ANR TECAP
Deliverable D2.5: def. of encodings for prover communication
From CryptoVerif to EasyCrypt

This document presents an encoding from CryptoVerif to EasyCrypt. This encoding deals with the language in which security assumptions on cryptographic primitives are expressed in CryptoVerif: CryptoVerif typically has more difficulties proving primitives than protocols, so our goal is to delegate to EasyCrypt the proof of security assumptions on cryptographic primitives that CryptoVerif is unable to perform.

1 The CryptoVerif language for specifying security assumptions

CryptoVerif specifies assumptions on primitives as indistinguishability axioms $L \approx_p R$, meaning that an adversary has probability at most p of distinguishing the left-hand side L from the right-hand side R. (p is a typically is function of the runtime of the adversary, the number of calls to oracles, and possibly other parameters.) CryptoVerif then uses these axioms by replacing L with R inside a bigger game [1].

The games L and R are written in the language specified in Figure 1. In this language, terms represent computations on bitstrings. The replication index i is an integer which serves in distinguishing different copies of a replicated structure **foreach** $i \leq N$ **do** . (Replication indices are typically used as array indices; in CryptoVerif, arrays replace lists often used in cryptographic proofs, for instance to store all arguments and results of calls to some oracles.) The variable access $x[M_1, \dots, M_m]$ returns the content of the cell of indices $M_1, \dots, M_m$ of the m-dimensional array variable x. We use x, y, z, u as variable names. The function application $f(M_1, \dots, M_m)$ returns the result of applying function f to $M_1, \dots, M_m$. Each function symbol comes with a type declaration $f : T_1 \times \dots \times T_m \to T$ and represents a deterministic efficiently computable function from $T_1 \times \dots \times T_m$ to T, where the types $T_1, \dots, T_m, T$ are sets of values, typically bitstrings.

The oracle bodies represent computations made to obtain the result of an oracle. The return instruction $\mathbf{return}(M)$ simply returns the result of M. The random choice $x[i_1, \dots, i_m] \leftarrow_\$ T;; B$ chooses a new random number uniformly in T, stores it in $x[i_1, \dots, i_m]$, and executes B. The process $x[i_1, \dots, i_m] : T \leftarrow M; B$ stores the result of M in $x[i_1, \dots, i_m]$ and executes B. The type T is the type of x and of M; it can be omitted since it can be inferred from M. The conditional **if** M **then** B **else** B' executes B if $M_1, \dots, M_l$ are defined and M evaluates to *true*. Otherwise, it executes B'.

Next, we explain the **find** instruction: **find** $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}$ **suchthat defined**$(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ **then** $B_j)$ **else** B, where $\widetilde{i}$ denotes a tuple of indices $i_1, \dots, i_{m'}$. The order and array indices on tuples are taken component-wise, so for instance, $u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}$ can be further abbreviated $\widetilde{u_j}[\widetilde{i}] = \widetilde{i_j} \leq \widetilde{n_j}$. A simple example is the following: **find** $u = i \leq n$ **suchthat defined**$(x[i]) \wedge x[i] = a$ **then** B' **else** B tries to find an index i such that $x[i]$ is defined and $x[i] = a$, and when such a i is found, it stores it in u and executes B'; otherwise, it executes B. In other words, this **find** construct looks for the value a in the array x, and when a is found, it stores in u an index such that $x[u] = a$. Therefore, the **find** construct allows us to access arrays. More generally, **find** $u_1[\widetilde{i}] = i_1 \leq n_1, \dots, u_m[\widetilde{i}] = i_m \leq n_m$ **suchthat defined**$(M_1, \dots, M_l) \wedge M$ **then** B' **else** B tries to find values of $i_1, \dots, i_m$ for which $M_1, \dots, M_l$

$M ::=$ terms

- i — replication index
- $x[M_1, \ldots, M_m]$ — variable access
- $f(M_1, \ldots, M_m)$ — function application

$B ::=$ oracle bodies

- $\mathbf{return}(M)$ — return
- $x[i_1, \ldots, i_m] \leftarrow_{\$} T; B$ — random choice
- $x[i_1, \ldots, i_m] : T \leftarrow M; B$ — assignment
- $\mathbf{if}\ M\ \mathbf{then}\ B\ \mathbf{else}\ B'$ — conditional
- $\mathbf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq N_{j1}, \ldots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq N_{jm_j}$
  $\mathbf{suchthat}\ \mathbf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathbf{then}\ B_j)$
  $\mathbf{else}\ B$ — array lookup

$S ::=$ oracle structures

- $\mathbf{foreach}\ i \leq N\ \mathbf{do}\ y_1[\widetilde{i}] \leftarrow_{\$} T_1; \ldots y_l[\widetilde{i}] \leftarrow_{\$} T_l; (S_1 \mid \ldots \mid S_m)$ — replication, random choices
- $\mathcal{O}(x_1[\widetilde{i}] : T_1, \ldots, x_l[\widetilde{i}] : T_l) := B$ — oracle

$S_\top, L, R ::=$ oracle structures (toplevel)

- $y_1 \leftarrow_{\$} T_1; \ldots y_l \leftarrow_{\$} T_l; (S_1 \mid \ldots \mid S_m)$ — optional random choices

Figure 1: Grammar of CryptoVerif (language for specifying security assumptions)

are defined and M is true. In case of success, it stores them $u_1[\widetilde{i}]$, $\ldots$, $u_m[\widetilde{i}]$ and executes B'. In case of failure, it executes B. This is further generalized to m branches: $\mathbf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}\ \mathbf{suchthat}\ \mathbf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathbf{then}\ B_j)\ \mathbf{else}\ B$ tries to find a branch j in $[1, m]$ such that there are values of $i_{j1}, \ldots, i_{jm_j}$ for which $M_{j1}, \ldots, M_{jl_j}$ are defined and M_j is true. In case of success, it stores them in $u_{j1}[\widetilde{i}]$, $\ldots$, $u_{jm_j}[\widetilde{i}]$ and executes B_j. In case of failure for all branches, it executes B. More formally, it evaluates the conditions $\mathbf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ for each j and each value of $i_{j1}, \ldots, i_{jm_j}$ in $[1, n_{j1}] \times \ldots \times [1, n_{jm_j}]$. If none of these conditions is *true*, it executes B. Otherwise, it chooses randomly with uniform[1] probability one j and one value of $i_{j1}, \ldots, i_{jm_j}$ such that the corresponding condition is *true*, stores it in $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ and executes B_j.

Finally, the oracle structures define oracles: the oracle structure $\mathcal{O}(x_1[\widetilde{i}] : T_1, \ldots, x_l[\widetilde{i}] : T_l) := B$ defines an oracle O with arguments $x_1, \ldots, x_l$ of types $T_1, \ldots, T_l$ respectively. The parallel composition of oracle structures $S_1 \mid \ldots \mid S_m$ defines simultaneously the oracles defined in the structures S_1, $\ldots$, S_m. The replication $\mathbf{foreach}\ i \leq N\ \mathbf{do}\ S$ defines N copies of the oracles in S, with index $i \in [1, N]$, and furthermore random values may be chosen after each replication by $y_1[\widetilde{i}] \leftarrow_{\$} T_1; \ldots y_l[\widetilde{i}] \leftarrow_{\$} T_l;$. ($l = 0$ is allowed when there is no random choice.) The replication may be omitted at toplevel.

These games are required to satisfy some syntactic constraints. Variables can be defined only at the current replication indices. That is, at a point in the game under replications $\mathbf{foreach}\ i_1 \leq N_1\ \mathbf{do}\ \ldots \mathbf{foreach}\ i_m \leq N_m\ \mathbf{do}$, a variable definition is possible only with indices $i_1, \ldots, i_m$. This constraint guarantees that each execution of a variable definition uses different indices. The following constructs of Figure 1 define variables: random choice ($x[i_1, \ldots, i_m]$), assignment ($x[i_1, \ldots, i_m]$), array

[1] A probabilistic bounded-time Turing machine can choose a random number uniformly in a set of cardinal m only when m is a power of 2. When m is not a power of 2, there exist approximate algorithms: for example, in order to obtain a random integer in $[0, m-1]$, we can choose a random integer r uniformly among $[0, 2^k - 1]$ for a certain k large enough and return $r \mod m$. The distribution can be made as close as we wish to the uniform distribution by choosing k large enough.

lookup ($u_{j1}[\widetilde{i}]$, ..., $u_{jm_j}[\widetilde{i}]$), random choices in the oracle structures ($y_1[\widetilde{i}]$, ..., $y_l[\widetilde{i}]$), oracle ($x_1[\widetilde{i}]$, ..., $x_l[\widetilde{i}]$).

Furthermore, the array access $x[M_1, \dots, M_m]$ is allowed only when x is guaranteed to be defined at indices $M_1, \dots, M_m$, either because x is defined syntactically above its usage and $M_1, \dots, M_m$ are the current replication indices at the definition of x, or because $x[M_1, \dots, M_m]$ occurs in the **defined** condition of a **find** above its usage.

To lighten the writing, the array indices can be omitted when they are the current replication indices, writing x instead of $x[M_1, \dots, M_m]$. Hence, array indices can be omitted at all variable definitions (even though the variables are always implicitly arrays).

In indistinguishability axioms $L \approx_p R$, both sides L and R must define the same oracles, with arguments of the same types, and under the same structure of replications with the same bounds N. The random choices may differ between L and R. The facilitate the automatic detection that a game can be implemented by calling oracles of L, the oracle bodies in L are restricted to be of the form **return**(M); there is no such restriction for oracles of R.

2 Translation to EasyCrypt

Our goal is prove in EasyCrypt indistinguishability assumptions $L \approx_p R$ used by CryptoVerif. Therefore, we translate from CryptoVerif to EasyCrypt the games L and R. We first translate CryptoVerif terms to EasyCrypt expressions.

Definition 2.1 (Terms to expressions)**.** *Let M be a CryptoVerif term. The* expression-level translation *of M, written $[\![M]\!]^e$, is defined as follows:*

$$\begin{aligned} [\![i]\!]^e &= i \\ [\![x[M_j]_j]\!]^e &= x_{\mathbf{map}}[[\![M_j]\!]^e]_j \\ [\![f(M_j)_j]\!]^e &= \hat{f}([\![M_j]\!]^e)_j \end{aligned}$$

The translation of a term M, $[\![M]\!]^e = e$, is the EasyCrypt expression e corresponding to M. Each array x in CryptoVerif is associated to a map $x_{\mathbf{map}}$ in EasyCrypt (which maps the array indices to the content of x at these indices), and each function f is associated to its corresponding function $\hat{f}$ in EasyCrypt.

Definition 2.2 (Oracle structures to instructions)**.** *The* instruction-level translation *of an oracle structure is defined in Figures 2, 3, and 4.*

The translation of oracle bodies (Figure 2) $B \rightsquigarrow_{\widetilde{i}} c, V$ translates the oracle body B into the EasyCrypt code c, knowing that the current replication indices at B are $\widetilde{i}$. These indices are used to give correct indices to the variables defined in B. It also collects in the set V all variables declared in B, under the form of triples $(x_{\mathbf{map}}, \widetilde{i}, T)$ representing an EasyCrypt map $x_{\mathbf{map}}$ that maps integer indices $\widetilde{i}$ to elements of type T. This set of variables is used in the toplevel translation (Figure 4) to declare all global EasyCrypt variables.

For simplicity, in the translation of **find**, we only consider a single index and a single branch. We obviously implemented a translation that deals with the general case, along the same ideas. We collect in a list ks the tuples containing the branch number and indices for which the condition succeeds. If ks is empty, we run the **else** branch. Otherwise, we choose randomly an element of ks and execute the corresponding **then** branch.

The translation of oracle structures (Figure 3) $S \rightsquigarrow^O_{\widetilde{i}} c, V$ translates the oracle structure S into the EasyCrypt code c, knowing that the current replication indices at S are $\widetilde{i}$, and O is an EasyCrypt procedure call that initializes all random variables defined above S. O is initially empty (denoted ϵ below). This translation also collects the set of variables declared in S, like the translation of oracle bodies.

[FIND]

$$\frac{B_t \rightsquigarrow_{\widetilde{i}} c_t, V_t \qquad B_e \rightsquigarrow_{\widetilde{i}} c_e, V_e \qquad ks \text{ fresh local variable}}{\textbf{find } u = k \leq N \textbf{ suchthat defined } [x_i[M_{i,j}]_{j<n_i}]_i \wedge M_c \textbf{ then } B_t \textbf{ else } B_e \rightsquigarrow_{\widetilde{i}}}$$

$$\left(\begin{array}{l} ks \leftarrow \left[k \,\middle|\, \bigwedge_i ([\![M_{i,j}]\!]^e)_{j<n_i} \in x_{i\mathbf{map}} \wedge [\![M_c]\!]^e \right] \\ \textbf{if } (ks = []) \ \{ \\ \quad c_e; \\ \} \textbf{ else } \{ \\ \quad u_{\mathbf{map}}[\widetilde{i}] \leftarrow_{\$} \mathcal{U}(ks); \\ \quad c_t; \\ \} \end{array}\right), \{(u_{\mathbf{map}}, \widetilde{i}, \mathbf{int})\} \cup V_t \cup V_e$$

[IF]

$$\frac{B_t \rightsquigarrow_{\widetilde{i}} c_t, V_t \qquad B_e \rightsquigarrow_{\widetilde{i}} c_e, V_e}{\textbf{if } M_c \textbf{ then } B_t \textbf{ else } B_e \rightsquigarrow_{\widetilde{i}} (\textbf{if } ([\![M_c]\!]^e) \ \{c_t\} \textbf{ else } \{c_e\}), V_t \cup V_e}$$

[ASSIGN]

$$\frac{B \rightsquigarrow_{\widetilde{i}} c, V}{x : T \leftarrow M; B \rightsquigarrow_{\widetilde{i}} (x_{\mathbf{map}}[\widetilde{i}] \leftarrow [\![M]\!]^e; c), \{(x_{\mathbf{map}}, \widetilde{i}, T)\} \cup V}$$

[RND]

$$\frac{B \rightsquigarrow_{\widetilde{i}} c, V}{x \leftarrow_{\$} T; B \rightsquigarrow_{\widetilde{i}} (x_{\mathbf{map}}[\widetilde{i}] \leftarrow_{\$} \mathcal{U}(T); c), \{(x_{\mathbf{map}}, \widetilde{i}, T)\} \cup V}$$

[RETURN]

$$\frac{}{\textbf{return}(M_r) \rightsquigarrow_{\widetilde{i}} (\textbf{res} \leftarrow [\![M_r]\!]^e;), \emptyset}$$

Figure 2: Oracle bodies translation

[Foreach]

$$\frac{S \leadsto^{O}_{i::\tilde{i}} c, V}{\textbf{foreach } i \leq n \textbf{ do } S \leadsto^{O}_{\tilde{i}} c, V}$$

[RndPar]

$$\frac{l > 0 \qquad \mathcal{O}_r \text{ fresh oracle name} \qquad S_i \leadsto^{\mathcal{O}_r([j:\textbf{int}]_{j\in\tilde{i}});}_{\tilde{i}} c_i, V_i \text{ for } i \leq m}{x_1 \leftarrow_\$ T_1; \dots x_l \leftarrow_\$ T_l; (S_1 \mid \dots \mid S_m) \leadsto^{O}_{\tilde{i}}}$$

$$\left(\begin{array}{l}
\textbf{proc } \mathcal{O}_r([j:\textbf{int}]_{j\in\tilde{i}}) = \{ \\
\quad O; \\
\quad \textbf{if } (\tilde{i} \notin x_{1\textbf{map}}) \ \{ \\
\qquad [\text{for } i \in [1,\dots,l]\ ] \\
\qquad x_{i\textbf{map}}[\tilde{i}] \leftarrow_\$ \mathcal{U}(T_i) \\
\quad \} \\
\} \\
\\
c_1 \dots c_m
\end{array}\right), \{(x_{i\textbf{map}}, \tilde{i}, T_i)_{i\leq l}\} \cup V_1 \cup \dots \cup V_m$$

[Par]

$$\frac{S_i \leadsto^{O}_{\tilde{i}} c_i, V_i \text{ for } i \leq m}{S_1 \mid \dots \mid S_m \leadsto^{O}_{\tilde{i}} c_1 \dots c_m, V_1 \cup \dots \cup V_m}$$

[Oracle]

$$\frac{r \text{ fresh variable} \qquad B \leadsto_{\tilde{i}} c, V \qquad T \text{ is the type of } B}{\mathcal{O}([x_i : T_i]_i) := B \leadsto^{O}_{\tilde{i}}}$$

$$\left(\begin{array}{l}
\textbf{proc } \mathcal{O}([j:\textbf{int}]_{j\in\tilde{i}}, [x_i : T_i]_i) = \{ \\
\quad \textbf{var res} : T; \\
\quad \textbf{if } (\tilde{i} \notin r) \ \{ \\
\qquad O \\
\qquad [x_{i\textbf{map}}[\tilde{i}] \leftarrow x_i;]_i \\
\qquad c \\
\qquad r[\tilde{i}] \leftarrow \textbf{res}; \\
\quad \} \textbf{ else} \\
\qquad \textbf{res} \leftarrow r[\tilde{i}]; \\
\quad \textbf{return res}; \\
\}
\end{array}\right), \{(r, \tilde{i}, T)\} \cup \{(x_{i\textbf{map}}, \tilde{i}, T_i)_i\} \cup V$$

Figure 3: Oracle structures translation

$$\frac{S \leadsto^{\epsilon}_{\epsilon} c, V}{S \leadsto \begin{pmatrix} \textbf{module } = \{ \\ \quad [\text{for } (x, \widetilde{i}, T) \in V] \\ \quad \textbf{var } x : (\textbf{int}^{|\widetilde{i}|}, T) \textbf{ map} \\ \\ \quad c \\ \} \end{pmatrix}}$$

Figure 4: Top-level translation

The translation of **foreach** just updates the current replication indices.

The translation of random number generations creates an EasyCrypt procedure that initializes these random variables if they are not initialized yet. This procedure first calls O to initialize the random variables above. It serves as O for the translation of oracle structures S_i under the random number generations.

The translation of an oracle creates an EasyCrypt procedure that corresponds to the oracle. It takes as argument the current replication indices and the arguments of the oracle in CryptoVerif. It declares a local variable **res** for the result. It checks that the oracle has not already been called with the same indices, by checking that the map r is not defined at $\widetilde{i}$. The map r is used to store the result of oracle calls. If the oracle has not already been called with the same indices, then it executes O to initialize the random variables above $\mathcal{O}$, it stores the arguments in the corresponding maps, executes the code c corresponding to the oracle body B, and finally adds the obtained result to the map r. If the oracle has already been called with the same indices, then the previous result is recovered from r. Finally, the procedure returns the result **res**.

The toplevel translation (Figure 4) builds a module that declares all maps collected in V and provides all EasyCrypt procedure defined in c.

References

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