb{I}$ and $O'[\widetilde{a'}] \in Dom(l_{\mathsf{tok}})$, and if $R = O'[\_, \widetilde{a''}]$, then for all a, $O'\big[[a, +\infty[, \widetilde{a''}\big] \notin \mathbb{I}' \setminus \mathbb{I}$ and $O'[\_, \widetilde{a''}] \in Dom(\tau_{\mathsf{O}})$.*

Then $correctclosure(R, \mathbb{I}', E', \mathcal{Q}_0', l_{\mathsf{tok}}', \tau_{\mathsf{O}}') \supseteq correctclosure(R, \mathbb{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

Proof Let us consider the first situation.

- Case $R = O[\widetilde{a}]$. Oracle O is not under replication. Since $\mathbb{I} - \{O[\widetilde{a}]\}$ is defined, we have $O[\widetilde{a}] \in \mathbb{I}$, and since $\mathbb{I}' = \mathbb{I} - \{O[\widetilde{a}]\}$, we have $O[\widetilde{a}] \notin \mathbb{I}'$. We also have $l_{\mathsf{tok}}'(O[\widetilde{a}]) = l_{\mathsf{tok}}(O[\widetilde{a}])$. So, by definition of *correctclosure*,

$$\begin{aligned}
&correctclosure(O[\widetilde{a}], \mathbb{I}', E', \mathcal{Q}_0', l_{\mathsf{tok}}', \tau_{\mathsf{O}}')\\
&\quad = \{\mathbf{tagfunction}^{O,\tau}[env, pm_{\mathsf{Once}}(Q)] \mid\\
&\qquad\quad \text{for any } Q, env(token) = l_{\mathsf{tok}}'(O[\widetilde{a}])\}\\
&\quad \supseteq \{\mathbf{tagfunction}^{O,\tau}[env, pm_{\mathsf{Once}}(\mathcal{Q}(O[\widetilde{a}]))] \mid\\
&\qquad\quad env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}(O[\widetilde{a}])), env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\}\\
&\quad \supseteq correctclosure(O[\widetilde{a}], \mathbb{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,.
\end{aligned}$$

- Case $R = O[\_, \widetilde{a''}]$ where $\widetilde{a} = a', \widetilde{a''}$ for some a'. Oracle O is under replication. Since $\mathbb{I} - \{O[\widetilde{a}]\}$ is defined, $O\big[[a', +\infty[, \widetilde{a''}\big] \in \mathbb{I}$, and since $\mathbb{I}' = \mathbb{I} - \{O[\widetilde{a}]\}$, we have $O\big[[a'+1, +\infty[, \widetilde{a''}\big] \in \mathbb{I}'$.

  Suppose that $a' < N_O$. Since $\mathcal{Q}_0 \leftrightarrow \mathbb{RI}, \mathbb{I}$, there exist Q and i such that i does not occur in $fv(Q)$, $\mathcal{Q}_0(O[a', \widetilde{a''}]) = Q\{a'/i\}$, and $\mathcal{Q}_0(O[a'+1, \widetilde{a''}]) = Q\{a'+1/i\}$. It is easy to see that $pm_{\mathsf{Any}}(Q\{a'+1/i\}) = pm_{\mathsf{Any}}(Q\{a'/i\})$, since the translation into OCaml does not depend on the indices. Moreover, $fv(Q\{a'+1/i\}) = fv(Q\{a'/i\})$ since i does not occur in $fv(Q)$, so $env(E, Q\{a'+1/i\}) = env(E, Q\{a'/i\})$. Since $fv(\mathcal{Q}_0) \subseteq Dom(E)$ 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 *correctclosure*,

$$\begin{aligned}
&correctclosure(O[\_, \widetilde{a''}], \mathbb{I}', E', \mathcal{Q}_0', l_{\mathsf{tok}}', \tau_{\mathsf{O}}')\\
&\quad = \{\mathbf{tagfunction}^{O,\tau}[env, pm_{\mathsf{Any}}(\mathcal{Q}_0'(O[a'+1, \widetilde{a''}]))] \mid\\
&\qquad\quad \tau = \tau_{\mathsf{O}}'(O[\_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E', \mathcal{Q}_0'(O[a'+1, \widetilde{a''}]))\}\\
&\quad = \{\mathbf{tagfunction}^{O,\tau}[env, pm_{\mathsf{Any}}(\mathcal{Q}_0(O[a', \widetilde{a''}]))] \mid\\
&\qquad\quad \tau = \tau_{\mathsf{O}}(O[\_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}_0(O[a', \widetilde{a''}]))\}\\
&\quad = correctclosure(O[\_, \widetilde{a''}], \mathbb{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,.
\end{aligned}$$

Suppose that $a' = N_O$. By definition of *correctclosure*,

$$\begin{aligned}
&\mathit{correctclosure}(O[\_,\widetilde{a''}],\mathbb{I}',E',\mathcal{Q}'_0,l'_{\mathsf{tok}},\tau'_{\mathsf{O}})\\
&\quad= \{\mathbf{tagfunction}^{O,\tau}[\mathit{env},\mathit{pm}_{\mathsf{Any}}(Q)] \mid \tau = \tau'_{\mathsf{O}}(O[\_,\widetilde{a''}]), \text{for any } Q, \mathit{env}\}\\
&\quad\supseteq \{\mathbf{tagfunction}^{O,\tau}[\mathit{env},\mathit{pm}_{\mathsf{Any}}(\mathcal{Q}_0(O[a',\widetilde{a''}]))] \mid\\
&\qquad\quad \tau = \tau_{\mathsf{O}}(O[\_,\widetilde{a''}]), \mathit{env} \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E,\mathcal{Q}_0(O[a',\widetilde{a''}]))\}\\
&\quad\supseteq \mathit{correctclosure}(O[\_,\widetilde{a''}],\mathbb{I},E,\mathcal{Q}_0,l_{\mathsf{tok}},\tau_{\mathsf{O}})\,.
\end{aligned}$$

Suppose that $a' > N_O$. We have $\mathit{correctclosure}(O[\_,\widetilde{a''}],\mathbb{I}',E',\mathcal{Q}'_0,l'_{\mathsf{tok}},\tau'_{\mathsf{O}}) = \mathit{correctclosure}(O[\_,\widetilde{a''}],\mathbb{I},E,\mathcal{Q}_0,l_{\mathsf{tok}},\tau_{\mathsf{O}})$ 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 *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 $\mathit{fv}(\mathcal{Q}_0) \subseteq \mathit{Dom}(E)$ and E' is an extension of E, we have $\mathit{env}(E',\mathcal{Q}'_0(O'[\widetilde{a'}])) = \mathit{env}(E',\mathcal{Q}_0(O'[\widetilde{a'}])) = \mathit{env}(E,\mathcal{Q}_0(O'[\widetilde{a'}]))$. Moreover, when $R = O'[\widetilde{a'}]$, we have $l'_{\mathsf{tok}}(O'[\widetilde{a'}]) = l_{\mathsf{tok}}(O'[\widetilde{a'}])$. Hence, by definition of *correctclosure*, $\mathit{correctclosure}(R,\mathbb{I}',E',\mathcal{Q}'_0,l'_{\mathsf{tok}},\tau'_{\mathsf{O}}) = \mathit{correctclosure}(R,\mathbb{I},E,\mathcal{Q}_0,l_{\mathsf{tok}},\tau_{\mathsf{O}})$.

Let us now consider the second situation.

- Case $R = O'[\widetilde{a'}]$. Since $O'[\widetilde{a'}] \notin \mathbb{I}' \setminus \mathbb{I}$, we have $O'[\widetilde{a'}] \in \mathbb{I}'$ if and only if $O'[\widetilde{a'}] \in \mathbb{I}$. We have $l'_{\mathsf{tok}}(O'[\widetilde{a'}]) = l_{\mathsf{tok}}(O'[\widetilde{a'}])$.

  If $O'[\widetilde{a'}] \notin \mathbb{I}$, these points are sufficient to conclude that $\mathit{correctclosure}(R,\mathbb{I}',E',\mathcal{Q}'_0,l'_{\mathsf{tok}},\tau'_{\mathsf{O}}) = \mathit{correctclosure}(R,\mathbb{I},E,\mathcal{Q}_0,l_{\mathsf{tok}},\tau_{\mathsf{O}})$.

  If $O'[\widetilde{a'}] \in \mathbb{I}$, 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 $\mathit{fv}(\mathcal{Q}_0) \subseteq \mathit{Dom}(E)$ and E' is an extension of E, we have $\mathit{env}(E',\mathcal{Q}'_0(O'[\widetilde{a'}])) = \mathit{env}(E',\mathcal{Q}_0(O'[\widetilde{a'}])) = \mathit{env}(E,\mathcal{Q}_0(O'[\widetilde{a'}]))$. So $\mathit{correctclosure}(R,\mathbb{I}',E',\mathcal{Q}'_0,l'_{\mathsf{tok}},\tau'_{\mathsf{O}}) = \mathit{correctclosure}(R,\mathbb{I},E,\mathcal{Q}_0,l_{\mathsf{tok}},\tau_{\mathsf{O}})$.

- Case $R = O'[\_,\widetilde{a''}]$. Since for all a, $O'\big[[a,+\infty[,\widetilde{a''}\big] \notin \mathbb{I}' \setminus \mathbb{I}$, we have $O'\big[[a',+\infty[,\widetilde{a''}\big] \in \mathbb{I}'$ if and only if $O'\big[[a',+\infty[,\widetilde{a''}\big] \in \mathbb{I}$.

  If there is no a' such that $O'\big[[a',+\infty[,\widetilde{a''}\big] \in \mathbb{I}$, this point is sufficient to conclude that $\mathit{correctclosure}(R,\mathbb{I}',E',\mathcal{Q}'_0,l'_{\mathsf{tok}},\tau'_{\mathsf{O}}) = \mathit{correctclosure}(R,\mathbb{I},E,\mathcal{Q}_0,l_{\mathsf{tok}},\tau_{\mathsf{O}})$.

  If $O'\big[[a',+\infty[,\widetilde{a''}\big] \in \mathbb{I}$ 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 $\mathit{fv}(\mathcal{Q}_0) \subseteq \mathit{Dom}(E)$ and E' is an extension of E, we obtain $\mathit{env}(E',\mathcal{Q}'_0(O'[a',\widetilde{a''}])) = \mathit{env}(E',\mathcal{Q}_0(O'[a',\widetilde{a''}])) = \mathit{env}(E,\mathcal{Q}_0(O'[a',\widetilde{a''}]))$. Moreover, we have $\tau'_{\mathsf{O}}(O'[\_,\widetilde{a''}]) = \tau_{\mathsf{O}}(O'[\_,\widetilde{a''}])$. So $\mathit{correctclosure}(R, \mathbb{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathit{correctclosure}(R, \mathbb{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

If $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathbb{I}$ and $a' > N_{O'}$, then we have $\tau'_{\mathsf{O}}(O'[\_,\widetilde{a''}]) = \tau_{\mathsf{O}}(O'[\_,\widetilde{a''}])$, so $\mathit{correctclosure}(R, \mathbb{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathit{correctclosure}(R, \mathbb{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. □

Let P'_{loop} be the process from line 8 to line 20 of Figure 18. Let P^j_{loop} be the process from line 13 to line 15 for the **if** $o = \mathrm{o}_j$ 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 8.35) Let us consider $\mathcal{C}^{\mathsf{cs}}$ and $\mathbb{CT}$ such that $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$. Let $\mathcal{C}^{\mathsf{cs}} = E, P_{\mathsf{loop}}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E}, \mathit{steps}, \mathbb{CS}$ and let $\mathbb{C}$ be the last configuration of $\mathbb{CT}$. Let

$$\begin{aligned}
\mathbb{CS} &= ([\mathit{th}_1, \dots, \mathit{th}_n], \mathit{globalstore}^{\mathsf{s}}, \mathit{tj}), \mathbb{RI}, \mathbb{I}\,,\\
\mathbb{C} &= [\mathit{th}'_1, \dots, \mathit{th}'_n], \mathit{globalstore}^{\mathsf{o}}, \mathit{tj}, \mathbb{MI}, \mathit{events}\,,\\
\mathit{th}_{\mathit{tj}} &= \mathit{th}^{\mathsf{s}} = \langle \mathit{env}^{\mathsf{s}}, \mathit{pe}^{\mathsf{s}}, \mathit{stack}^{\mathsf{s}}, \mathit{store}^{\mathsf{s}} \rangle\,,\\
\mathit{th}'_{\mathit{tj}} &= \mathit{th}^{\mathsf{o}} = \langle \mathit{env}^{\mathsf{o}}, \mathit{pe}^{\mathsf{o}}, \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}} \rangle\,.
\end{aligned}$$

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 T1 or Property T2 is satisfied for the current thread.

Case 1. Property T1 is satisfied for the current thread, that is, we are at the beginning of the initialization of a protocol thread. There exists a program $\mathit{program}'$ such that

$$\begin{aligned}
\mathit{th}^{\mathsf{s}} = \langle \emptyset, &\mathit{program}_{\mathsf{prim}};; \mathit{program}'(\mathsf{role}_1[\widetilde{a_1}]);; \dots;; \mathit{program}'(\mathsf{role}_m[\widetilde{a_m}]);;\\
&\mathit{program}', [\,], \emptyset\rangle\,.
\end{aligned}$$

There is no closure, no tagged function $\mathbf{tagfunction}^t\ \mathit{pm}$, no event, and no return in $\mathit{program}'$, except in $\mathit{program}(\mu_{\mathsf{role}})$ in arguments of **addthread**. The OCaml thread verifies $\mathit{th}^{\mathsf{o}} = \mathit{replaceinitpm}(\mathit{th}^{\mathsf{s}})$, so

$$\mathit{th}^{\mathsf{o}} = \langle \emptyset, \mathit{program}_{\mathsf{prim}};; \mathit{program}(\mu_{\mathsf{role}_1});; \dots;; \mathit{program}(\mu_{\mathsf{role}_m});; \mathit{program}', [\,], \emptyset\rangle\,.$$

By Assumption 8.2, there is exactly one complete thread trace $\mathbb{TT}$ that begins at $\langle \emptyset, \mathit{program}_{\mathsf{prim}};;, [\,], \emptyset\rangle$, and the last thread of this trace is $\langle \mathit{env}_{\mathsf{prim}}, \varepsilon, [\,], \emptyset\rangle$. So there is no call to the **random** function inside the initialization of the primitives. Let $\mathbb{TT}(\mathit{definitions})$ be the trace $\mathbb{TT}$ where, in each thread, we replace the empty definition list ε by $\mathit{definitions}$. As no OCaml reduction rule depends on the contents of a definition list, the trace $\mathbb{TT}(\mathit{definitions})$ is a valid trace for any definition list *definitions*. So, by taking *definitions* the definitions after $\mathit{program}_{\mathsf{prim}}$ in th^{s} and th^{o},

$$\begin{aligned} th^{\mathsf{s}} \to^* th^{\mathsf{s}}_e &\overset{\text{def}}{=} \langle \mathit{env}_{\mathsf{prim}}, \mathit{program}'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots;; \mathit{program}'(\mathsf{role}_m[\widetilde{a_m}]);; \\ &\qquad \mathit{program}', [\,], \emptyset\rangle \\ th^{\mathsf{o}} \to^* th^{\mathsf{o}}_e &\overset{\text{def}}{=} \langle \mathit{env}_{\mathsf{prim}}, \mathit{program}(\mu_{\mathsf{role}_1});; \ldots;; \mathit{program}(\mu_{\mathsf{role}_m});; \mathit{program}', [\,], \emptyset\rangle \end{aligned}$$

in exactly the same number of steps.

Let l_j ($j \le m$) be m distinct locations. For $j \le m+1$, let $th^{\mathsf{s}}_j \overset{\text{def}}{=} \langle \mathit{env}^{\mathsf{s}}_j, \mathit{program}_j, [\,], \mathit{store}^{\mathsf{s}}_j\rangle$ where $\mathit{env}^{\mathsf{s}}_j \overset{\text{def}}{=} \mathit{env}_{\mathsf{prim}} \cup \{\mu_{\mathsf{role}_i}.\mathit{init} \mapsto c_i \mid i < j\}$ with $c_i \overset{\text{def}}{=} \mathbf{tagfunction}^{\mathsf{role}_i, \tau_i}[\mathit{env}_i, \mathit{pm}'_{\mathsf{role}_i[\widetilde{a_i}]}]$ and $\mathit{env}_i \overset{\text{def}}{=} \mathit{env}^{\mathsf{s}}_i[\mathit{token} \mapsto l_i]$, $\mathit{program}_j \overset{\text{def}}{=} \mathit{program}'(\mathsf{role}_j[\widetilde{a_j}]);; \ldots;; \mathit{program}'(\mathsf{role}_m[\widetilde{a_m}]);; \mathit{program}'$ for $j \le m$, $\mathit{program}_j \overset{\text{def}}{=} \mathit{program}'$ for $j = m+1$, and $\mathit{store}^{\mathsf{s}}_j \overset{\text{def}}{=} \{l_i \mapsto \mathbf{Callable} \mid i < j\}$. For $j \le m$, the thread th^{s}_j reduces as follows:

$$\begin{aligned} th^{\mathsf{s}}_j &= \langle \mathit{env}^{\mathsf{s}}_j, \mathbf{let}\ \mu_{\mathsf{role}_j}.\mathit{init} = e^s_j;; \mathit{program}_{j+1}, [\,], \mathit{store}^{\mathsf{s}}_j\rangle \\ &\qquad \text{where } e^s_j \overset{\text{def}}{=} \mathbf{let}\ \mathit{token} = \mathbf{ref\ Callable\ in\ tagfunction}^{\mathsf{role}_j}\ \mathit{pm}'_{\mathsf{role}_j[\widetilde{a_j}]} \\ &\to \langle \mathit{env}^{\mathsf{s}}_j, e^s_j, \mathit{stack}^{\mathsf{s}}_j, \mathit{store}^{\mathsf{s}}_j\rangle \\ &\qquad \text{where } \mathit{stack}^{\mathsf{s}}_j \overset{\text{def}}{=} [\mathit{env}^{\mathsf{s}}_j, \mathbf{let}\ \mu_{\mathsf{role}_j}.\mathit{init} = [\cdot];; \mathit{program}_{j+1}] \\ &\to^* \langle \mathit{env}_j, \mathbf{tagfunction}^{\mathsf{role}_j}\ \mathit{pm}'_{\mathsf{role}_j[\widetilde{a_j}]}, \mathit{stack}^{\mathsf{s}}_j, \mathit{store}^{\mathsf{s}}_{j+1}\rangle \\ &\qquad \text{since } \mathit{env}_j = \mathit{env}^{\mathsf{s}}_j[\mathit{token} \mapsto l_j] \text{ and } \mathit{store}^{\mathsf{s}}_{j+1} = \mathit{store}^{\mathsf{s}}_j[l_j \mapsto \mathbf{Callable}] \\ &\to \langle \mathit{env}_j, c_j, \mathit{stack}^{\mathsf{s}}_j, \mathit{store}^{\mathsf{s}}_{j+1}\rangle \\ &\to^* th^{\mathsf{s}}_{j+1} = \langle \mathit{env}^{\mathsf{s}}_{j+1}, \mathit{program}_{j+1}, [\,], \mathit{store}^{\mathsf{s}}_{j+1}\rangle \\ &\qquad \text{since } \mathit{env}^{\mathsf{s}}_{j+1} = \mathit{env}^{\mathsf{s}}_j[\mu_{\mathsf{role}_j}.\mathit{init} \mapsto c_j] \end{aligned}$$

Let $th^{\mathsf{o}}_j \overset{\text{def}}{=} \mathit{replaceinitpm}(th^{\mathsf{s}}_j)$. We have $th^{\mathsf{o}}_j = \langle \mathit{env}^{\mathsf{o}}_j, \mathit{program}'_j, [\,], \mathit{store}^{\mathsf{s}}_j\rangle$ where $\mathit{env}^{\mathsf{o}}_j$ is the environment $\mathit{env}^{\mathsf{s}}_j$ in which we replace $\mathit{pm}'_{\mathsf{role}_i[\widetilde{a_i}]}$ with $\mathit{pm}_{\mu_{\mathsf{role}_i}}$ for all $i < j$, $\mathit{program}'_j \overset{\text{def}}{=} \mathit{program}(\mu_{\mathsf{role}_j});; \ldots;; \mathit{program}(\mu_{\mathsf{role}_m});; \mathit{program}'$ for $j \le m$, and $\mathit{program}'_j \overset{\text{def}}{=} \mathit{program}'$ for $j = m+1$. For $j \le m$, the thread th^{o}_j reduces as follows:

$$\begin{aligned} th^{\mathsf{o}}_j &= \langle \mathit{env}^{\mathsf{o}}_j, \mathbf{let}\ \mu_{\mathsf{role}_j}.\mathit{init} = e^o_j;; \mathit{program}'_{j+1}, [\,], \mathit{store}^{\mathsf{s}}_j\rangle \\ &\qquad \text{where } e^o_j \overset{\text{def}}{=} \mathbf{let}\ \mathit{token} = \mathbf{ref\ Callable\ in\ tagfunction}^{\mathsf{role}_j}\ \mathit{pm}_{\mu_{\mathsf{role}_j}} \\ &\to \langle \mathit{env}^{\mathsf{o}}_j, e^o_j, \mathit{stack}^{\mathsf{o}}_j, \mathit{store}^{\mathsf{s}}_j\rangle \\ &\qquad \text{where } \mathit{stack}^{\mathsf{o}}_j \overset{\text{def}}{=} [\mathit{env}^{\mathsf{o}}_j, \mathbf{let}\ \mu_{\mathsf{role}_j}.\mathit{init} = [\cdot];; \mathit{program}'_{j+1}] \\ &\to^* \langle \mathit{env}'_j, \mathbf{tagfunction}^{\mathsf{role}_j}\ \mathit{pm}_{\mu_{\mathsf{role}_j}}, \mathit{stack}^{\mathsf{o}}_j, \mathit{store}^{\mathsf{s}}_{j+1}\rangle \\ &\qquad \text{where } \mathit{env}'_j \overset{\text{def}}{=} \mathit{env}^{\mathsf{o}}_j[\mathit{token} \mapsto l_j] \end{aligned}$$

$\to \langle \mathit{env}'_j, c'_j, \mathit{stack}^{\mathsf{o}}_j, \mathit{store}^{\mathsf{s}}_{j+1}\rangle$

where $c'_j \overset{\text{def}}{=} \mathbf{tagfunction}^{\mathsf{role}_j,\tau_j}[\mathit{env}'_j, \mathit{pm}_{\mu_{\mathsf{role}_j}}]$

$\to^* \mathit{th}^{\mathsf{o}}_{j+1} = \langle \mathit{env}^{\mathsf{o}}_{j+1}, \mathit{program}'_{j+1}, [\,], \mathit{store}^{\mathsf{s}}_{j+1}\rangle$

since $\mathit{env}^{\mathsf{o}}_{j+1} = \mathit{env}^{\mathsf{o}}_j[\mu_{\mathsf{role}_j}.\mathit{init} \mapsto c'_j]$

Moreover, $\mathit{th}^{\mathsf{s}}_e = \mathit{th}^{\mathsf{s}}_1$ and $\mathit{th}^{\mathsf{o}}_e = \mathit{th}^{\mathsf{o}}_1$, so $\mathit{th}^{\mathsf{s}} \to^* \mathit{th}^{\mathsf{s}}_{m+1}$ and $\mathit{th}^{\mathsf{o}} \to^* \mathit{th}^{\mathsf{o}}_{m+1}$. There are exactly the same number of steps in both traces. Let steps^s be this number of steps.

Let $\mathbb{CT}_1$ be the extension of the trace $\mathbb{CT}$ until $\mathbb{C}[\mathsf{th} \mapsto \mathit{th}^{\mathsf{o}}_{m+1}]$. Since $\mathit{th}^{\mathsf{s}} \to^* \mathit{th}^{\mathsf{s}}_{m+1}$ without using (Random), we have $\mathbb{CS} \to^* \mathbb{CS}[\mathsf{th} \mapsto \mathit{th}^{\mathsf{s}}_{m+1}]$ 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 $|\mathbb{CT}_1| = |\mathbb{CT}| + \mathit{steps}^s \leq N_{\mathsf{steps}}$. Hence, by Property I14, $\mathit{steps} \geq N_{\mathsf{steps}} - |\mathbb{CT}| \geq \mathit{steps}^s$. So, with $\mathcal{C}^{\mathsf{cs}}_1 \overset{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathit{steps} - \mathit{steps}^s, \mathbb{CS}[\mathsf{th} \mapsto \mathit{th}^{\mathsf{s}}_{m+1}]$, we have $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathcal{C}^{\mathsf{cs}}_1$ 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 I14. We shall not detail this point in the other cases.)

Let us prove that $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}_1$. Properties I1, I2, I3, I4, I7, I8, I9, I10, I11, I12 are inherited from $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$. As mentioned in Section 5.2.6, there are no local store locations in the initial program, so there are no local store locations in $\mathit{program}'$, so the locations $l_1, \dots, l_m$ are the only local store locations present in $\mathit{th}^{\mathsf{s}}_{m+1}$, and they are all in the domain of $\mathit{store}^{\mathsf{s}}_{m+1}$. So Property I5 holds. Let l_{tok} be the empty function. The set $\mathcal{O}_{\mathbf{call}}(\mathit{th}^{\mathsf{s}}_{m+1})$ is empty. We have that $\mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(\mathit{th}^{\mathsf{s}}_{m+1}), l_{\mathsf{tok}}) = \emptyset$ and $\mathit{th}^{\mathsf{o}}_{m+1} = \mathit{replaceinitpm}(\mathit{th}^{\mathsf{s}}_{m+1}) \in \mathit{replacecalls}(\mathit{replaceinitpm}(\mathit{th}^{\mathsf{s}}_{m+1}), \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so Property T2(a) 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 T2(b) holds for the current thread. The environment of the tagged closures that we created contains $\mathit{env}_{\mathsf{prim}}$, so Property T2(c) holds for the current thread. Since there is no tagged function, no event and no return in $\mathit{program}'$ except in $\mathit{program}(\mu_{\mathsf{role}})$ in arguments of **addthread**, Property T2(d) holds for the current thread. Threads that are not the current thread did not change, so Property I6 holds. The only change in the oracle sets is that the roles $\mathsf{role}_j[\widetilde{a_j}]$ are transferred from $\mathcal{R}_{\mathsf{init\text{-}function}}(\mathit{th}^{\mathsf{s}})$ to $\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathit{th}^{\mathsf{s}})$, so Property I13 is preserved. We have

$$|\mathbb{CT}_1| + \mathit{steps} - \mathit{steps}^s = |\mathbb{CT}| + \mathit{steps}^s + \mathit{steps} - \mathit{steps}^s \geq N_{\mathsf{steps}}$$

so Property I14 holds. Properties I15, I16, and I17 are preserved, because all components of these inequalities are unchanged. Therefore, we have proved that $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}_1$.

Case 2. Property T2 is satisfied for the current thread. We now distinguish cases on the form of the simulator configuration $\mathbb{CS}$.

Case 2.1. The current expression of $\mathbb{CS}$ is $\mathit{pe}^{\mathsf{s}} = \mathbf{call}(O_j[\widetilde{a}])\ (v_1, \dots, v_{m_j})$ and

$\mathbb{CS}$ cannot reduce, that is, the configuration $\mathbb{CS}$ makes a successful call to $O_j[\widetilde{a}]$, an oracle not under replication. By definition of *simreturn*, $\mathit{simreturn}(\mathbb{CS}) \stackrel{\text{def}}{=} (\mathit{repr}(\mathbb{CS}), \mathrm{o}_j, \widetilde{a}, \mathit{args})$ where $\mathit{args} \stackrel{\text{def}}{=} (b_1, \dots, b_{m_j})$ and $b_k \stackrel{\text{def}}{=} \mathbb{G}^{-1}_{\mathsf{val}T_{j,k}}(v_k)$ for $k \leq m_j$.

So $\mathcal{C}^{\mathsf{cs}}$ reduces in several steps into the configuration $E_1, P'_{\mathsf{loop}}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E}$ that corresponds to line 8 where

$$\begin{aligned}E_1 \stackrel{\text{def}}{=} E[&\mathit{xs}[\alpha] \mapsto (\mathit{repr}(\mathbb{CS}), \mathrm{o}_j, \widetilde{a}, \mathit{args}),\\ &s'[\alpha] \mapsto \mathit{repr}(\mathbb{CS}), o[\alpha] \mapsto \mathrm{o}_j, i[\alpha] \mapsto \widetilde{a}, \mathit{args}[\alpha] \mapsto \mathit{args}]\,.\end{aligned}$$

Let $a'_1, \dots, a'_{n_j} \stackrel{\text{def}}{=} \widetilde{a}$. As $E_1(o[\alpha]) = \mathrm{o}_j$, this configuration reduces in several steps into the configuration $E_2, P^j_{\mathsf{loop}}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E}$ where

$$\begin{aligned}E_2 \stackrel{\text{def}}{=} E_1[&\mathit{xa}_j[\alpha] \mapsto \mathit{args}, a_{j,1}[\alpha] \mapsto b_1, \dots, a_{j,m_j}[\alpha] \mapsto b_{m_j},\\ &\mathit{xi}_j[\alpha] \mapsto \widetilde{a}, i_{j,1}[\alpha] \mapsto a'_1, \dots, i_{j,n_j}[\alpha] \mapsto a'_{n_j}]\,.\end{aligned}$$

The oracle $O_j[\widetilde{a}]$ is in $\mathbb{I}$, otherwise $\mathbb{CS}$ could reduce using (FailedCall1). By Property I3 of the invariant, there exists $\mathcal{Q}_0$ such that $\mathcal{Q}_0 \leftrightarrow \mathbb{I}, \mathbb{RI}$ and $\mathcal{Q} = \{Q_{\mathsf{loop}}\{a/i'\} \mid \alpha < a \leq N_{\mathsf{rand+calls}}\} \cup \mathcal{Q}_0$. Let $Q \stackrel{\text{def}}{=} \mathcal{Q}_0(O_j[\widetilde{a}])$. The oracle definition Q is of the form $O_j[\widetilde{a}](x_1[\widetilde{a}] : T_{j,1}, \dots, x_{m_j}[\widetilde{a}] : T_{j,m_j}) := P_O$. The previous configuration reduces in one step into $\mathcal{C} \stackrel{\text{def}}{=} E_3, P_O, \mathcal{T}, \mathcal{Q}_1, \mathcal{S}_1, \mathcal{E}$ where

$$\begin{aligned}E_3 &\stackrel{\text{def}}{=} E_2[x_1[\widetilde{a}] \mapsto b_1, \dots, x_{m_j}[\widetilde{a}] \mapsto b_{m_j}]\\ \mathcal{S}_1 &\stackrel{\text{def}}{=} ((r_{j,1}, \dots, r_{j,m'_j}), \mathsf{return}(\text{simulate}_{\mathsf{ret}O_j}(s', (r_{j,1}, \dots, r_{j,m'_j})), \text{continue}),\\ &\qquad \mathsf{return}(\text{simulate}_{\mathsf{end}O_j}(s'), \text{continue})) :: \mathcal{S}_{\mathsf{loop}}(\alpha)\\ \mathcal{Q}_1 &\stackrel{\text{def}}{=} \mathcal{Q} \setminus \{Q\}\end{aligned}$$

Let us now look at $\mathbb{C}$. By the invariant, there exists an injection l_{tok} that satisfies Property T2(a). The current expression pe° is of the form $c\ (v_1, \dots, v_{m_j})$, where $c \in \mathit{correctclosure}(O_j[\widetilde{a}], \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $c = \mathbf{tagfunction}^{O_j,\tau}[\mathit{env}^\circ_1, \mathit{pm}_{\mathsf{Once}}(Q)]$ where $\mathit{env}^\circ_1 \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E, Q)$ and $\mathit{env}^\circ_1(\mathit{token}) = l_{\mathsf{tok}}(O_j[\widetilde{a}])$. By the same property, $\mathit{store}^\circ(l_{\mathsf{tok}}(O_j[\widetilde{a}])) = \mathbf{Callable}$.

$$\begin{aligned}\mathit{th}^\circ &= \langle \mathit{env}^\circ, c\ (v_1, \dots, v_{m_j}), \mathit{stack}^\circ, \mathit{store}^\circ\rangle\\ &\to \langle \mathit{env}^\circ_1, \mathbf{match}\ (v_1, \dots, v_{m_j})\ \mathbf{with}\ \mathit{pm}_{\mathsf{Once}}(Q), \mathit{stack}^\circ, \mathit{store}^\circ\rangle\\ \to \mathit{th}^\circ_1 &\stackrel{\text{def}}{=} \langle \mathit{env}^\circ_2, e, \mathit{stack}^\circ, \mathit{store}^\circ\rangle\\ &\quad \text{where } \mathit{env}^\circ_2 \stackrel{\text{def}}{=} \mathit{env}^\circ_1[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto v_1, \dots, \mathbb{G}_{\mathsf{var}}(x_{m_j}) \mapsto v_{m_j}] \text{ and}\\ &\quad e \stackrel{\text{def}}{=} \mathbf{if}\ (!\mathit{token} = \mathbf{Callable})\ \&\&\\ &\qquad\quad (\mathbb{G}_{\mathsf{pred}}(T_{j,1})\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \dots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_{j,m_j})\ \mathbb{G}_{\mathsf{var}}(x_{m_j}))\\ &\qquad \mathbf{then}\ (\mathit{token} := \mathbf{Invalid}; e')\ \mathbf{else\ raise\ Bad\_Call}\\ &\quad e' \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \dots; \mathbb{G}_{\mathsf{file}}(x_{m_j}[\widetilde{a}]); \mathbb{G}(P_O)\end{aligned}$$

For all $k \leq m_j$, 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 8.5. Moreover, $env_2^\circ(token) = l_{\mathsf{tok}}(O_j[\widetilde{a}])$. So, the configuration $\mathbb{C}$ reduces as follows

$$\begin{aligned}
&\mathbb{C} \to^* \mathbb{C}' \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto th_1^\circ] \\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle env_2^\circ, token := \mathbf{Invalid}; e', stack^\circ, store_1^\circ\rangle] \\
&\quad \to^* \mathbb{C}[\mathsf{th} \mapsto \langle env_2^\circ, e', stack^\circ, store_2^\circ\rangle] \\
&\qquad\quad \text{where } store_2^\circ \overset{\text{def}}{=} store_1^\circ[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathbf{Invalid}] \\
&\to^* \mathbb{C}_1 \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle env_2^\circ, \mathbb{G}(P_O), stack^\circ, store_3^\circ\rangle, \mathsf{globalstore} \mapsto globalstore_1^\circ] \\
&\hspace{20em} \text{by Lemma D.1 applied } m_j \text{ times}
\end{aligned}$$

where $store_3^\circ \supseteq store_2^\circ$, $store_1^\circ \supseteq store^\circ$, $globalstore_1^\circ \supseteq globalstore(E_3, \mathcal{T})$ since $globalstore^\circ \supseteq globalstore(E, \mathcal{T})$ by Property I9 of the invariant, and $globalstore_1^\circ(l) = globalstore^\circ(l)$ for all $l \notin Loc_{\mathsf{priv}}$.

We prove that for any traces $\mathcal{CT}_1, \dots, \mathcal{CT}_m$ beginning at $\mathcal{C}$ such that $\sum_{i \leq m} \Pr[\mathcal{CT}_i] = 1$, 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 $\mathbb{CTS}_1, \dots, \mathbb{CTS}_m$ all starting at $\mathbb{C}_1$ such that none of these traces is a prefix of another of these traces, $\Pr[\mathbb{CTS}_i] = \Pr[\mathcal{CT}_i]$ for all $i \leq m$, and if $\mathbb{C}_4$ is the last configuration of a trace $\mathbb{CT}' \in \mathbb{CTS}_i$, then $\mathbb{C}_4 = \mathbb{C}[\mathsf{th} \mapsto th_4^\circ, \mathsf{globalstore} \mapsto globalstore_4^\circ, \mathsf{events} \mapsto events_4]$ where

$$\begin{aligned}
th_4^\circ &= \langle env_4^\circ, \mathbb{G}(P_4), stack^\circ, store_4^\circ\rangle \text{ with} \\
&\qquad env_4^\circ \supseteq env_{\mathsf{prim}} \cup env(E_4, P_4) \text{ and } store_4^\circ \supseteq store_3^\circ\,, \\
globalstore_4^\circ &\supseteq globalstore(E_4, \mathcal{T}_4)\,, \\
globalstore_4^\circ(l) &= globalstore^\circ(l) \text{ for all } l \notin Loc_{\mathsf{priv}}\,, \\
events_4 &= \mathbb{G}_{\mathsf{ev}}(\mathcal{E}_4)\,,
\end{aligned}$$

and the last configuration of $\mathcal{CT}_i$ is $E_4, P_4, \mathcal{T}_4, \mathcal{Q}_1, \mathcal{S}_1, \mathcal{E}_4$.

The proof proceeds by induction on the total length of the traces $\mathcal{CT}_1, \dots, \mathcal{CT}_m$. In the base case, $m = 1$ and $\mathcal{CT}_1$ is the trace that consists only of the configuration $\mathcal{C}$. Let $\mathbb{CTS}_1$ consist of the single trace that contains just the configuration $\mathbb{C}_1$. We have $env_2^\circ \supseteq env_{\mathsf{prim}} \cup env(E_3, P_O)$ since $env^\circ \supseteq env_{\mathsf{prim}} \cup env(E, Q)$, the variables $x_1[\widetilde{a}], \dots, x_{m_j}[\widetilde{a}]$ are added on the CryptoVerif side, and $\mathbb{G}_{\mathsf{var}}(x_1), \dots, \mathbb{G}_{\mathsf{var}}(x_{m_j})$ are added correspondingly on the OCaml side. As shown above, $globalstore_1^\circ \supseteq globalstore(E_3, \mathcal{T})$ and $globalstore_1^\circ(l) = globalstore^\circ(l)$ for all $l \notin Loc_{\mathsf{priv}}$. By Property I12 of the invariant, $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$. So the property holds for the base case. The inductive case follows from Lemma 8.10.

Let us take the maximal CryptoVerif traces $\mathcal{CT}_1, \dots, \mathcal{CT}_n$ that begin at $\mathcal{C}$ and that contain no return, end, call, or loop as current process in intermediate configurations. Let $\mathbb{CTS}_1, \dots, \mathbb{CTS}_n$ the trace sets as defined above. The final configurations of the CryptoVerif traces $\mathcal{CT}_i$ 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 $\mathcal{CT}_i$ and a trace $\mathbb{CT}' \in \mathbb{CTS}_i$. Let $\mathcal{C}_4$ and $\mathbb{C}_4$ be the last configurations of $\mathcal{CT}_i$ and $\mathbb{CT}'$ respectively. Let $\mathcal{C}_4 = E_4, P_4, \mathcal{T}_4, \mathcal{Q}_1, \mathcal{S}_1, \mathcal{E}_4$. We distinguish cases on the form of P_4.

- If $P_4 = \mathsf{end}$,

$$
\begin{aligned}
\mathcal{C}_4 &\rightsquigarrow E_4, \mathsf{return}(\mathrm{simulate}_{\mathsf{end}O_j}(s'[\alpha]), \mathrm{continue}), \mathcal{T}_4, \mathcal{Q}_1, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E}_4\\
&\rightsquigarrow E_5, P_{\mathsf{return\text{-}loop}}(\alpha), \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4\\
&\qquad \text{where } E_5 \overset{\text{def}}{=} E_4[r'_{\alpha,r}[\,] \mapsto s, b_{\alpha,r} \mapsto \mathrm{continue}],\\
&\qquad\qquad s \overset{\text{def}}{=} \mathit{repr}(\mathbb{CS}'),\\
&\qquad\qquad \mathbb{CS}' \text{ is } \mathbb{CS} \text{ in which the current expression is replaced}\\
&\qquad\qquad \text{with } \textbf{raise Match\_failure} \text{ and the set } \mathbb{I} \text{ is replaced}\\
&\qquad\qquad \text{with } \mathbb{I}' \overset{\text{def}}{=} \mathbb{I} - (O_j[\widetilde{a}])\\
&\rightsquigarrow E_5, P_5, \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4\\
&\qquad \text{where } P_5 \overset{\text{def}}{=} \mathsf{let}\ r[\,] : T_{\mathbb{CS}} = \mathsf{loop}\ \mathrm{O}_{\mathsf{loop}}[\alpha+1](r'_{\alpha,r}[\,])\\
&\qquad\qquad\qquad \mathsf{in\ end\ else\ end}\\
&\rightsquigarrow E_5, P_6, \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4\\
&\qquad \text{where } P_6 \overset{\text{def}}{=} \mathsf{let}\ (r'_{\alpha+1,r}[\,] : T_{\mathbb{CS}}, b_{\alpha+1,r}[\,] : \mathit{bool}) =\\
&\qquad\qquad\qquad \mathrm{O}_{\mathsf{loop}}[\alpha+1](r'_{\alpha,r})\ \mathsf{in}\ P_{\mathsf{return\text{-}loop}}(\alpha+1)\ \mathsf{else\ end}\\
&\rightsquigarrow E_6, P_{\mathsf{loop}}\{\alpha+1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{S}_{\mathsf{loop}}(\alpha+1), \mathcal{E}_4\\
&\qquad \text{where } E_6 \overset{\text{def}}{=} E_5[s[\alpha+1] \mapsto s],\\
&\qquad\qquad \mathcal{Q}_2 \overset{\text{def}}{=} \mathcal{Q}_1 \setminus \{Q_{\mathsf{loop}}\{\alpha+1/i'\}\}
\end{aligned}
$$

(Let $\mathbb{CT}''$ be the trace $\mathbb{CT}$ followed by $\mathbb{CT}'$. By definition of $N_{\mathsf{rand+calls}}$, $N_{\mathsf{rand+calls}} \geq \Big(N_{\mathsf{rand}}(\mathbb{CT}'') + \sum_{O,\tau} N_{\mathsf{calls}}(O,\tau,\mathbb{CT}'')\Big) + 1 = \Big(N_{\mathsf{rand}}(\mathbb{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O,\tau,\mathbb{CT})\Big) + 2$ since $\mathbb{CT}''$ makes one more call to O_j than $\mathbb{CT}$. So, by Property I15, $N_{\mathsf{rand+calls}} \geq \alpha+1$. So, by Property I3, $Q_{\mathsf{loop}}\{\alpha+1/i'\} \in \mathcal{Q}$, so $Q_{\mathsf{loop}}\{\alpha+1/i'\} \in \mathcal{Q}_1$.)

$$\rightsquigarrow \mathcal{C}_1^{\mathsf{cs}} \overset{\text{def}}{=} E_6, P_{\mathsf{loop}}\{\alpha+1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{S}_{\mathsf{loop}}(\alpha+1), \mathcal{E}_4, N_{\mathsf{steps}}, \mathbb{CS}'$$

By definition of the translation of end, the current expression of $\mathbb{C}_4$ is **raise Match_failure**. Let $\mathbb{CT}''$ be the trace $\mathbb{CT}$ followed by $\mathbb{CT}'$. The last configuration of $\mathbb{CT}''$ is $\mathbb{C}_4$.

Let us prove that $\mathcal{C}_1^{\mathsf{cs}} \equiv \mathbb{CT}''$. By the form of $\mathcal{C}_1^{\mathsf{cs}}$ and $\mathbb{C}_4$, Properties I1 and I2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ where we removed the oracles $O_j[\widetilde{a}]$ and $\mathrm{O}_{\mathsf{loop}}[\alpha+1]$. We have $\mathbb{I}' = \mathbb{I} - (O_j[\widetilde{a}])$, so Property I3 is preserved. Property I4 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 I5 is preserved.

For all threads $tj' \neq tj$, the thread tj' does not change so, to prove Property I6, we just have to show that Property T2(a) is preserved; the other elements of Property I6 are obviously preserved. Suppose that thread tj' satisfies Property T2(a) initially, with a function l_{tok}. By Lemma F.3, Item 1, for all $\mathbf{call}(R)$ that occur in $th''_{tj'} \overset{\text{def}}{=} \mathit{replaceinitpm}(th_{tj'})$, we have $\mathit{correctclosure}(R, \mathbb{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathit{correctclosure}(R, \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $\mathit{replacecalls}(th''_{tj'}, \mathbb{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathit{replacecalls}(th''_{tj'}, \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Furthermore, the oracle $O_j[\widetilde{a}]$ is in $\mathcal{O}_{\mathbf{call}}(th_{tj})$, so by Property O2 of the invariant, it is not in $\mathcal{O}_{\mathbf{call}}(th_{tj'})$. Hence, $\mathcal{O}_{\mathbf{call}}(th_{tj'}) \cap \mathbb{I}' = \mathcal{O}_{\mathbf{call}}(th_{tj'}) \cap \mathbb{I}$ and $\mathcal{O}_{\mathbf{call}}(th_{tj'}) \setminus \mathbb{I}' = \mathcal{O}_{\mathbf{call}}(th_{tj'}) \setminus \mathbb{I}$, so $\mathit{gettokens}(\mathbb{I}', \mathcal{O}_{\mathbf{call}}(th_{tj'}), l_{\mathsf{tok}}) = \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(th_{tj'}), l_{\mathsf{tok}})$. So thread tj' continues to verify Property T2(a) with the same function l_{tok}.

Let us now consider the current thread. The current thread of the simulator is $th^{\mathsf{s}}_1 = \langle \mathit{env}^{\mathsf{s}}, \textbf{raise Match\_failure}, \mathit{stack}^{\mathsf{s}}, \mathit{store}^{\mathsf{s}} \rangle$ and the current thread on the OCaml side is $th^{\mathsf{o}}_4 = \langle \mathit{env}^{\mathsf{o}}_4, \textbf{raise Match\_failure}, \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}}_4 \rangle$ where $\mathit{store}^{\mathsf{o}}_4 \supseteq \mathit{store}^{\mathsf{o}}_3$. By Property T2(a), there exist $\mathit{store}^{\mathsf{o}}_5$ and l_{tok} such that

$$\begin{aligned}
&\langle \mathit{env}^{\mathsf{o}}, \mathit{pe}^{\mathsf{o}}, \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}}_5 \rangle \\
&\qquad \in \mathit{replacecalls}(\mathit{replaceinitpm}(th^{\mathsf{s}}), \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \\
&\mathit{store}^{\mathsf{o}}_5 \cup \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}), l_{\mathsf{tok}}) \subseteq \mathit{store}^{\mathsf{o}} \,.
\end{aligned}$$

Let us denote $th'' \overset{\text{def}}{=} \mathit{replaceinitpm}(th^{\mathsf{s}})$ and $th''_1 \overset{\text{def}}{=} \mathit{replaceinitpm}(th^{\mathsf{s}}_1)$. 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

$$\begin{aligned}
th^{\mathsf{o}}_5 &\overset{\text{def}}{=} \langle \mathit{env}^{\mathsf{o}}_4, \textbf{raise Match\_failure}, \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}}_5 \rangle \\
&\in \mathit{replacecalls}(th''_1, \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \,.
\end{aligned}$$

Let $l'_{\mathsf{tok}} \overset{\text{def}}{=} l_{\mathsf{tok}|\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_1)}$. By Lemma F.3, Item 1, for all $\mathbf{call}(R)$ that occur in th''_1, we have $\mathit{correctclosure}(R, \mathbb{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathit{correctclosure}(R, \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $th^{\mathsf{o}}_5 \in \mathit{replacecalls}(th''_1, \mathbb{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau_{\mathsf{O}})$. We have

$$\begin{aligned}
&\mathit{store}^{\mathsf{o}}_5 \cup \mathit{gettokens}(\mathbb{I}', \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_1), l'_{\mathsf{tok}}) \\
&\quad \subseteq \mathit{store}^{\mathsf{o}}_5 \cup \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}), l_{\mathsf{tok}})[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \textbf{Invalid}] \\
&\quad \subseteq \mathit{store}^{\mathsf{o}}[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \textbf{Invalid}] \\
&\quad \subseteq \mathit{store}^{\mathsf{o}}_1[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \textbf{Invalid}] = \mathit{store}^{\mathsf{o}}_2 \subseteq \mathit{store}^{\mathsf{o}}_3 \subseteq \mathit{store}^{\mathsf{o}}_4 \,,
\end{aligned}$$

so Property T2(a) holds with the function l'_{tok}. Properties T2(b), T2(c), T2(d) are clearly preserved, so Property I6 holds.

Properties I7, I8, and I11 are also preserved. Properties I9, I10, and I12 hold because they are kept inside Lemma 8.10. We have $\mathcal{O}^{\infty}(\mathbb{I}') = \mathcal{O}^{\infty}(\mathbb{I}) \setminus \{O_j[\widetilde{a}]\}$. If there remains no occurrence of

$\mathbf{call}(O_j[\widetilde{a}])$ in the thread th^{s}_1, then $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_1) = \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}) \setminus \{O_j[\widetilde{a}]\}$ and $\mathcal{O}_{\mathbf{call}}(\mathbb{CS}') = \mathcal{O}_{\mathbf{call}}(\mathbb{CS}) \setminus \{O_j[\widetilde{a}]\}$, so $\mathcal{O}^{\infty}(\mathbb{I}') \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS}') = \mathcal{O}^{\infty}(\mathbb{I}) \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS}) \setminus \{O_j[\widetilde{a}]\}$. Otherwise, $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_1) = \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}})$ and $\mathcal{O}_{\mathbf{call}}(\mathbb{CS}') = \mathcal{O}_{\mathbf{call}}(\mathbb{CS})$, so $\mathcal{O}^{\infty}(\mathbb{I}') \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS}') = \mathcal{O}^{\infty}(\mathbb{I}) \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS})$. We also have $\mathcal{O}_{\mathbf{call\text{-}repl}}(th^{\mathsf{s}}_1) = \mathcal{O}_{\mathbf{call\text{-}repl}}(th^{\mathsf{s}})$, $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}function}}(th^{\mathsf{s}}_1)) = \mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}function}}(th^{\mathsf{s}}))$, and $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(th^{\mathsf{s}}_1)) = \mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(th^{\mathsf{s}}))$, so Property O2 is preserved. The set $\mathcal{O}^{\infty}(\mathbb{I}') \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS}')$ is included in $\mathcal{O}^{\infty}(\mathbb{I}) \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS})$ and the set $\mathit{willbeavailable}(\mathbb{CS}')$ is included in $\mathit{willbeavailable}(\mathbb{CS})$, so Property O1 is preserved, which shows Property I13. We have $|\mathbb{CT}''| + N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property I14 holds. The number of calls to O_j increases by 1 and α increases by 1, so Property I15 is preserved. Properties I16 and I17 are preserved, because all components of these inequalities are unchanged. So $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}''$ in this case.

- If $P_4 = \mathsf{return}(M_1, \dots, M_{m'_j}); Q'$, let c_i $(i \leq m'_j)$ be the CryptoVerif values such that $E_4 \cdot M_i \Downarrow c_i$.

$$\begin{aligned}
\mathcal{C}_4 \rightsquigarrow\ & E_5, \mathsf{return}(\text{simulate}_{\mathsf{ret}O_j}(s'[\alpha], (r_{j,1}[\alpha], \dots, r_{j,m'_j}[\alpha])), \text{continue}),\\
& \quad \mathcal{T}_4, \mathcal{Q}_1, \mathcal{S}, \mathcal{E}_4\\
& \quad \text{where } E_5 \stackrel{\text{def}}{=} E_4[r_{j,1} \mapsto c_1, \dots, r_{j,m'_j} \mapsto c_{m'_j}]\\
\rightsquigarrow^* \mathcal{C}^{\mathsf{cs}}_1 \stackrel{\text{def}}{=}\ & E_6, P_{\mathsf{loop}}\{\alpha + 1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{S}_{\mathsf{loop}}(\alpha + 1), \mathcal{E}_4, N_{\mathsf{steps}}, \mathbb{CS}'\\
& \quad \text{where } E_6 \supseteq E_5[s[\alpha + 1] \mapsto \mathit{repr}(\mathbb{CS}')],\\
& \qquad \mathcal{Q}_2 \stackrel{\text{def}}{=} \mathcal{Q}_1 \cup \mathit{oracledefset}(Q') \setminus \{Q_{\mathsf{loop}}\{\alpha + 1/i'\}\}\\
& \qquad \mathit{repr}(\mathbb{CS}') = \text{simulate}_{\mathsf{ret}O_j}(\mathit{repr}(\mathbb{CS}), (c_1, \dots, c_{j,m'_j}))
\end{aligned}$$

where we show that $Q_{\mathsf{loop}}\{\alpha + 1/i'\} \in \mathcal{Q}_1$ using Property I15 as in the case $P_4 = \mathsf{end}$.

Let $\mathit{oracledeflist}(Q') = [(Q_1, \gamma_1), \dots, (Q_l, \gamma_l)]$ and $\mathit{oraclelist}(Q') = [O'_1[\widetilde{a_1}], \dots, O'_l[\widetilde{a_l}]]$. A thread $\langle \mathit{env}, \mathbb{G}_{\mathsf{O}}(Q_i, \gamma_i), \mathit{stack}, \mathit{store}\rangle$ where $\mathit{env} \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_4, P_4)$ reduces into $\langle \mathit{env}', c(Q_i, \gamma_i), \mathit{stack}, \mathit{store}'\rangle$ where $c(Q_i, \gamma_i) \stackrel{\text{def}}{=} \mathbf{tagfunction}^{O'_i, \tau_i}[\mathit{env}', \mathit{pm}_{\gamma_i}(Q_i)]$ and

- if $\gamma_i = \mathsf{Once}$, then $\mathit{env}' = \mathit{env}[\mathit{token} \mapsto l_i]$ and $\mathit{store}' = \mathit{store}[l_i \mapsto \mathbf{Callable}]$ where l_i is a fresh location: $l_i \notin \mathit{Dom}(\mathit{store})$;
- if $\gamma_i = \mathsf{Any}$, then $\mathit{env}' = \mathit{env}$ and $\mathit{store}' = \mathit{store}$.

So in both cases, $\mathit{env}' \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_4, P_4)$.

Let $th^{\mathsf{o}}_4 = \langle \mathit{env}^{\mathsf{o}}_4, \mathbb{G}(P_4), \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}}_4\rangle$ be the current thread of $\mathbb{C}_4$. We have $\mathit{env}^\circ_4 \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_4, P_4)$ and $\mathit{store}^\circ_4 \supseteq \mathit{store}^\circ_3$.

$$\begin{aligned}
\mathit{th}^\circ_4 = \langle &\mathit{env}^\circ_4, (\mathbb{G}_{\mathsf{O}}(Q_1,\gamma_1),\dots,\mathbb{G}_{\mathsf{O}}(Q_l,\gamma_l),\mathbb{G}_{\mathsf{M}}(M_1),\dots,\mathbb{G}_{\mathsf{M}}(M_{m'_j})),\\
&\mathit{stack}^\circ, \mathit{store}^\circ_4\rangle\\
\to^* \langle &\mathit{env}^\circ_4, (\mathbb{G}_{\mathsf{O}}(Q_1,\gamma_1),\dots,\mathbb{G}_{\mathsf{O}}(Q_l,\gamma_l),\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1),\dots,\\
&\mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), \mathit{stack}^\circ, \mathit{store}^\circ_5\rangle\\
&\qquad\qquad\qquad\qquad\text{by Lemma 8.8 applied } m'_j \text{ times}\\
\to^* \mathit{th}^\circ_5 \overset{\text{def}}{=} \langle &\mathit{env}^\circ_4, (c(Q_1,\gamma_1),\dots,c(Q_l,\gamma_l),\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1),\dots,\mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})),\\
&\mathit{stack}^\circ, \mathit{store}^\circ_6\rangle
\end{aligned}$$

where $\mathit{store}^\circ_6 \overset{\text{def}}{=} \mathit{store}^\circ_5 \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} \supseteq \mathit{store}^\circ_5 \supseteq \mathit{store}^\circ_4$.

Let $\mathbb{CT}''$ be the trace $\mathbb{CT}$ followed by $\mathbb{CT}'$ and extended until $\mathbb{C}_5 \overset{\text{def}}{=} \mathbb{C}_4[\mathsf{th} \mapsto \mathit{th}^\circ_5]$. Let $\mathbb{I}'$ be the set $\mathbb{I}$ of $\mathbb{CS}'$.

Let us now prove that $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}''$. We define $\tau'_{\mathsf{O}} \overset{\text{def}}{=} \tau_{\mathsf{O}} \cup \{O'_i[\_,\widetilde{a}] \mapsto \tau_i \mid \gamma_i = \mathsf{Any}\}$. (This function is well defined, because $O'_i[\_,\widetilde{a}] \notin \mathit{Dom}(\tau_{\mathsf{O}})$. Indeed, for any a', $O'_i[a',\widetilde{a}] \in \mathit{willbeavailable}(\mathbb{CS})$, so by Property O1, $O'_i[a',\widetilde{a}] \notin \mathcal{O}^\infty(\mathbb{I})$, hence for any a', $O'_i\big[[a',+\infty[,\widetilde{a}\big] \notin \mathbb{I}$. The main reason why we introduced the set $\mathcal{O}^\infty(\mathbb{I})$ 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 $\mathcal{O}(\mathbb{I})$ instead here, we would not be able to conclude that there is no oracle $O'_i\big[[a',+\infty[,\widetilde{a}\big]$ in $\mathbb{I}$: if $a' > N_{O'_i}$, then $\mathcal{O}(\{O'_i\big[[a',+\infty[,\widetilde{a}\big]\}) = \emptyset$.) By the form of $\mathcal{C}^{\mathsf{cs}}_1$ and $\mathbb{C}_5$, Properties I1 and I2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ where we removed the oracles $O_j[\widetilde{a}]$ and $\mathrm{O}_{\mathsf{loop}}[\alpha+1]$ and where we added the new oracles $\mathit{oracledeflist}(Q')$. By definition of $\mathrm{simulate}_{\mathsf{ret}O_j}$, the set $\mathbb{I}'$ is the set $\mathbb{I}$ where we removed $O_j[\widetilde{a}]$ and added the elements of $\mathit{oraclelist}(Q')$. So Property I3 is preserved. We also have $\mathcal{Q}_2(O'_i[\widetilde{a}_i]) = Q_i$ for $i \le l$. Property I4 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 I5 is preserved.

For all threads $tj' \neq tj$, the thread tj' does not change so, to prove Property I6, we just have to show that Property T2(a) is preserved; the other elements of Property I6 are obviously preserved. Suppose that thread tj' satisfies Property T2(a) initially, with a function l_{tok}. By Lemma F.3, Items 1 and 2, for all $\mathbf{call}(R)$ that occur in $\mathit{th}''_{tj'} \overset{\text{def}}{=} \mathit{replaceinitpm}(\mathit{th}_{tj'})$, we have $\mathit{correctclosure}(R,\mathbb{I}',E_6,\mathcal{Q}_2,l_{\mathsf{tok}},\tau'_{\mathsf{O}}) \supseteq \mathit{correctclosure}(R,\mathbb{I},E,\mathcal{Q}, l_{\mathsf{tok}},\tau_{\mathsf{O}})$, so $\mathit{replacecalls}(\mathit{th}''_{tj'},\mathbb{I}',E_6,\mathcal{Q}_2,l_{\mathsf{tok}},\tau'_{\mathsf{O}}) \supseteq \mathit{replacecalls}(\mathit{th}''_{tj'},\mathbb{I},E, \mathcal{Q},l_{\mathsf{tok}},\tau_{\mathsf{O}})$. Furthermore, the oracle $O_j[\widetilde{a}]$ is in $\mathcal{O}_{\mathbf{call}}(\mathit{th}_{tj})$ and the oracles $O'_i[\widetilde{a}_i]$ that are not under replication are in $\mathit{willbeavailable}(\mathbb{CS})$, so by Property I13 of the invariant, they are not in $\mathcal{O}_{\mathbf{call}}(\mathit{th}_{tj'})$. Hence, $\mathcal{O}_{\mathbf{call}}(\mathit{th}_{tj'}) \cap \mathbb{I}' = \mathcal{O}_{\mathbf{call}}(\mathit{th}_{tj'}) \cap \mathbb{I}$ and $\mathcal{O}_{\mathbf{call}}(\mathit{th}_{tj'}) \setminus \mathbb{I}' = \mathcal{O}_{\mathbf{call}}(\mathit{th}_{tj'}) \setminus \mathbb{I}$, so

$\mathit{gettokens}(\mathbb{I}', \mathcal{O}_{\mathbf{call}}(th_{tj'}), l_{\mathsf{tok}}) = \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(th_{tj'}), l_{\mathsf{tok}})$. So thread tj' continues to verify Property T2(a) with the same function l_{tok}.

Let us now consider the current thread. The current thread of the simulator is $th^{\mathsf{s}}_1 = \langle env^{\mathsf{s}}, pe^{\mathsf{s}}_1, stack^{\mathsf{s}}, store^{\mathsf{s}} \rangle$ where $pe^{\mathsf{s}}_1 \overset{\text{def}}{=} (\mathbf{call}(O'_1[\widetilde{a}_1]), \dots, \mathbf{call}(O'_l[\widetilde{a}_l]), \mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \dots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$ and the current thread on the OCaml side is $th^{\mathsf{o}}_5 = \langle env^{\mathsf{o}}_4, pe^{\mathsf{o}}_6, stack^{\mathsf{o}}, store^{\mathsf{o}}_6 \rangle$ where $pe^{\mathsf{o}}_6 \overset{\text{def}}{=} (c(Q_1, \gamma_1), \dots, c(Q_l, \gamma_l), \mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \dots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$. By Property T2(a), there exist $store^{\mathsf{o}}_7$ and l_{tok} such that

$$\begin{aligned}
&\langle env^{\mathsf{o}}, pe^{\mathsf{o}}, stack^{\mathsf{o}}, store^{\mathsf{o}}_7 \rangle \\
&\qquad \in \mathit{replacecalls}(\mathit{replaceinitpm}(th^{\mathsf{s}}), \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \\
&store^{\mathsf{o}}_7 \cup \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}), l_{\mathsf{tok}}) \subseteq store^{\mathsf{o}}\,.
\end{aligned}$$

Let us denote $th'' \overset{\text{def}}{=} \mathit{replaceinitpm}(th^{\mathsf{s}})$ and $th''_1 \overset{\text{def}}{=} \mathit{replaceinitpm}(th^{\mathsf{s}}_1)$. The thread th''_1 is the thread th'' where the current expression is replaced with pe^{s}_1. Let $th^{\mathsf{s}}_2 \overset{\text{def}}{=} \langle env^{\mathsf{s}}, (), stack^{\mathsf{s}}, store^{\mathsf{s}} \rangle$ 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 $\gamma_i = \mathsf{Once}$, $O'_i[\widetilde{a}_i]$ is in $\mathit{willbeavailable}(\mathbb{CS})$, so by Property O1, $O'_i[\widetilde{a}_i]$ is not in $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}})$, so we can define $l'_{\mathsf{tok}} \overset{\text{def}}{=} l_{\mathsf{tok}|\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_2)} \cup \{O'_i[\widetilde{a}_i] \mapsto l_i \mid \gamma_i = \mathsf{Once}\}$ and l'_{tok} is an extension of $l_{\mathsf{tok}|\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_2)}$. By Lemma F.3, Items 1 and 2, for all $\mathbf{call}(R)$ that occur in $th''_2 \overset{\text{def}}{=} \mathit{replaceinitpm}(th^{\mathsf{s}}_2)$, we have $\mathit{correctclosure}(R, \mathbb{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathit{correctclosure}(R, \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Moreover, $c(Q_i, \gamma_i) \in \mathit{correctclosure}(O'_i[\widetilde{a}_i], \mathbb{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$ for $i \leq l$, and pe^{s}_1 is a value so *replacecalls* allows any environment in the threads it returns, so $\langle env^{\mathsf{o}}_4, pe^{\mathsf{o}}_6, stack^{\mathsf{o}}, store^{\mathsf{o}}_7 \rangle \in \mathit{replacecalls}(th''_1, \mathbb{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$. We have

$$\begin{aligned}
&store^{\mathsf{o}}_7 \cup \mathit{gettokens}(\mathbb{I}', \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_1), l'_{\mathsf{tok}}) \\
&\quad \subseteq store^{\mathsf{o}}_7 \cup \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}), l_{\mathsf{tok}})[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathbf{Invalid}] \\
&\qquad \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} \\
&\quad \subseteq store^{\mathsf{o}}[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathbf{Invalid}] \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} \\
&\quad \subseteq store^{\mathsf{o}}_1[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathbf{Invalid}] \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} \\
&\quad \subseteq store^{\mathsf{o}}_2 \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} \\
&\quad \subseteq store^{\mathsf{o}}_5 \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} = store^{\mathsf{o}}_6\,,
\end{aligned}$$

so Property T2(a) holds with the function l'_{tok}. Properties T2(b), T2(c), T2(d) are preserved, so Property I6 holds.

Properties I7, I8, and I11 are also preserved. Properties I9, I10, I12 hold because they are kept inside Lemma 8.10. The oracles coming from $\mathit{oraclelist}(Q')$ are removed from $\mathit{willbeavailable}(\mathbb{CS})$ and added to $\mathcal{O}^{\infty}(\mathbb{I}) \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS})$. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}^{\infty}(\mathbb{I})$; it is also removed from $\mathcal{O}_{\mathbf{call}}(\mathbb{CS})$ if there remains no occurrence of $\mathbf{call}(O_j[\widetilde{a}])$ in the thread

th_1^{s}. So Property O1 is preserved. The oracles coming from $\mathit{oraclelist}(Q')$ are added to $\mathcal{O}^\infty(\mathbb{I}) \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS})$ and to $\mathcal{O}_{\textbf{call-}\mathsf{repl}}(th_1^{\mathsf{s}})$ or $\mathcal{O}_{\mathbf{call}}(th_1^{\mathsf{s}})$ depending on whether they are under replication or not. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}_{\mathbf{call}}(th_1^{\mathsf{s}})$ if and only if it is removed from $\mathcal{O}^\infty(\mathbb{I}) \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS})$. So Property O2 is preserved, which shows Property I13. We have $|\mathbb{CT}''| + N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property I14 holds. The number of calls to O_j increases by 1 and α increases by 1, so Property I15 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 $\mathbb{I}$; Property I16 is obviously satisfied for these oracles because the number of calls to these oracles is not negative. Property I16 for the previously present oracles and Property I17 are preserved, because all components of these inequalities are unchanged. So $\mathcal{C}_1^{\mathsf{cs}} \equiv \mathbb{CT}''$ in this case.

- If $P_4 = \mathsf{return}(M_1, \dots, M_{m'_j})\}; Q'$, the CryptoVerif process reduces in exactly the same manner as above. The configuration $\mathbb{CS}'$ is the configuration $\mathbb{CS}$ in which we replace the current expression pe^{s} with $(\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \dots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$, the set $\mathbb{I}$ with $\mathbb{I}' \overset{\text{def}}{=} \mathbb{I} \setminus \{O_j[\widetilde{a}]\}$, and the set $\mathbb{RI}$ with $\mathbb{RI}' \overset{\text{def}}{=} \mathbb{RI} \cup \{\mathsf{role}[\widetilde{a}] \mid (\mu_{\mathsf{role}}, \mathsf{Once}) \in \mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q')\} \cup \{\mathsf{role}\big[[1, +\infty[, \widetilde{a}\big] \mid (\mu_{\mathsf{role}}, \mathsf{Any}) \in \mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q')\}$.

  Let th_4^{o} be the current thread of $\mathbb{C}_4$. We have

$$\begin{aligned} th_4^{\mathsf{o}} &= \langle env_4^{\mathsf{o}}, \mathbf{return}(\mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q'), (\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_{m'_j}))), \\ &\qquad stack^{\mathsf{o}}, store_4^{\mathsf{o}}\rangle \\ &\to \langle env_4^{\mathsf{o}}, (\mathbb{G}_{\mathsf{M}}(M_1), \dots, \mathbb{G}_{\mathsf{M}}(M_{m'_j})), stack_1^{\mathsf{o}}, store_4^{\mathsf{o}}\rangle \\ &\qquad \text{where } stack_1^{\mathsf{o}} \overset{\text{def}}{=} (env_4^{\mathsf{o}}, \mathbf{return}(\mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q'), [\cdot])) :: stack^{\mathsf{o}} \\ \to^* th_5^{\mathsf{o}} &\overset{\text{def}}{=} \langle env_4^{\mathsf{o}}, (\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \dots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), stack_1^{\mathsf{o}}, store_5^{\mathsf{o}}\rangle \\ &\qquad \text{by Lemma 8.8 applied } m'_j \text{ times} \end{aligned}$$

  where $store_5^{\mathsf{o}} \supseteq store_4^{\mathsf{o}}$ and $\mathbb{C}_4 \to^* \mathbb{C}_4[\mathsf{th} \mapsto th_5^{\mathsf{o}}] \to^* \mathbb{C}_5 \overset{\text{def}}{=} \mathbb{C}_4[\mathsf{th} \mapsto th_6^{\mathsf{o}}, \mathsf{MI} \mapsto \mathbb{MI}']$ where $th_6^{\mathsf{o}} \overset{\text{def}}{=} \langle env_4^{\mathsf{o}}, (\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \dots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), stack^{\mathsf{o}}, store_5^{\mathsf{o}}\rangle$ and $\mathbb{MI}' \overset{\text{def}}{=} \mathbb{MI} \cup \mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q')$. Let $\mathbb{CT}''$ be the trace $\mathbb{CT}$ followed by $\mathbb{CT}'$ and extended until $\mathbb{C}_5$.

  Let us now prove that $\mathcal{C}_1^{\mathsf{cs}} \equiv \mathbb{CT}''$. By the form of $\mathcal{C}_1^{\mathsf{cs}}$ and $\mathbb{C}''$, Properties I1 and I2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ from which we removed the oracles $O_j[\widetilde{a}]$ and $\mathsf{O}_{\mathsf{loop}}[\alpha + 1]$ and to which we added the new oracles of $\mathit{oracledeflist}(Q')$. The set $\mathbb{I}'$ is the set $\mathbb{I}$ from which we removed $O_j[\widetilde{a}]$. We added the elements of $\mathit{oraclelist}(Q')$ to $\mathcal{O}(\mathbb{RI}')$. So Property I3 is preserved. Properties I4 to I10, I12, and I14 to I17 are proved as in the case $P_4 = \mathsf{end}$. We added matching elements in $\mathbb{MI}'$ and in $\mathbb{RI}'$, so Property I11 is preserved. The oracles coming from $\mathit{oraclelist}(Q')$ are removed from $\mathit{willbeavailable}(\mathbb{CS})$ and added to $\mathcal{O}^\infty(\mathbb{RI})$. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}^\infty(\mathbb{I})$; it is also removed from $\mathcal{O}_{\mathbf{call}}(\mathbb{CS})$ if there remains no occurrence of $\mathbf{call}(O_j[\widetilde{a}])$ in the thread th^{s}_1. So Property O1 is preserved. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}}_1)$ if and only if it is removed from $\mathcal{O}^\infty(\mathbb{I}) \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS})$, and the other oracle sets of Property O2 are unchanged, so Property O2 is preserved, which proves Property I13. So $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}''$ in this case.

Case 2.2. The current expression of $\mathbb{CS}$ is $\mathit{pe}^{\mathsf{s}} = \mathbf{call}(O_j[\_,\widetilde{a}])\ (v_1,\dots,v_{m_j})$ and $\mathbb{CS}$ cannot reduce, that is, the configuration $\mathbb{CS}$ 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 I16, as follows. By Property T2(a), $\mathit{pe}^{\circ} = \mathbf{tagfunction}^{O_j,\tau}[\mathit{env}^{\circ}_1, \mathit{pm}_{\mathsf{Any}}(Q)]\ (v_1, \dots, v_{m_j})$, with $\tau = \tau_{\mathsf{O}}(O_j[\_,\widetilde{a}])$ and $O\big[[a',+\infty[,\widetilde{a}\big] \in \mathbb{I}$ for some a'. Let $\mathbb{CT}'$ be the extension of $\mathbb{CT}$ with one step. By definition of N_{O_j}, we have $N_{O_j} \geq N_{\mathsf{calls}}(O_j,\tau,\mathbb{CT}') = N_{\mathsf{calls}}(O_j,\tau,\mathbb{CT}) + 1$, so by Property I16, $a' \leq N_{O_j}$, so $O_j[a',\widetilde{a}] \in \mathcal{O}(\mathbb{I})$, so by Property I3, $\mathcal{Q}_0$ contains a process of the form $O_j[a',\widetilde{a}](x_1[a',\widetilde{a}] : T_{j,1},\dots,x_{m_j}[a',\widetilde{a}] : T_{j,m_j}) := P_O$.

Due to the call, the index a' such that $O\big[[a',+\infty[,\widetilde{a}\big] \in \mathbb{I}$ increases by 1 and the number of calls to the closure with tag O_j,τ increases by 1, so Property I16 is preserved.

Case 2.3. The current expression of $\mathbb{CS}$ is $\mathit{pe}^{\mathsf{s}} = \mathbf{random}\ ()$, that is, the configuration $\mathbb{CS}$ samples a random boolean. By Property I6, the current expression of $\mathbb{C}$ is $\mathit{pe}^{\circ} = \mathbf{random}\ ()$. For $b \in \{\text{true},\text{false}\}$, $\mathbb{C} \to_{1/2} \mathbb{C}_b$ where $\mathbb{C}_b \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}^{\circ}, \mathbb{G}_{\mathsf{val}\,\mathit{bool}}(b), \mathit{stack}^{\circ}, \mathit{store}^{\circ}\rangle]$. Let $\mathbb{CT}_b$ be the extension of the trace $\mathbb{CT}$ until $\mathbb{C}_b$.

The configuration $\mathbb{CS}$ cannot reduce, and $\mathit{simreturn}(\mathbb{CS}) = (\mathit{repr}(\mathbb{CS}), \mathsf{o_R}, (), ())$. Let us denote $s \overset{\text{def}}{=} \mathit{repr}(\mathbb{CS})$. The simulator configuration reduces in the following way for a CryptoVerif value $b \in \{\text{true},\text{false}\}$.

$$\begin{aligned}
\mathcal{C}^{\mathsf{cs}} &\rightsquigarrow^* E_1, P'_{\mathsf{loop}}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E} \\
&\qquad \text{where } E_1 \overset{\text{def}}{=} E[\mathit{xs}[\alpha] \mapsto (s, \mathsf{o_R}, (), ()), s'[\alpha] \mapsto s, o[\alpha] \mapsto \mathsf{o_R}, \\
&\qquad\qquad\qquad i[\alpha] \mapsto (), \mathit{args}[\alpha] \mapsto ()] \\
&\rightsquigarrow^* E_1, P^R_{\mathsf{loop}}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E} \\
&\rightsquigarrow_{1/2} E_2, \mathsf{return}(\text{simulate}_{\mathsf{R}}(s'[\alpha], b_{\mathsf{R}}[\alpha]), \text{continue}), \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E} \\
&\qquad \text{where } E_2 \overset{\text{def}}{=} E_1[b_{\mathsf{R}}[\alpha] \mapsto b] \\
&\rightsquigarrow^* \mathcal{C}^{\mathsf{cs}}_b \overset{\text{def}}{=} E_3, P_{\mathsf{loop}}\{\alpha+1/i'\}, \mathcal{T}, \mathcal{Q}_1, \mathcal{S}_{\mathsf{loop}}(\alpha+1), \mathcal{E}, N_{\mathsf{steps}}, \mathbb{CS}_b \\
&\qquad \text{where } E_3 \supseteq E_2[s[\alpha+1] \mapsto \mathit{repr}(\mathbb{CS}_b)], \\
&\qquad\qquad \mathcal{Q}_1 \overset{\text{def}}{=} \mathcal{Q} \setminus \{Q_{\mathsf{loop}}\{\alpha+1/i'\}\}, \\
&\qquad\qquad \mathbb{CS}_b \overset{\text{def}}{=} \mathbb{CS}[\mathsf{th} \mapsto \langle \mathit{env}^{\mathsf{s}}, \mathbb{G}_{\mathsf{val}\,\mathit{bool}}(b), \mathit{stack}^{\mathsf{s}}, \mathit{store}^{\mathsf{s}}\rangle]
\end{aligned}$$

We verify that $Q_{\mathsf{loop}}\{\alpha + 1/i'\} \in \mathcal{Q}$ using Property I15, as follows. By definition of $N_{\mathsf{rand+calls}}$, $N_{\mathsf{rand+calls}} \geq \Big(N_{\mathsf{rand}}(\mathbb{CT}_b) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathbb{CT}_b)\Big) + 1 = \Big(N_{\mathsf{rand}}(\mathbb{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathbb{CT})\Big) + 2$ since $\mathbb{CT}_b$ makes one more random number generation than $\mathbb{CT}$. So by Property I15, $N_{\mathsf{rand+calls}} \geq \alpha + 1$. So by Property I3, $Q_{\mathsf{loop}}\{\alpha + 1/i'\} \in \mathcal{Q}$.

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 $\mathcal{C}^{\mathsf{cs}}_{\mathrm{true}} \equiv \mathbb{CT}_{\mathrm{true}}$ and $\mathcal{C}^{\mathsf{cs}}_{\mathrm{false}} \equiv \mathbb{CT}_{\mathrm{false}}$.

Case 2.4. The configuration $\mathbb{CS}$ does not reduce, and does not make a call to an oracle nor sample a random boolean. In this case, $\mathit{simreturn}(\mathbb{CS}) = (\mathit{repr}(\mathbb{CS}), \mathrm{o}_{\mathsf{S}}, (), ())$. Let us denote $s \overset{\mathrm{def}}{=} \mathit{repr}(\mathbb{CS})$. The simulator configuration reduces in the following way.

$$
\begin{aligned}
\mathcal{C}^{\mathsf{cs}} \rightsquigarrow^* \; & E_1, P'_{\mathsf{loop}}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E} \\
& \quad \text{where } E_1 \overset{\mathrm{def}}{=} E[\mathit{xs}[\alpha] \mapsto (s, \mathrm{o}_{\mathsf{S}}, (), ()), s'[\alpha] \mapsto s, o[\alpha] \mapsto \mathrm{o}_{\mathsf{S}}, i[\alpha] \mapsto (), \\
& \qquad\qquad\qquad \mathit{args}[\alpha] \mapsto ()] \\
\rightsquigarrow \; & E_1, \mathsf{return}(s'[\alpha], \mathrm{stop}), \mathcal{T}, \mathcal{Q}, \mathcal{S}_{\mathsf{loop}}(\alpha), \mathcal{E} \\
\rightsquigarrow \; & E_2, P_{\mathsf{return\text{-}loop}}(\alpha), \mathcal{T}, \mathcal{Q}, \mathcal{S}_1, \mathcal{E} \\
& \quad \text{where } E_2 \overset{\mathrm{def}}{=} E_1[r'_{\alpha,r}[\,] \mapsto s, b_{\alpha,r}[\,] \mapsto \mathrm{stop}], \\
& \qquad\quad \mathcal{S}_1 \overset{\mathrm{def}}{=} [x[\,], \mathsf{return}(x[\,]), \mathsf{end}] \\
\rightsquigarrow \; & E_2, r[\,] \leftarrow r'_{\alpha,r}[\,]; \mathsf{end}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_1, \mathcal{E} \\
\rightsquigarrow \; & E_3, \mathsf{end}, \mathcal{T}, \mathcal{Q}, \mathcal{S}_1, \mathcal{E} \\
& \quad \text{where } E_3 \overset{\mathrm{def}}{=} E_2[r[\,] \mapsto s] \\
\rightsquigarrow \; & \mathcal{C}^{\mathsf{cs}}_1 \overset{\mathrm{def}}{=} E_3, \mathsf{end}, \mathcal{T}, \mathcal{Q}, [\,], \mathcal{E}
\end{aligned}
$$

This configuration cannot reduce. By Property I6, the OCaml configuration $\mathbb{C}$ also cannot reduce. (If it could reduce, then the simulator configuration $\mathbb{CS}$ would reduce by the same rule as the OCaml configuration.) Moreover, by Property I12 of the invariant, $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$, so this case satisfies the second point of Lemma 8.35.

Case 2.5. The current expression of $\mathbb{CS}$ is $\mathit{pe}^{\mathsf{s}} = \textbf{tagfunction}^{t,\tau}[\mathit{env}, \mathit{pm}]\ v$, that is, $\mathbb{CS}$ calls a tagged closure. By Property T2(c), the tagged closures present in the current thread are of the form $\textbf{tagfunction}^{\mathsf{role},\tau}[\mathit{env}^{\mathsf{s}}_1, \mathit{pm}'_{\mathsf{role}[\widetilde{a}]}]$ for a given role $\mathsf{role}[\widetilde{a}]$, with $\mathit{env}_{\mathsf{prim}} \subseteq \mathit{env}^{\mathsf{s}}_1$. 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 $\mathbb{CS}$ is $\mathit{pe}^{\mathsf{s}} = \textbf{tagfunction}^{\mathsf{role},\tau}[\mathit{env}^{\mathsf{s}}_1, \mathit{pm}'_{\mathsf{role}[\widetilde{a}]}]\ ()$.

Let Q_i, γ_i for $i \leq m$ be the oracles present in $\mathit{oracledeflist}(Q(\mathsf{role})[\widetilde{a}])$, and let $\widetilde{a_i} = \widetilde{a}$ or $\_, \widetilde{a}$ such that $O'_i[\widetilde{a_i}]$ is the oracle associated to Q_i, γ_i in $\mathit{oraclelist}(Q(\mathsf{role})[\widetilde{a}])$.

By Property T2(a),

$$pe^{\mathsf{o}} = \mathbf{tagfunction}^{\mathsf{role},\tau}[env_1^{\mathsf{o}}, pm_{\mu_{\mathsf{role}}}]\ ()\,.$$

By Property T2(b), $env_1^{\mathsf{s}}(token) = l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}])$ is a location. Let us denote l this location. By Property T2(a), we have $env_1^{\mathsf{o}}(token) = env_1^{\mathsf{s}}(token) = l$. By Property I5, l is in the domain of $store^{\mathsf{s}}$. By Property T2(a), l is also in the domain of $store^{\mathsf{o}}$.

Let $x_1[\,], \dots, x_k[\,]$ be the free variables of the role role.

Let us denote

$$\begin{aligned} pe_e^{\mathsf{s}} &\overset{\text{def}}{=} (\mathbf{call}(O'_1[\widetilde{a_1}]), \dots, \mathbf{call}(O'_m[\widetilde{a_m}]))\\ pe_e^{\mathsf{o}} &\overset{\text{def}}{=} \mathbb{G}_{\mathsf{read}}(x_1[\,])\ \mathbf{in}\ \dots\ \mathbf{in}\ \mathbb{G}_{\mathsf{read}}(x_k[\,])\ \mathbf{in}\\ &\qquad (\mathbb{G}_{\mathsf{O}}(Q_1,\gamma_1), \dots, \mathbb{G}_{\mathsf{O}}(Q_m,\gamma_m)) \end{aligned}$$

The simulator reduces as follows:

$$\begin{aligned} th^{\mathsf{s}} &= \langle env^{\mathsf{s}}, \mathbf{tagfunction}^{\mathsf{role},\tau}[env_1^{\mathsf{s}}, pm'_{\mathsf{role}}]\ (), stack^{\mathsf{s}}, store^{\mathsf{s}}\rangle\\ &\to \langle env_1^{\mathsf{s}}, \mathbf{match}\ ()\ \mathbf{with}\ pm'_{\mathsf{role}}, stack^{\mathsf{s}}, store^{\mathsf{s}}\rangle\\ \to th_1^{\mathsf{s}} &\overset{\text{def}}{=} \langle env_1^{\mathsf{s}}, pe_1^{\mathsf{s}}, stack^{\mathsf{s}}, store^{\mathsf{s}}\rangle\\ &\quad\text{where } pe_1^{\mathsf{s}} \overset{\text{def}}{=} \mathbf{if}\ !token = \mathbf{Callable}\ \mathbf{then}\ (token := \mathbf{Invalid}; pe_e^{\mathsf{s}})\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ raise\ Bad\_Call} \end{aligned}$$

and the OCaml side reduces as follows:

$$\begin{aligned} th^{\mathsf{o}} &= \langle env^{\mathsf{o}}, \mathbf{tagfunction}^{\mathsf{role},\tau}[env_1^{\mathsf{o}}, pm_{\mu_{\mathsf{role}}}]\ (), stack^{\mathsf{o}}, store^{\mathsf{o}}\rangle\\ &\to \langle env_1^{\mathsf{o}}, \mathbf{match}\ ()\ \mathbf{with}\ pm_{\mu_{\mathsf{role}}}, stack^{\mathsf{o}}, store^{\mathsf{o}}\rangle\\ \to th_1^{\mathsf{o}} &\overset{\text{def}}{=} \langle env_1^{\mathsf{o}}, pe_1^{\mathsf{o}}, stack^{\mathsf{o}}, store^{\mathsf{o}}\rangle\\ &\quad\text{where } pe_1^{\mathsf{o}} \overset{\text{def}}{=} \mathbf{if}\ !token = \mathbf{Callable}\ \mathbf{then}\ (token := \mathbf{Invalid}; pe_e^{\mathsf{o}})\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ raise\ Bad\_Call} \end{aligned}$$

- If $store^{\mathsf{s}}(l) = \mathbf{Invalid}$, then by Property T2(a), $store^{\mathsf{o}}(l) = \mathbf{Invalid}$, so

$$\begin{aligned} th_1^{\mathsf{s}} \to^* th_2^{\mathsf{s}} &\overset{\text{def}}{=} \langle env_1^{\mathsf{s}}, \mathbf{raise\ Bad\_Call}, stack^{\mathsf{s}}, store^{\mathsf{s}}\rangle\\ th_1^{\mathsf{o}} \to^* th_2^{\mathsf{o}} &\overset{\text{def}}{=} \langle env_1^{\mathsf{o}}, \mathbf{raise\ Bad\_Call}, stack^{\mathsf{s}}, store^{\mathsf{s}}\rangle \end{aligned}$$

  Let $\mathbb{CT}_1$ be the extension of the trace $\mathbb{CT}$ until $\mathbb{C}[\mathsf{th} \mapsto th_2^{\mathsf{o}}]$, $\mathbb{CS}_1 \overset{\text{def}}{=} \mathbb{CS}[\mathsf{th} \mapsto th_2^{\mathsf{s}}]$, $steps^s$ the number of steps of the trace $\mathbb{CS} \to^* \mathbb{CS}_1$, and $\mathcal{C}_1^{\mathsf{cs}} \overset{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, steps - steps^s, \mathbb{CS}_1$. We have $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathcal{C}_1^{\mathsf{cs}}$.

  Let us prove that $\mathcal{C}_1^{\mathsf{cs}} \equiv \mathbb{CT}_1$. 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 $\mathbb{I}$ did not change, so Property I6 is preserved. The number of steps in the reduction is the same on both sides, so Property I14 is preserved. All other properties of Definition 8.32 are trivially inherited from $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$.

- Otherwise, $\mathit{store}^{\mathsf{s}}(l) = \textbf{Callable}$. By Property T2(a), $\mathit{store}^{\mathsf{o}}(l) = \textbf{Callable}$.

  On the simulator side:

$$\begin{aligned} \mathit{th}^{\mathsf{s}}_1 \rightarrow^* \mathit{th}^{\mathsf{s}}_3 &\overset{\text{def}}{=} \langle \mathit{env}^{\mathsf{s}}_1, \mathit{pe}^{\mathsf{s}}_e, \mathit{stack}^{\mathsf{s}}, \mathit{store}^{\mathsf{s}}_1 \rangle \\ &\quad \text{where } \mathit{store}^{\mathsf{s}}_1 \overset{\text{def}}{=} \mathit{store}^{\mathsf{s}}[l \mapsto \textbf{Invalid}] \end{aligned}$$

  By Property I4, the variables $x_1[\,], \dots, x_k[\,]$ are present in the environment E. Let $a'_1, \dots, a'_k$ be the values of these variables in the environment E. By Property I9, $\mathit{globalstore}(E, \mathcal{T}) \subseteq \mathit{globalstore}^o$, so $\mathit{globalstore}^o(f_i) = \mathit{ser}(T_{x_i}, a'_i)$ where $(x_i[\,], f_i) \in \mathit{Files}$ for all $i \leq k$. Let $\mathbb{C}_1 \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \mathit{th}^{\mathsf{o}}_1]$. We have

$$\begin{aligned} \mathbb{C}_1 &\rightarrow^* \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}^{\mathsf{o}}_1, \mathit{pe}^{\mathsf{o}}_e, \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}}_1 \rangle] \\ &\qquad \text{where } \mathit{store}^{\mathsf{o}}_1 \overset{\text{def}}{=} \mathit{store}^{\mathsf{o}}[l \mapsto \textbf{Invalid}] \\ &\rightarrow^* \mathbb{C}_2 \overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}^{\mathsf{o}}_2, \mathit{pe}^{\mathsf{o}}_2, \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}}_2 \rangle] \\ &\qquad \text{where } \mathit{pe}^{\mathsf{o}}_2 \overset{\text{def}}{=} (\mathbb{G}_{\mathsf{O}}(Q_1, \gamma_1), \dots, \mathbb{G}_{\mathsf{O}}(Q_m, \gamma_m)), \\ &\qquad\qquad \mathit{env}^{\mathsf{o}}_2 \overset{\text{def}}{=} \mathit{env}^{\mathsf{o}}_1[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_{x_1}}(a'_1), \dots, \\ &\qquad\qquad\qquad \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_{x_k}}(a'_k)], \\ &\qquad\qquad \mathit{store}^{\mathsf{o}}_2 \supseteq \mathit{store}^{\mathsf{o}}_1 \end{aligned}$$

  by Proposition 8.5 applied k times to show the correctness of the deserialization primitives.

  Let $l_1, \dots, l_m$ be pairwise distinct locations that are not in $\mathit{Dom}(\mathit{store}^{\mathsf{o}}_2)$ and $\tau_1, \dots, \tau_m$ be pairwise distinct fresh tags. By the same reasoning as in Case 2.1, sub-case $P_4 = \mathsf{return}(M_1, \dots, M_{m'_j}); Q'$, we have

$$\begin{aligned} \mathbb{C}_2 \rightarrow^* \mathbb{C}_3 &\overset{\text{def}}{=} \mathbb{C}[\mathsf{th} \mapsto \langle \mathit{env}^{\mathsf{o}}_2, \mathit{pe}^{\mathsf{o}}_3, \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}}_3 \rangle] \\ &\quad \text{where } \mathit{pe}^{\mathsf{o}}_3 \overset{\text{def}}{=} (\textbf{tagfunction}^{O'_1, \tau_1}[\mathit{env}^{\mathsf{o}}_{c,1}, \mathit{pm}_{\gamma_1}(Q_1)], \dots, \\ &\qquad\qquad \textbf{tagfunction}^{O'_m, \tau_m}[\mathit{env}^{\mathsf{o}}_{c,m}, \mathit{pm}_{\gamma_m}(Q_m)]), \\ &\qquad \mathit{store}^{\mathsf{o}}_3 \overset{\text{def}}{=} \mathit{store}^{\mathsf{o}}_2 \cup \{ l_i \mapsto \textbf{Callable} \mid i \leq m, \gamma_i = \mathsf{Once} \} \end{aligned}$$

  where, for all $i \leq m$, $\mathit{env}^{\mathsf{o}}_{c,i}$ is $\mathit{env}^{\mathsf{o}}_2$ when γ_i is Any and $\mathit{env}^{\mathsf{o}}_2[\mathit{token} \mapsto l_i]$ otherwise.

  Let $\mathbb{CT}_2$ be an extension of the trace $\mathbb{CT}$ until $\mathbb{C}_3$. Let $\mathbb{CS}_3 \overset{\text{def}}{=} \mathbb{CS}[\mathsf{th} \mapsto \mathit{th}^{\mathsf{s}}_3]$. Let steps^s be the number of steps of $\mathbb{CS} \rightarrow^* \mathbb{CS}_3$. Let $\mathcal{C}^{\mathsf{cs}}_2 \overset{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathit{steps} - \mathit{steps}^s, \mathbb{CS}_3$. We have $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathcal{C}^{\mathsf{cs}}_2$.

Let us prove that $\mathcal{C}_2^{\mathsf{cs}} \equiv \mathbb{CT}_2$. We define τ'_{O} as τ_{O} except that for all $i \leq m$, if $\gamma_i = \mathsf{Any}$, then $\tau'_{\mathsf{O}}(O'_i[\_, \widetilde{a}]) = \tau_i$. Properties I1, I2, I3, I4, I5, I7, I8, I9, I10, I11, I12, I15, I17 hold because they hold for $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$.

Let us prove Property I6. First, we prove Property T2 for the current thread. For all $i \leq m$, the free variables of Q_i are contained in $\{x_1[], \dots, x_k[]\}$, so $\mathit{env}^{\circ}_{c,i} \supseteq \mathit{env}(E, Q_i)$. Moreover, by Properties T2(c) and T2(a), $\mathit{env}_{\mathsf{prim}} \subseteq \mathit{env}^{\circ}_1$, so $\mathit{env}^{\circ}_{c,i} \supseteq \mathit{env}^{\circ}_2 \supseteq \mathit{env}^{\circ}_1 \supseteq \mathit{env}_{\mathsf{prim}}$. We have $\mathsf{role}[\widetilde{a}] \in \mathcal{R}_{\mathsf{init\text{-}closure}}(\mathit{th}^{\mathsf{s}})$. By Property O2, $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathit{th}^{\mathsf{s}}))$ is included in $\mathcal{O}^{\infty}(\mathbb{I}) \cup \mathcal{O}_{\mathbf{call}}(\mathbb{CS})$, and furthermore $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathit{th}^{\mathsf{s}}))$ is disjoint from $\mathcal{O}_{\mathbf{call}}(\mathit{th}_i)$ for all $i \leq n$, so from $\mathcal{O}_{\mathbf{call}}(\mathbb{CS})$, so $\mathcal{O}^{\infty}(\{\mathsf{role}[\widetilde{a}]\})$ is included in $\mathcal{O}^{\infty}(\mathbb{I})$. Hence, when O'_i is not under replication (that is, $\gamma_i = \mathsf{Once}$), $O'_i[\widetilde{a_i}] \in \mathbb{I}$, and when O'_i is under replication, $\widetilde{a_i} = \_, \widetilde{a}$ and $O'_i\big[[1, +\infty[, \widetilde{a}\big] \in \mathbb{I}$. By Property I3, 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 T2(a), there exist store°_4 and l_{tok} such that

$$\begin{aligned}
&\langle \mathit{env}^{\circ}, \mathit{pe}^{\circ}, \mathit{stack}^{\circ}, \mathit{store}^{\circ}_4 \rangle \\
&\qquad \in \mathit{replacecalls}(\mathit{replaceinitpm}(\mathit{th}^{\mathsf{s}}), \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \\
&\mathit{store}^{\circ}_4 \cup \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(\mathit{th}^{\mathsf{s}}), l_{\mathsf{tok}}) \subseteq \mathit{store}^{\circ}\,.
\end{aligned}$$

Since $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathit{th}^{\mathsf{s}}))$ is disjoint from $\mathcal{O}_{\mathbf{call}}(\mathbb{CS})$ as noticed above, the oracles $O'_i[\widetilde{a_i}]$ are not present in $\mathcal{O}_{\mathbf{call}}(\mathbb{CS})$. So we can define the injective function $l'_{\mathsf{tok}} \overset{\text{def}}{=} l_{\mathsf{tok}} \cup \{O'_i[\widetilde{a_i}] \mapsto l_i \mid i \leq m, \gamma_i = \mathsf{Once}\}$. By Lemma F.3, Item 2, for all $\mathbf{call}(R)$ that occur in $\mathit{replaceinitpm}(\mathit{th}^{\mathsf{s}})$, $\mathit{correctclosure}(R, \mathbb{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathit{correctclosure}(R, \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, noticing that, when $i \leq m$ and $\gamma_i = \mathsf{Any}$, $O'_i[N_{O'_i} + 1, \widetilde{a}] \in \mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathit{th}^{\mathsf{s}}))$, so by Property O2, $O'_i[N_{O'_i} + 1, \widetilde{a}] \notin \mathcal{O}_{\mathbf{call\text{-}repl}}(\mathit{th}^{\mathsf{s}})$, so $\mathbf{call}(O'_i[\_, \widetilde{a}])$ does not occur in $\mathit{replaceinitpm}(\mathit{th}^{\mathsf{s}})$, so the transformation of τ_{O} into τ'_{O} does not affect the computation of these correct closures. Moreover, $\mathbf{tagfunction}^{O'_i, \tau_i}[\mathit{env}^{\circ}_{c,i}, \mathit{pm}_{\gamma_i}(Q_i)] \in \mathit{correctclosure}(O'_i[\widetilde{a_i}], \mathbb{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$ for $i \leq m$ and $\mathit{pe}^{\mathsf{s}}_e$ is a value so $\mathit{replacecalls}$ allows any environment in the threads it returns, so $\langle \mathit{env}^{\circ}_2, \mathit{pe}^{\circ}_3, \mathit{stack}^{\circ}, \mathit{store}^{\circ}_4[l \mapsto \mathbf{Invalid}] \rangle \in \mathit{replacecalls}(\mathit{replaceinitpm}(\mathit{th}^{\mathsf{s}}_3), \mathbb{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$. We have

$$\begin{aligned}
&\mathit{store}^{\circ}_4[l \mapsto \mathbf{Invalid}] \cup \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(\mathit{th}^{\mathsf{s}}_3), l'_{\mathsf{tok}}) \\
&\quad \subseteq \mathit{store}^{\circ}_4[l \mapsto \mathbf{Invalid}] \cup \mathit{gettokens}(\mathbb{I}, \mathcal{O}_{\mathbf{call}}(\mathit{th}^{\mathsf{s}}), l_{\mathsf{tok}}) \\
&\qquad \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} \\
&\quad \subseteq \mathit{store}^{\circ}[l \mapsto \mathbf{Invalid}] \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} \\
&\quad \subseteq \mathit{store}^{\circ}_1 \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} \\
&\quad \subseteq \mathit{store}^{\circ}_2 \cup \{l_i \mapsto \mathbf{Callable} \mid \gamma_i = \mathsf{Once}\} = \mathit{store}^{\circ}_3\,,
\end{aligned}$$

so Property T2(a) holds with the function l'_{tok}. Properties T2(b), T2(c), T2(d) are preserved, so Property I6 holds for the current thread. The other threads and $\mathbb{I}, E, \mathcal{Q}$ 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 I6 holds for all threads.

The role $\mathsf{role}[\widetilde{a}]$ is removed from $\mathcal{R}_{\mathsf{init\text{-}closure}}(th^{\mathsf{s}})$, so the elements added to $\mathcal{O}_{\mathbf{call}}(th^{\mathsf{s}})$ and $\mathcal{O}_{\mathbf{call\text{-}repl}}(th^{\mathsf{s}})$ are removed from $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(th^{\mathsf{s}}))$, hence Property I13 is preserved. There are more steps on the OCaml side than on the CryptoVerif side, so Property I14 is preserved. For the oracles $O'_i[\widetilde{a_i}]$ $(i \leq l)$, when O'_i is under replication, we have already shown that $O'_i\big[[1, +\infty[, \widetilde{a}\big] \in \mathbb{I}$; Property I16 is obviously satisfied for these oracles because the number of calls to these oracles is not negative. Property I16 is preserved for the other oracles, because all components of these inequalities are unchanged.

Case 2.6. The current expression of $\mathbb{CS}$ is $pe^{\mathsf{s}} = \mathbf{addthread}(\mathit{program})$, that is, we add a new thread to the current configuration. By Property T2(a), the expression pe^{o} is $\mathbf{addthread}(\mathit{program})$, and by Property T2(d), $\mathit{program}$ contains no closure, no tagged function, no event, no return except in parts $\mathit{program}(\mu_{\mathsf{role}})$, and in $\mathit{program}(\mu_{\mathsf{role}})$ in arguments of $\mathbf{addthread}$.

Suppose first that $\mathit{program}$ is an attacker program: it does not contain $\mathit{program}(\mu_{\mathsf{role}})$ except in arguments of $\mathbf{addthread}$. In this case,

$$\begin{aligned}\mathbb{CS} \to \mathbb{CS}_1 &\overset{\text{def}}{=} ([th_1, \dots, th_{tj-1}, \langle env^{\mathsf{s}}, (), stack^{\mathsf{s}}, store^{\mathsf{s}}\rangle, th_{tj+1}, \dots, th_n,\\ &\qquad \langle \emptyset, \mathit{program}, [\,], \emptyset\rangle], globalstore^{s}, tj), \mathbb{RI}, \mathbb{I}\\ \mathbb{C} \to \mathbb{C}_1 &\overset{\text{def}}{=} [th'_1, \dots, th'_{tj-1}, \langle env^{\mathsf{o}}, (), stack^{\mathsf{o}}, store^{\mathsf{o}}\rangle, th_{tj+1}, \dots, th_n,\\ &\qquad \langle \emptyset, \mathit{program}, [\,], \emptyset\rangle], globalstore^{o}, tj, \mathbb{MI}, events\end{aligned}$$

Let $\mathbb{CT}_1$ be the extension of the trace $\mathbb{CT}$ until $\mathbb{C}_1$ and $\mathcal{C}^{\mathsf{cs}}_1 \overset{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, steps - 1, \mathbb{CS}_1$. We have $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow \mathcal{C}^{\mathsf{cs}}_1$. Let us prove that $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}_1$. The new thread contains no closures and no tagged functions. It contains no **call** since $\mathit{program}$ is an OCaml program (not a simulator program), so it satisfies Property T2. The other properties are inherited from $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$.

Otherwise, the program $\mathit{program}$ is of the form

$$\mathit{program}_{\mathsf{prim}};;\ \mathit{program}(\mu_{\mathsf{role}_1});;\ \dots;;\ \mathit{program}(\mu_{\mathsf{role}_m});;\ \mathit{program}'\,,$$

where $\mathit{program}'$ does not contain $\mathit{program}(\mu)$ for any $\mu \in \mathbb{M}_{\mathsf{g}}$. By Assumption 6.1, for $\mathbb{M} \overset{\text{def}}{=} \{\mu_{\mathsf{role}_1}, \dots, \mu_{\mathsf{role}_m}\}$, we have $\mathbb{M} \subseteq \mathbb{M}_{\mathsf{g}}$ and $\forall \mu \in \mathbb{M}, \exists \gamma, (\mu, \gamma) \in \mathbb{MI}$. By Property I11, for each $i \leq m$, if role_i is not under replication, then the set $\mathbb{RI}$ contains $\mathsf{role}_i[\widetilde{a}]$ for some $\widetilde{a}$, and if role_i is under replication, then the set $\mathbb{RI}$ contains $\mathsf{role}_i\big[[a', +\infty[, \widetilde{a}\big]$ for some $a', \widetilde{a}$. By Property O1, the oracles present in $\mathbb{RI}$ are not in $\mathbb{I}$. We can then define

$$\begin{aligned}
\widetilde{a_1} &\overset{\text{def}}{=} \mathit{smallest}(\mathbb{RI}, \mathsf{role}_1), \dots, \widetilde{a_m} \overset{\text{def}}{=} \mathit{smallest}(\mathbb{RI}, \mathsf{role}_m)\\
\mathbb{RI}'' &\overset{\text{def}}{=} \{\mathsf{role}_1[\widetilde{a_1}], \dots, \mathsf{role}_m[\widetilde{a_m}]\}\\
\mathbb{RI}' &\overset{\text{def}}{=} \mathbb{RI} - \mathbb{RI}'' \qquad \mathbb{I}' \overset{\text{def}}{=} \mathit{add}(\mathbb{I}, \mathbb{RI}'')\\
\mathit{program}^b &\overset{\text{def}}{=} \mathit{program}_{\mathsf{prim}};; \mathit{program}'(\mathsf{role}_1[\widetilde{a_1}]);; \dots;; \mathit{program}'(\mathsf{role}_m[\widetilde{a_m}]);;\\
&\qquad \mathit{program}'\\
\mathbb{MI}' &\overset{\text{def}}{=} \{(\mu, \mathsf{Once}) \mid \mu \in \mathbb{M} \wedge (\mu, \mathsf{Once}) \in \mathbb{MI}\}
\end{aligned}$$

We have

$$\begin{aligned}
\mathbb{CS} \to \mathbb{CS}_2 \overset{\text{def}}{=} {}& ([\mathit{th}_1, \dots, \mathit{th}_{tj-1}, \langle \mathit{env}^{\mathsf{s}}, (), \mathit{stack}^{\mathsf{s}}, \mathit{store}^{\mathsf{s}}\rangle, \mathit{th}_{tj+1}, \dots, \mathit{th}_n,\\
&\langle \emptyset, \mathit{program}^b, [\,], \emptyset\rangle], \mathit{globalstore}^s, \mathit{tj}), \mathbb{RI}', \mathbb{I}'\\
\mathbb{C} \to \mathbb{C}_2 \overset{\text{def}}{=} {}& [\mathit{th}'_1, \dots, \mathit{th}'_{tj-1}, \langle \mathit{env}^{\mathsf{o}}, (), \mathit{stack}^{\mathsf{o}}, \mathit{store}^{\mathsf{o}}\rangle, \mathit{th}_{tj+1}, \dots, \mathit{th}_n,\\
&\langle \emptyset, \mathit{program}, [\,], \emptyset\rangle], \mathit{globalstore}^o, \mathit{tj}, \mathbb{MI} \setminus \mathbb{MI}', \mathit{events}
\end{aligned}$$

Let $\mathbb{CT}_2$ be the extension of the trace $\mathbb{CT}$ until $\mathbb{C}_2$ and $\mathcal{C}_2^{\mathsf{cs}} \overset{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathit{steps} - 1, \mathbb{CS}_2$. We have $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow \mathcal{C}_2^{\mathsf{cs}}$. Let us prove that $\mathcal{C}_2^{\mathsf{cs}} \equiv \mathbb{CT}_2$. The oracles under replication added to $\mathbb{I}$ are the oracles $O\big[[1, +\infty[, \widetilde{a_i}\big]$ such that $O[\_, \widetilde{a_i}] \in \mathit{oraclelist}(Q(\mathsf{role}_i)[\widetilde{a_i}])$ for any $i \le m$. 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 I1, I2, I4, I8, I9, I10, I12, I14, and I15 are inherited from $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$. By Property I3, $\mathcal{Q} = \{Q_{\mathsf{loop}}\{a/i'\} \mid \alpha < a \le N_{\mathsf{rand+calls}}\} \cup \mathcal{Q}_0$ and $\mathcal{Q}_0 \leftrightarrow \mathbb{RI}, \mathbb{I}$. If role_i is under replication, then by definition of *smallest*, $\mathsf{role}_i\big[[a'_i, +\infty[, \widetilde{a''_i}\big] \in \mathbb{RI}$ with $\widetilde{a_i} = a'_i, \widetilde{a''_i}$. By definition of N_{role_i}, $N_{\mathsf{role}_i} \ge N_{\mathsf{exec}}(\mathsf{role}_i, \mathbb{CT}_2) = N_{\mathsf{exec}}(\mathsf{role}_i, \mathbb{CT}) + 1$. By Property I17, $a'_i \le N_{\mathsf{role}_i}$. Therefore, the set $\mathcal{O}(\mathbb{RI})$ contains the first oracles of $\mathsf{role}_i[\widetilde{a_i}]$ for $i \le m$. The set $\mathcal{O}(\mathbb{RI}')$ is the set $\mathcal{O}(\mathbb{RI})$ from which we remove the first oracles of $\mathsf{role}_i[\widetilde{a_i}]$ for $i \le m$ and $\mathcal{O}(\mathbb{I}')$ is the set $\mathcal{O}(\mathbb{I})$ to which we add these oracles. So $\mathcal{Q}_0 \leftrightarrow \mathbb{RI}', \mathbb{I}'$ and Property I3 holds. There are no local store locations in *program*, so Property I5 holds. For each thread th_i of the simulator except the new thread, let us show that Property I6 is preserved. The only changes are that the current expression is replaced with $()$ and that $\mathbb{I}' = \mathit{add}(\mathbb{I}, \mathbb{RI}'')$, so we just have to show that Property T2(a) is preserved; the other elements of Property I6 are obviously preserved. By Lemma F.3, Item 2, the correct closures are preserved. By Property O2, the set $\mathcal{O}_{\mathbf{call}}(\mathit{th}_i)$ does not contain any of the oracles added to $\mathbb{I}$, so the tokens are preserved. Hence, Property T2(a) is preserved. Since $\mathit{program}'$ already occurs in the initial program, it does not contain closures. By Property T2(d), it does not contain tagged functions, events, or returns, except in $\mathit{program}(\mu_{\mathsf{role}})$ in arguments of **addthread**, so Property T1 holds for the new thread, which implies Property I6. By Property I7, $\mathit{program}'$ does not contain any location $l \in \mathit{Loc}_{\mathsf{priv}}$ except in $\mathit{program}(\mu_{\mathsf{role}})$ in arguments of **addthread**, so Property I7 holds. When role_i is not under replication, we remove one copy of the module μ_{role_i} from the multiset $\mathbb{MI}$, and correspondingly, we remove $\mathsf{role}_i[\widetilde{a_i}]$ from $\mathbb{RI}$. When role_i is under replication, we add 1 to the first index of roles $\mathsf{role}_i[\widetilde{a_i}]$ in $\mathbb{RI}$, and $\mathbb{MI}$ is not affected by this change. (The role role_i can still be called.) So Property I11 is preserved. The first oracles of $\mathsf{role}_1[\widetilde{a_1}], \dots, \mathsf{role}_m[\widetilde{a_m}]$ are transferred from $\mathcal{O}^\infty(\mathbb{RI})$ to $\mathcal{O}^\infty(\mathbb{I})$, so Property O1 is preserved. More precisely, these oracles are added to $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(th_{n+1}))$, where $th_{n+1} \overset{\text{def}}{=} \langle \emptyset, \mathit{program}^b, [\,], \emptyset \rangle$ is the new thread, so Property O2 is preserved, which proves Property I13. For the oracles $O\big[[1, +\infty[, \widetilde{a_i}\big]$ added to $\mathbb{I}$, Property I16 is obviously satisfied because the number of calls to an oracle is not negative. Property I16 is preserved for the previously present oracles, because all components of these inequalities are unchanged. For the roles $\mathsf{role}_i\big[[a'_i, +\infty[, \widetilde{a''_i}\big] \in \mathbb{RI}$, with $\widetilde{a_i} = a'_i, \widetilde{a''_i}$, we have $\mathsf{role}_i\big[[a'_i + 1, +\infty[, \widetilde{a''_i}\big] \in \mathbb{RI}$; the elements $\mathsf{role}_i[\dots]$ in $\mathbb{RI}$ with indices that do not end with $\widetilde{a''_i}$ are unchanged; and $N_{\mathsf{exec}}(\mathsf{role}_i, \mathbb{CT})$ increases by 1, so Property I17 is preserved for the roles $\mathsf{role}_1, \dots, \mathsf{role}_m$. Property I17 is preserved for the other roles, because all components of the inequalities are unchanged. Therefore, $\mathcal{C}_2^{\mathsf{cs}} \equiv \mathbb{CT}_2$.

Case 2.7. The current expression of $\mathbb{CS}$ is of the form $pe^{\mathsf{s}} = \mathbf{call}(O[\widetilde{a}])\ v$ and $\mathbb{CS}$ reduces, that is, we call an oracle but the call fails. By reduction rule (FailedCall1) or (FailedCall2),

$$th^{\mathsf{s}} \to th_1^{\mathsf{s}} \overset{\text{def}}{=} \langle env^{\mathsf{s}}, \mathbf{raise\ Bad\_Call}, stack^{\mathsf{s}}, store^{\mathsf{s}} \rangle,$$

and $\mathbb{CS} \to \mathbb{CS}_1 \overset{\text{def}}{=} \mathbb{CS}[\mathsf{th} \mapsto th_1^{\mathsf{s}}]$. By Property T2(a), $pe^{\circ} = c\ v'$, where $c \in \mathit{correctclosure}(O[\widetilde{a}], \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

We suppose that the program is well typed, so the value v is a k-tuple $(v_1, \dots, v_k)$, where k is the number of arguments of oracle O. Let $T_1, \dots, T_k$ be their CryptoVerif types. Let $x_1, \dots, x_k$ be the CryptoVerif variables that are the arguments of O. By Assumption 8.3, 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 $O\big[[a'', +\infty[, \widetilde{a'}\big] \in \mathbb{I}$, because otherwise we would have $\mathit{correctclosure}(O[\widetilde{a}], \mathbb{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \emptyset$. The closure c is of the form $\mathbf{tagfunction}^{O,\tau}[env_1^{\circ}, pm_{\mathsf{Any}}(Q)]$. Let $\mathbb{CT}'$ be the extension of $\mathbb{CT}$ by one step. By definition of N_O, $N_O \geq N_{\mathsf{calls}}(O, \tau, \mathbb{CT}') = N_{\mathsf{calls}}(O, \tau, \mathbb{CT}) + 1$. Hence, by Property (I16), $a'' \leq N_O$. Therefore, by definition of *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 8.5, for any environment *env*, stack *stack* and store *store*,

$$\begin{aligned} &\langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{pred}}(T_i))\ v_i, stack, store \rangle \\ &\to^* \langle env', \mathbf{false}, stack, store' \rangle \text{ where } store' \supseteq store \end{aligned}$$

So,

$$\begin{aligned}
th^{\circ} &= \langle env^{\circ}, \mathbf{tagfunction}^{O,\tau}[env^{\circ}_1, pm_{\mathsf{Any}}(Q)]\ v, stack^{\circ}, store^{\circ}\rangle \\
&\to^* \langle env^{\circ}_2, pe^{\circ}_2, stack^{\circ}, store^{\circ}\rangle \\
&\qquad \text{where } env^{\circ}_2 \overset{\text{def}}{=} env^{\circ}_1[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto v_1, \dots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto v_k], \\
&\qquad\qquad pe^{\circ}_2 \overset{\text{def}}{=} \mathbf{if}\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \dots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k)) \\
&\qquad\qquad\qquad \mathbf{then}\ (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \dots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P)) \\
&\qquad\qquad\qquad \mathbf{else\ raise\ Bad\_Call}
\end{aligned}$$

$$\to^* th^{\circ}_1 \overset{\text{def}}{=} \langle env^{\circ}_2, \mathbf{raise\ Bad\_Call}, stack^{\circ}, store^{\circ}_1\rangle$$

where $store^{\circ}_1 \supseteq store^{\circ}$ by Proposition 8.5 applied k times.

If the oracle O is not under replication, then (FailedCall1) applies, so either $O[\widetilde{a}] \notin \mathbb{I}$ and in this case by Property T2(a), $store^{\circ}[l_{\mathsf{tok}}(O[\widetilde{a}])] = \mathbf{Invalid}$, 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 $\mathbb{CT}_1$ be an extension of the trace $\mathbb{CT}$ until $\mathbb{C}[\mathsf{th} \mapsto th^{\circ}_1]$ and $\mathcal{C}^{\mathsf{cs}}_1 \overset{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, steps-1, \mathbb{CS}_1$. We have $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow \mathcal{C}^{\mathsf{cs}}_1$. Let us prove that $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}_1$. The current expression is an exceptional value, so *replacecalls* allows any environment in the current thread, and $store^{\circ}_1 \supseteq store^{\circ}$, so Property T2(a) is preserved for the current thread. The OCaml side uses more reductions than the simulator side, so Property I14 is preserved. There is one more oracle call, and α and $\mathbb{I}$ are unchanged, so Properties I15 and I16 are preserved. The other properties are inherited from $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$.

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 $\mathbb{CS}$ does not reduce. Case 2.6 covers the reduction rule (Simulator add thread). So $\mathbb{CS}$ reduces using rule (Simulator toplevel). So let us denote $\mathbb{CS}_1$ the configuration such that $\mathbb{CS} = \mathbb{C}^s, \mathbb{RI}, \mathbb{I} \to \mathbb{CS}_1 = \mathbb{C}^s_1, \mathbb{RI}, \mathbb{I}$. Since the case of failed oracle calls is already handled in Case 2.7, $\mathbb{C}^s \to \mathbb{C}^s_1$ is obtained by rules of the OCaml semantics, not by (FailedCall1) or (FailedCall2).

If $pe^{\mathsf{s}} = \mathbf{schedule}(tj')$, then by Property T2(a), $pe^{\circ} = \mathbf{schedule}(tj')$, so $\mathbb{C}^s$ and $\mathbb{C}$ reduce in the same way by (Toplevel schedule1) or (Toplevel schedule2) for $\mathbb{C}^s$ and by (New toplevel schedule1) or (New toplevel schedule2) for $\mathbb{C}$. Let $\mathbb{C}_1$ be the configuration such that $\mathbb{C} \to \mathbb{C}_1$ and $\mathbb{CT}_1$ be the extension of the trace $\mathbb{CT}$ until $\mathbb{C}_1$. Let $\mathcal{C}^{\mathsf{cs}}_1 \overset{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, steps-1, \mathbb{CS}_1$. We have $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow \mathcal{C}^{\mathsf{cs}}_1$ and $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}_1$.

In all other cases, $\mathbb{C}^s$ reduces by (Toplevel). By Property T2(a), 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 $\mathbb{C}$ 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 $\mathbb{C}_1$ be the configuration such that $\mathbb{C} \to \mathbb{C}_1$ and $\mathbb{CT}_1$ be the extension of the trace $\mathbb{CT}$ until $\mathbb{C}_1$. Let $\mathcal{C}^{\mathsf{cs}}_1 \overset{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, steps-1, \mathbb{CS}_1$. We have $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow \mathcal{C}^{\mathsf{cs}}_1$ and we briefly show that $\mathcal{C}^{\mathsf{cs}}_1 \equiv \mathbb{CT}_1$.

If the reduction touches a local store location l, then by Properties T2(a) and T2(b), 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 I6 holds for $\mathcal{C}_1^{\mathsf{cs}}$ and $\mathbb{CT}_1$. (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 I7, the location l is not in Loc_{priv}, and in OCaml the same operation is carried out on l. So in all cases, Properties I7, I8, I9, and I10 hold for $\mathcal{C}_1^{\mathsf{cs}}$ and $\mathbb{CT}_1$. The oracle sets may only decrease, in case a subexpression is removed by reduction, so Property I13 is preserved. The reduction is performed in one step on both sides, so Property I14 is preserved. The other properties are inherited from $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$, so $\mathcal{C}_1^{\mathsf{cs}} \equiv \mathbb{CT}_1$. □

G Proof of Proposition 8.36

Definition G.1 *The relation $\mathcal{CT}_1^{\mathsf{cs}}, \dots, \mathcal{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathbb{CTS}_1, \dots, \mathbb{CTS}_n$ is verified when the following properties hold:*

1. *All traces $\mathcal{CT}_1^{\mathsf{cs}}, \dots, \mathcal{CT}_n^{\mathsf{cs}}$ start at $\mathcal{C}_0(Q_0, program_0)$, and none of these traces is a prefix of another of these traces.*
2. *The trace sets $\mathbb{CTS}_1, \dots, \mathbb{CTS}_n$ are pairwise disjoint, all traces in these sets start at $\mathbb{C}_0(Q_0, program_0)$, and none of these traces is a prefix of another of these traces.*
3. *$\forall i \leq n, \Pr[\mathcal{CT}_i^{\mathsf{cs}}] = \Pr[\mathbb{CTS}_i]$.*
4. *$\sum_{i \leq n} \Pr[\mathcal{CT}_i^{\mathsf{cs}}] = 1$.*
5. *For each trace $\mathcal{CT}_i^{\mathsf{cs}}$, $i \leq n$,*
   - (a) *either $\mathcal{CT}_i^{\mathsf{cs}}$ is complete, every trace $\mathbb{CT} \in \mathbb{CTS}_i$ is complete, and the event list $\mathcal{E}$ of the last configuration of $\mathcal{CT}_i^{\mathsf{cs}}$ and the event list events of the last configuration of $\mathbb{CT}$ verify events $= \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$,*
   - (b) *or for every trace $\mathbb{CT} \in \mathbb{CTS}_i$, the last configuration $\mathcal{C}^{\mathsf{cs}}$ of $\mathcal{CT}_i^{\mathsf{cs}}$ verifies $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$.*

The next lemma applies to any traces, so in particular to OCaml traces and CryptoVerif traces.

Lemma G.2 *Let $\mathbb{CT}_1, \dots, \mathbb{CT}_n$ be traces such that none of these traces is a prefix of another of these traces. If $\mathbb{CT}_1'', \dots, \mathbb{CT}_{n'}''$ are extensions of $\mathbb{CT}_n$ such that none of these traces is a prefix of another, then none of the traces $\mathbb{CT}_1, \dots, \mathbb{CT}_{n-1}, \mathbb{CT}_1'', \dots, \mathbb{CT}_{n'}''$ is a prefix of another of these traces.*

In particular, this is true when, for all $i \leq n'$, $\mathbb{CT}_i''$ is the concatenation of $\mathbb{CT}_n$ and $\mathbb{CT}_i'$ where $\mathbb{CT}_1', \ldots, \mathbb{CT}_{n'}'$ are traces that start at the last configuration of $\mathbb{CT}_n$ 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 $\mathbb{CT}_1, \ldots, \mathbb{CT}_{n-1}, \mathbb{CT}_1'', \ldots, \mathbb{CT}_{n'}''$. If they are both among $\mathbb{CT}_1, \ldots, \mathbb{CT}_{n-1}$, they are not prefix of one another by hypothesis. If they are both among $\mathbb{CT}_1'', \ldots, \mathbb{CT}_{n'}''$, they are also not prefix of one another by hypothesis. Now consider $\mathbb{CT}_i$ with $i \leq n-1$ and $\mathbb{CT}_j''$ with $j \leq n'$. If $\mathbb{CT}_i$ was a prefix of $\mathbb{CT}_j''$, then either its length is less or equal to the length of $\mathbb{CT}_n$, so $\mathbb{CT}_i$ would be a prefix of $\mathbb{CT}_n$, which is impossible by hypothesis, or its length is greater than the length of $\mathbb{CT}_n$, so $\mathbb{CT}_i$ would be an extension of $\mathbb{CT}_n$, that is, $\mathbb{CT}_n$ would be a prefix of $\mathbb{CT}_i$, which is also impossible by hypothesis. If $\mathbb{CT}_j''$ was a prefix of $\mathbb{CT}_i$, then a fortiori $\mathbb{CT}_n$ would be a prefix of $\mathbb{CT}_i$, which is impossible by hypothesis. Hence, none of the traces $\mathbb{CT}_1, \ldots, \mathbb{CT}_{n-1}, \mathbb{CT}_1'', \ldots, \mathbb{CT}_{n'}''$ is a prefix of another of these traces.

To show the second point, if $\mathbb{CT}_i''$ was a prefix of $\mathbb{CT}_j''$, then $\mathbb{CT}_i'$ would be a prefix of $\mathbb{CT}_j'$, which is a contradiction. So we can apply the first point in this case. $\square$

Lemma G.3 *Suppose that $\mathcal{CT}_1^{\mathsf{cs}}, \ldots, \mathcal{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathbb{CTS}_1, \ldots, \mathbb{CTS}_n$. Either all traces $\mathcal{CT}_1^{\mathsf{cs}}, \ldots, \mathcal{CT}_n^{\mathsf{cs}}$ are complete, or there exist $\mathcal{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathcal{CT}_{n'}^{\mathsf{cs}\prime}$ and $\mathbb{CTS}_1'$, $\ldots, \mathbb{CTS}_{n'}'$ such that there are strictly more reduction steps in traces $\mathcal{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathcal{CT}_{n'}^{\mathsf{cs}\prime}$ than in traces $\mathcal{CT}_1^{\mathsf{cs}}, \ldots, \mathcal{CT}_n^{\mathsf{cs}}$ and $\mathcal{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathcal{CT}_{n'}^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathbb{CTS}_1', \ldots, \mathbb{CTS}_{n'}'$.*

Proof Suppose that $\mathcal{CT}_1^{\mathsf{cs}}, \ldots, \mathcal{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathbb{CTS}_1, \ldots, \mathbb{CTS}_n$ and there is a trace $\mathcal{CT}_i^{\mathsf{cs}}$ that is not complete. We can renumber the traces so that the last trace $\mathcal{CT}_n^{\mathsf{cs}}$ is not complete.

By Property 5(b), the last configuration $\mathcal{C}^{\mathsf{cs}}$ of the trace $\mathcal{CT}_n^{\mathsf{cs}}$ and all traces $\mathbb{CT} \in \mathbb{CTS}_n$ verify $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}$. By Property 2, $\mathbb{CT}$ is a trace beginning at $\mathbb{C}_0(Q_0, \mathit{program}_0)$. Let us denote $\mathbb{CTS}_n = \{\mathbb{CT}_1, \ldots, \mathbb{CT}_m\}$. We can then apply Lemma 8.35 to $\mathcal{C}^{\mathsf{cs}}$.

- Either there exist n' configurations $\mathcal{C}_1^{\mathsf{cs}}, \ldots, \mathcal{C}_{n'}^{\mathsf{cs}}$, n' traces $\mathcal{C}^{\mathsf{cs}} \rightsquigarrow_{p_1}^{+} \mathcal{C}_1^{\mathsf{cs}}$, $\ldots, \mathcal{C}^{\mathsf{cs}} \rightsquigarrow_{p_{n'}}^{+} \mathcal{C}_{n'}^{\mathsf{cs}}$ such that none of these traces is a prefix of another, $\sum_{i \leq n} p_i = 1$, and for each trace $\mathbb{CT}_j, j \leq m$, there exist n' pairwise disjoint trace sets $\mathbb{CTS}_{j,1}, \ldots, \mathbb{CTS}_{j,n'}$ such that all traces in these sets are extensions of $\mathbb{CT}_j$, none of these traces is a prefix of another, and for each trace $\mathbb{CT} \in \mathbb{CTS}_{j,i}, \mathcal{C}_i^{\mathsf{cs}\prime} \equiv \mathbb{CT}$ and $\Pr[\mathbb{CTS}_{j,i}] = p_i \cdot \Pr[\mathbb{CT}_j]$. Let us denote $\mathbb{CTS}_i' \stackrel{\text{def}}{=} \bigcup_{j \leq m} \mathbb{CTS}_{j,i}$. Let us also denote $\mathcal{C}_i^{\mathsf{cs}\prime}$ the extension of the trace $\mathcal{C}^{\mathsf{cs}}$ until $\mathcal{C}_i^{\mathsf{cs}}$, for $i \leq n'$. There is at least one new reduction step, so there are more reduction steps in $\mathcal{CT}_1^{\mathsf{cs}}, \ldots, \mathcal{CT}_{n-1}^{\mathsf{cs}}, \mathcal{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathcal{CT}_{n'}^{\mathsf{cs}\prime}$ than in $\mathcal{CT}_1^{\mathsf{cs}}, \ldots, \mathcal{CT}_n^{\mathsf{cs}}$. Let us prove that $\mathcal{CT}_1^{\mathsf{cs}}, \ldots, \mathcal{CT}_{n-1}^{\mathsf{cs}}, \mathcal{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathcal{CT}_{n'}^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathbb{CTS}_1, \ldots, \mathbb{CTS}_{n-1}, \mathbb{CTS}_1', \ldots, \mathbb{CTS}_{n'}'$. All traces $\mathcal{CT}_1^{\mathsf{cs}}, \ldots, \mathcal{CT}_{n-1}^{\mathsf{cs}}, \mathcal{CT}_1^{\mathsf{cs}\prime}$, $\ldots, \mathcal{CT}_{n'}^{\mathsf{cs}\prime}$ start at $\mathcal{C}_0(Q_0, \mathit{program}_0)$ 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 $\mathbb{CT}_j$ for $j \leq m$, Property 2 is verified. By Property 3 on the initial traces, $\forall i \leq n, \Pr[\mathcal{C}^{\mathsf{cs}}_i] = \Pr[\mathbb{CTS}_i]$. We have that $\Pr[\mathbb{CTS}'_i] = \sum_{j \leq m} \Pr[\mathbb{CTS}_{j,i}]$ because all the sets $\mathbb{CTS}_{j,i}$ are disjoint. So,

$$\Pr[\mathbb{CTS}'_i] = \sum_{j \leq m} p_i \cdot \Pr[\mathbb{CT}_j] = p_i \cdot \Pr[\mathbb{CTS}_n]\,, \text{ so}$$
$$\Pr[\mathcal{CT}^{\mathsf{cs}\prime}_i] = p_i \cdot \Pr[\mathcal{CT}^{\mathsf{cs}}_n] = \Pr[\mathbb{CTS}'_i]\,,$$

Property 3 is verified. By Property 4 on the initial traces, we have $\sum_{i \leq n} \Pr[\mathcal{CT}^{\mathsf{cs}}_i] = 1$. We have that

$$\sum_{i \leq n'} \Pr[\mathcal{CT}^{\mathsf{cs}\prime}_i] = \sum_{i \leq n'} p_i \cdot \Pr[\mathcal{CT}^{\mathsf{cs}}_n] = \Pr[\mathcal{CT}^{\mathsf{cs}}_n]\,, \text{ so}$$
$$\sum_{i \leq n-1} \Pr[\mathcal{CT}^{\mathsf{cs}}_i] + \sum_{i \leq n'} \Pr[\mathcal{CT}^{\mathsf{cs}\prime}_i] = \sum_{i \leq n} \Pr[\mathcal{CT}^{\mathsf{cs}}_i] = 1\,.$$

So Property 4 is verified. We inherit Property 5 for the $n-1$ first elements. For all $i \leq n'$, for all traces $\mathbb{CT} \in \mathbb{CTS}'_i$, we have $\mathcal{C}^{\mathsf{cs}\prime}_i \equiv \mathbb{CT}$, and $\mathcal{C}^{\mathsf{cs}\prime}_i$ is the last configuration of $\mathcal{CT}^{\mathsf{cs}\prime}_i$. So Property 5(b) is verified for all the new elements. Hence Property 5 is verified. Therefore, $\mathcal{CT}^{\mathsf{cs}}_1, \dots, \mathcal{CT}^{\mathsf{cs}}_{n-1}$, $\mathcal{CT}^{\mathsf{cs}\prime}_1, \dots, \mathcal{CT}^{\mathsf{cs}\prime}_{n'} \equiv_{\mathsf{t}} \mathbb{CTS}_1, \dots, \mathbb{CTS}_{n-1}, \mathbb{CTS}'_1, \dots, \mathbb{CTS}'_{n'}$.

- Otherwise, each trace $\mathbb{CT} \in \mathbb{CTS}_n$ is complete, $\mathcal{C}^{\mathsf{cs}} \to^* \mathcal{C}^{\mathsf{cs}}_1$, $\mathcal{C}^{\mathsf{cs}}_1$ cannot reduce, and the event list $\mathcal{E}$ of $\mathcal{C}^{\mathsf{cs}}_1$ and the event list *events* of the last configuration of $\mathbb{CT}$ satisfy $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$. Let $\mathcal{CT}^{\mathsf{cs}\prime}_n$ be the extension of the trace $\mathcal{CT}^{\mathsf{cs}}_n$ until $\mathcal{C}^{\mathsf{cs}}_1$. The trace $\mathcal{CT}^{\mathsf{cs}\prime}_n$ contains more steps than $\mathcal{CT}^{\mathsf{cs}}_n$, so there are more reduction steps in $\mathcal{CT}^{\mathsf{cs}}_1, \dots, \mathcal{CT}^{\mathsf{cs}}_{n-1}, \mathcal{CT}^{\mathsf{cs}\prime}_n$ than in $\mathcal{CT}^{\mathsf{cs}}_1, \dots, \mathcal{CT}^{\mathsf{cs}}_n$. Let us prove that $\mathcal{CT}^{\mathsf{cs}}_1, \dots, \mathcal{CT}^{\mathsf{cs}}_{n-1}, \mathcal{CT}^{\mathsf{cs}\prime}_n \equiv_{\mathsf{t}} \mathbb{CTS}_1, \dots,$ $\mathbb{CTS}_n$. By Lemma G.2, no traces in $\mathcal{CT}^{\mathsf{cs}}_1, \dots, \mathcal{CT}^{\mathsf{cs}}_{n-1}, \mathcal{CT}^{\mathsf{cs}\prime}_n$ are prefixes of one another, so Property 1 is verified. Property 2 is inherited. We have that $\Pr[\mathcal{CT}^{\mathsf{cs}\prime}_n] = \Pr[\mathcal{CT}^{\mathsf{cs}}_n]$, so Properties 3 and 4 are verified. The trace $\mathcal{CT}^{\mathsf{cs}\prime}_n$ is complete, every trace $\mathbb{CT} \in \mathbb{CTS}_n$ is complete, and the event list *events* of the last configuration of traces in $\mathbb{CTS}_n$ and the event list $\mathcal{E}$ of $\mathcal{C}^{\mathsf{cs}}_1$ 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, $\mathcal{CT}^{\mathsf{cs}}_1$, $\dots, \mathcal{CT}^{\mathsf{cs}}_{n-1}, \mathcal{CT}^{\mathsf{cs}\prime}_n \equiv_{\mathsf{t}} \mathbb{CTS}_1, \dots, \mathbb{CTS}_n$. □

Proof (of Proposition 8.36) By Lemma 8.34, we have a trace $\mathcal{CT}_0 = \mathcal{C}_0(Q_0, \mathit{program}_0) \rightsquigarrow^* \mathcal{C}^{\mathsf{cs}}$ where $\mathcal{C}^{\mathsf{cs}} \equiv \mathbb{CT}_0$ and $\mathbb{CT}_0 = \mathbb{C}_0(Q_0, \mathit{program}_0)$. We prove easily that $\mathcal{CT}_0 \equiv_{\mathsf{t}} \{\mathbb{CT}_0\}$. The number of steps in complete traces from configuration $\mathcal{C}_0(Q_0, \mathit{program}_0)$ is finite. Let us consider traces such that $\mathcal{CT}^{\mathsf{cs}}_1, \dots, \mathcal{CT}^{\mathsf{cs}}_n \equiv_{\mathsf{t}} \mathbb{CTS}_1, \dots, \mathbb{CTS}_n$ with the maximum number of reduction steps. By Lemma G.3, the traces $\mathcal{CT}^{\mathsf{cs}}_1, \dots, \mathcal{CT}^{\mathsf{cs}}_n$ are complete. (Otherwise, we could extend them.) Since the sum of their probabilities is 1, these are all complete traces starting at $\mathcal{C}_0(Q_0, \mathit{program}_0)$. The proposition follows. □

H Index of Notations

$\mathcal{Q} \leftrightarrow \mathbb{RI}, \mathbb{I}$	Correspondence between processes in $\mathcal{Q}$ and oracles in $\mathbb{RI}$, $\mathbb{I}$	71
$\mathbb{RI} - \mathsf{role}[\widetilde{a}]$	The set $\mathbb{RI}$ without role $\mathsf{role}[\widetilde{a}]$	59
$\mathbb{RI} - \mathbb{RI}'$	The set $\mathbb{RI}$ without all roles in $\mathbb{RI}'$	59
$\mathbb{I} - O[\widetilde{a}]$	The set $\mathbb{I}$ without oracle $O[\widetilde{a}]$	59
$\rightarrow_p$	Semantic reduction	16
$\rightsquigarrow$	Reduction of intermediate CryptoVerif semantics	66
$\widetilde{\cdot}$	list of $\cdot$, where $\cdot$ is either i or a	10
$env \oplus env'$	Add bindings of env to env'	26
$\approx_v$	Value equivalence between configurations	39
$\approx_{vth}$	Value equivalence between threads	39
$\Downarrow$	CryptoVerif term evaluation	13
$\equiv$	Relation between CryptoVerif and OCaml traces	76
γ	Boolean indicating whether the oracle or role can be called only once or not. Either Once or Any.	37
μ	OCaml module	34
μ_{prim}	OCaml module for cryptographic primitives	21
μ_{role}	Translation of role	40
τ	Fresh tag for every closure	36
τ_{O}	Function that maps oracles to a closure tag τ	72
a	CryptoVerif constant	13
$add(\mathbb{I}, \mathbb{RI})$	Add first oracles present in $\mathbb{RI}$ in $\mathbb{I}$	60
addthread	Add thread to thread list	25
$almostunif(A, b)$	Probability of $b \in A$ according to an almost-uniform distribution	8
Any	Can be called any number of times	37
$\mathbb{C}$	OCaml configuration (after Section 7, the instrumented one)	31
$\mathcal{C}$	CryptoVerif semantic configuration	13
$\mathbb{C}_0(program)$	Initial OCaml configuration for $program$	34
$\mathbb{C}[\mathsf{th} \mapsto th, \mathsf{globalstore} \mapsto globalstore, \mathsf{MI} \mapsto \mathbb{MI}, \mathsf{events} \mapsto events]$	Modified OCaml configuration	54
$\mathcal{C}^{\mathsf{cs}}$	Intermediate semantic configuration	66
$\mathbb{C}_{events}(\mathbb{C})$	Event list of configuration $\mathbb{C}$	53
$\mathbb{C}_{globalstore}(\mathbb{C})$	Global store of configuration $\mathbb{C}$	53
$\mathcal{C}_{\mathsf{i}}(Q_0)$	Initial CryptoVerif configuration for Q_0	17
C_{m}	OCaml minimal evaluation context	27
C_{me}	OCaml minimal expression evaluation context	27
C_{mp}	OCaml minimal program evaluation context	27
$\mathbb{C}_{pe}(\mathbb{C})$	Current program or expression of configuration $\mathbb{C}$	53
$\mathbb{C}_{th}(\mathbb{C})$	Current thread of configuration $\mathbb{C}$	53
$\mathbf{call}(O[\widetilde{a}]), \mathbf{call}(O[\_, \widetilde{a}])$	Values representing calls to oracles in the simulator	60
Callable	Oracle is callable	42
$\mathbb{CI}$	OCaml instrumented configuration	37

continue	CryptoVerif loop continues. An alias of true.	11
$\mathbb{CS}$	Simulator configuration	58
$\mathbb{CT}$	OCaml trace	36
$\mathcal{CT}$	CryptoVerif trace	18
$\mathbb{CTS}$	OCaml trace set	36
$\mathcal{CTS}$	CryptoVerif trace set	18
D	Distinguisher on event lists	19
E	CryptoVerif environment	13
e	OCaml expression	23
$\mathcal{E}$	CryptoVerif event list	13
env	OCaml environment	25
$env(E, \cdot)$	Minimal environment for describing $\cdot$ where $\cdot = P, M$ or Q	53
env_{prim}	Environment after evaluation of μ_{prim}	49
ev	Event	11
$events$	OCaml event list	36
f	CryptoVerif function symbol	10
$Files$	Association set between files and variables to save	20
$\mathbf{function}[env, pm]$	OCaml closure	25
fv	Free variables	52
$\mathbb{G}$	Translation function for oracle bodies	42
$\mathbb{G}_{\mathsf{deser}}(T)$	Deserialization of type T	21
$\mathbb{G}_{\mathsf{ev}}$	Translation function for events	49
$\mathbb{G}_{\mathsf{f}}$	Translation function for CryptoVerif functions	21
$\mathbb{G}_{\mathsf{file}}$	Write a variable into a file	42
$\mathbb{G}_{\mathsf{get}\mathbb{MI}}(Q)$	Set of modules in Q	41
$\mathbb{G}_{\mathsf{M}}$	Translation function for terms	41
$\mathbb{G}_{\mathsf{O}}$	Translation function for oracles	42
$\mathbb{G}_{\mathsf{pred}}(T)(v)$	Return whether value v corresponds to a value of type T	21
$\mathbb{G}_{\mathsf{random}}(T)$	OCaml function for generating random values of type T	21
$\mathbb{G}_{\mathsf{read}}$	Read a variable from a file	44
$\mathbb{G}_{\mathsf{ser}}(T)$	Serialization of type T	21
$\mathbb{G}_{\mathsf{T}}(T)$	OCaml type corresponding to CryptoVerif type T	21
$\mathbb{G}_{\mathsf{tbl}}$	OCaml value of a table row	52
$\mathbb{G}_{\mathsf{tblel}}$	Serialized OCaml value of a table row	52
$\mathbb{G}_{\mathsf{val}T}(a)$	OCaml value corresponding to the CryptoVerif value $a : T$	49
$\mathbb{G}_{\mathsf{var}}$	Translation function for variables	40
$gettokens$	Function that returns part of the store corresponding to tokens	73
$globalstore$	OCaml global store	33
$globalstore(E, \mathcal{T})$	Global store corresponding to E and $\mathcal{T}$	53
$globalstore_0$	Initial global store	34
i	CrypvoVerif replication index	10
$\mathbb{I}$	Set containing available oracles with their replication indices	58
$initval_l$	Default value for location l in global store	34
$interface(\mu)$	Interface of module μ	34

$\mathit{interface}_{\mathsf{prim}}$	Interface of μ_{prim}	48
Invalid	Oracle or role cannot be called again	42
L	OCaml reduction label	26
l	OCaml store location	23
l_{tok}	Function that maps an oracle not under replication to its token	72
$\mathbf{letrec}[\mathit{env}, \{x_1 \mapsto \mathbf{function}\ pm_1, \dots, x_n \mapsto \mathbf{function}\ pm_n\}\ \mathbf{in}\ x_i]$	OCaml closure for recursive functions	25
$\mathit{Loc}_{\mathsf{g}}$	Locations allowed only in global store	33
Loc_{ℓ}	Locations allowed only in local store	33
$\mathit{Loc}_{\mathsf{priv}}$	Global locations reserved for private use	46
M	CryptoVerif term	10
$\mathbb{M}_{\mathsf{g}}$	Set of generated modules	37
$\mathbb{MI}$	Module set	37
N	Bound on replication	10
N_O	Bound on replication under oracle O	56
$N_{\mathsf{rand+calls}}$	Maximum number of oracle calls and random generations	56
N_{role}	Bound on replication under role role	56
N_{steps}	Maximum number of reduction steps of $\mathit{program}_0$	56
$\mathit{noinstr}_{\mathbb{CI}}$	Remove instrumentation	40
O	CryptoVerif oracle	10
$O_1, \dots, O_n$	Oracle names present in Q_0	58
$\mathrm{o}_1, \dots, \mathrm{o}_n$	CryptoVerif constants representing oracles in Q_0	58
o_{R}	CryptoVerif constant for random choice	58
o_{S}	CryptoVerif constant for end of program	58
$\mathcal{O}$	Function that returns the oracles present in its argument	71
$\mathcal{O}^{\infty}$	Same as $\mathcal{O}$, but ignoring replication bounds	68
$\mathcal{O}_{\mathbf{call}}$	Set of oracles not under replication appearing in argument	68
$\mathcal{O}_{\mathbf{call}\text{-}\mathsf{repl}}$	Set of oracles under replication appearing in argument	68
$\mathsf{O}_{\mathsf{start}}$	Oracle called at the beginning	17
Once	Can be called only once	37
$\mathit{oracledeflist}(Q)$	List of oracles in Q, no instantiation of replication indices	41
$\mathit{oraclelist}(Q)$	List of names and replication indices of oracles in Q	59
$\mathit{oracledefset}(Q)$	Set of oracles in Q, indices in replications instantiated	16
P	CryptoVerif oracle body	10
P_{loop}	Oracle body of $Q_{\mathsf{start}}(Q_0, \mathit{program}_0)$	64
$P_{\mathsf{return\text{-}loop}}(\alpha)$	The process in the loop between two steps, in the simulator	64
pat	OCaml pattern	23
pe	OCaml program or expression	27
pm	OCaml pattern matching	24
$\mathit{pm}_{\gamma}(Q)$	Translation of Q	44
$\mathit{pm}'_{\mathsf{role}[\widetilde{a}]}$	Pattern matching corresponding to role role	62
$\Pr[\cdot]$	Probability of a trace or trace set	18

$\Pr[\mathcal{C} :^{(\mathsf{CV})} D]$		
	Probability that a CryptoVerif trace beginning at $\mathcal{C}$ verifies D	19
$\Pr[\mathbb{C} :^{(\mathsf{ML})} D]$		
	Probability that an OCaml trace beginning at $\mathbb{C}$ verifies D	66
$\Pr[\mathcal{C}^{\mathsf{cs}} :^{(\leadsto)} D]$		
	Probability that a trace beginning at $\mathcal{C}^{\mathsf{cs}}$ using $\leadsto$ verifies D	67
prim	OCaml primitive	23
$\mathit{program}$	OCaml program	24
$\mathit{program}_{\mathsf{prim}}$	Program of module μ_{prim}	48
$\mathit{program}(\mu)$	Program of module μ	34
$\mathit{program}_0$	OCaml program that interacts with the translation of Q_0	48
$\mathit{program}'(\mathsf{role}[\widetilde{a}])$		
	Simulator program corresponding to role role	62
$\mathcal{Q}$	CryptoVerif set of callable oracle definitions	13
$\mathcal{Q}(O[\widetilde{a}])$	Element of $\mathcal{Q}$ corresponding to $O[\widetilde{a}]$	71
Q	CryptoVerif oracle definition	10
$Q[\widetilde{a}]$	Process Q where elements of $\widetilde{a}$ are substituted for elements of $\widetilde{i}$	60
$Q\{a/i\}$	Process Q where a is substituted for i	16
$Q(\mathsf{role})$	Oracle definition corresponding to role role	20
Q_0	The CryptoVerif process we consider	48
$Q_{\mathsf{adv}}(Q_0, \mathit{program}_0)$		
	CryptoVerif adversary that simulates $\mathit{program}_0$	48
$Q_{\mathsf{c}}(Q_0, \mathit{program}_0)$		
	Stepping oracle of $Q_{\mathsf{adv}}(Q_0, \mathit{program}_0)$	55
Q_{loop}	$\mathrm{O}_{\mathsf{loop}}[i'](s : T_{\mathbb{CS}}) := P_{\mathsf{loop}}$	64
$Q_{\mathsf{start}}(Q_0, \mathit{program}_0)$		
	Starting oracle of $Q_{\mathsf{adv}}(Q_0, \mathit{program}_0)$	55
$\mathcal{S}$	CryptoVerif call stack	13
$\mathcal{R}_{\mathsf{init\text{-}function}}$	Set of roles whose modules are not evaluated yet	68
$\mathcal{R}_{\mathsf{init\text{-}closure}}$	Set of roles whose modules are evaluated but whose *init* function has not been called yet	68
$\mathcal{S}_{\mathsf{loop}}(\alpha)$	The contents of the call stack, in the simulator	64
$\mathbf{random}_\ell$	Random element in list	42
$\mathit{replacecalls}$	Replace calls to oracles with corresponding closures	73
$\mathit{replaceinitpm}$		
	Replace simulator role initializations with OCaml ones	72
$\mathit{repr}(\mathbb{CS})$	Function returning the bitstring representation of $\mathbb{CS}$	58
return	Return list of new roles	39
$\mathit{returnoracles}(O[\widetilde{a}])$		
	Oracles returned by the return statements of $O[\widetilde{a}]$	60
$\mathbb{RI}$	Set containing available roles with their replication indices	58
$\mathbb{RI}_0$	Initial role set	64
role	CryptoVerif role	19
$s_0(Q_0, \mathit{program}_0)$		
	Representation of initial simulator configuration	64

schedule	Schedule another thread	25
$ser(T, a)$	Serialization of CryptoVerif value a	50
$simreturn(\mathbb{CS})$	Return value of simulator	62
$\text{simulate}_{\mathsf{ML}}$	CryptoVerif function that simulates OCaml code	58
$\text{simulate}_{\mathsf{end}O}$	Function that integrates the fact that an oracle terminated with end into the simulator configuration	63
$\text{simulate}_{\mathsf{R}}$	Function that integrates the result of a random choice into the simulator configuration	64
$\text{simulate}_{\mathsf{ret}O}$	Function that integrates the return values of an oracle into the simulator configuration	63
$smallest(\mathbb{RI}, \mathsf{role})$	Smallest indices for role role in $\mathbb{RI}$	59
$stack$	OCaml call stack	27
$steps$	Number of remaining steps	66
stop	CryptoVerif loop stops. An alias of false.	11
$store$	OCaml store	31
T	CryptoVerif type	9
t	Closure tag containing the oracle corresponding to the closure	36
$\mathcal{T}$	CryptoVerif set of tables	13
$\mathcal{T}_0$	Initial table set	17
$T_{\mathbb{CS}}$	CryptoVerif type that is the image of $repr$	58
T_M	Type of term M	41
T_{o}	CryptoVerif type containing constants for oracle names, random and stop	58
$Tables$	Association set between files and tables	20
$\textbf{tagfunction}^{t}\ pm$	Tagged function	36
$\textbf{tagfunction}^{t,\tau}[env, pm]$	Tagged closure	36
Tbl	CryptoVerif table	10
tj	Thread number	33
th	OCaml thread	31
th_0^s	Thread that only evaluates μ_{prim}	49
$token$	Boolean reference indicating whether an oracle can be called	42
$\mathbb{TT}$	Trace for a single thread	49
v	OCaml value	25
$willbeavailable$	Set of oracles that can eventually become available	68
x	Variable	10