EasyCrypt: A Tutorial*

Gilles Barthe[1], François Dupressoir[1], Benjamin Grégoire[2], César Kunz[3,**], Benedikt Schmidt[1], and Pierre-Yves Strub[1]

[1] IMDEA Software Institute, Madrid, Spain
{gilles.barthe,francois.dupressoir,benedikt.schmidt,
pierre-yves.strub}@imdea.org
[2] INRIA Sophia-Antipolis Méditerranée, France
benjamin.gregoire@inria.fr
[3] FireEye, Dresden, Germany
cesar.kunz@gmail.com

1 Introduction

Cryptography plays a key role in the security of modern communication and computer infrastructures; therefore, it is of paramount importance to design cryptographic systems that yield strong security guarantees. To achieve this goal, cryptographic systems are supported by security proofs that establish an upper bound for the probability that a resource-constrained adversary is able to break the cryptographic system. In most cases, security proofs are reductionist, i.e. they construct from an (arbitrary but computationally bounded) adversary that would break the security of the cryptographic construction with some reasonable probability another computationally bounded adversary that would break a hardness assumption with reasonable probability. This approach, known as provable security, is in principle able to deliver rigorous and detailed mathematical proofs. However, new cryptographic designs (and consequently their security analyses) are increasingly complex, and there is a growing emphasis on shifting from algorithmic descriptions to implementation-level descriptions that account for implementation details, recommendations from standards when they exist, and possibly side-channels. As a consequence, cryptographic proofs are becoming increasingly error-prone and difficult to check. One promising solution to address these concerns is to develop machine-checked frameworks that support the construction and automated verification of cryptographic systems. Although many such frameworks exist for the symbolic model of cryptography, comparatively little work has been done to develop machine-checked frameworks to reason directly in the computational model commonly used by cryptographers.

EasyCrypt[1] is an interactive framework for verifying the security of cryptographic constructions in the computational model. EasyCrypt adopts the code-based approach, in which security goals and hardness assumptions are modelled

* An up-to-date and living version of this document and the EasyCrypt formalization and proofs it refers to can be found at https://www.easycrypt.info/Tutorial

** Work performed while the author was working at IMDEA Software Institute.

[1] See http://www.easycrypt.info.

A. Aldini et al. (Eds.): FOSAD VII, LNCS 8604, pp. 146–166, 2014.

as probabilistic programs (called experiments or games) with unspecified adversarial code, and uses tools issued from program verification and programming language theory to rigorously justify cryptographic reasoning. Concretely, EasyCrypt supports common patterns of reasoning from the game-based approach, which decomposes reductionist proofs into a sequence (or possibly tree) of small steps (sometimes called hops) that are easier to understand and to check. As each step relates two programs, one central component of EasyCrypt is a relational Hoare logic for probabilistic programs. The logic, called pRHL, reasons about judments of the form

$$[c_1 \sim c_2 : \Phi \Longrightarrow \Psi]$$

where c_1 and c_2 are probabilistic programs, and Φ and Ψ are relational assertions, i.e. first-order formulae which relate two memories; an instance of a relational assertion is $x\langle 1\rangle = x\langle 2\rangle$, which states that the value of x coincides in both memories. Although pRHL judgments do not explicitly refer to probabilities, it is possible to derive probability claims from valid judgments; indeed, the validity of pRHL judgments is based on a notion of lifting, inspired from probabilistic process algebra, and from which one can derive equalities and inequalities between two probabilities. Specifically, one can derive from valid pRHL judgments of the form

$$[c_1 \sim c_2 : \Phi \Longrightarrow E\langle 1\rangle \rightarrow F\langle 2\rangle]$$

that $\Pr[c_1, m_1 : E] \leq \Pr[c_2, m_2 : F]$ for every initial memories m_1 and m_2 that are related by Φ and events E and F. In addition, pRHL subsumes reasoning about equivalence of probabilistic programs: given a valid judgment of the form

$$\left[c_1 \sim c_2 : \Phi \Longrightarrow \bigwedge_{i=1}^{n} x_i\langle 1\rangle = x_i\langle 2\rangle\right]$$

we have $\Pr[c_1, m_1 : A] = \Pr[c_2, m_2 : A]$ for every initial memories m_1 and m_2 that are related by Φ and event A that only depends on $\{x_1, \ldots, x_n\}$. A useful generalization of observational equivalence is observational equivalence up to a failure event F:

$$\left[c_1 \sim c_2 : \Phi \Longrightarrow \neg F\langle 2\rangle \rightarrow \bigwedge_{i=1}^{n} x_i\langle 1\rangle = x_i\langle 2\rangle\right]$$

It follows from the above judgment that for every initial memories m_1 and m_2 that are related by Φ and event A that only depends on $\{x_1, \ldots, x_n\}$, we have:

$$\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : A] + \Pr[c_2, m_2 : F]$$

In addition to relating the probability of events in different games, cryptographic proofs therefore require the computation of concrete upper bounds on the probability of some event, typically a failure event, in a game. A second component

of EasyCrypt is a probabilistic Hoare logic to reason about the probability of events in games. The logic, called pHL, reasons about judgments of the form

$$[c : \varsigma \Longrightarrow \varphi] \diamond p$$

where c is a probabilistic program, ς and φ are (non-relational) assertions, $\diamond$ is a comparison operator and p is a probability expression.

Both pRHL and pHL are embedded into a higher-order logic in which one can define operators and their associated axioms. Reasoning in this ambient logic is supported by a core proof engine; the proof engine is heavily inspired by the SsReflect extension of Coq, but also enables the use of SMT solvers to discharge proof obligations.

A key challenge for the formalization of cryptographic proofs is to support compositional reasoning. Indeed, many cryptographic systems achieve their functionality by combining (often in intricate ways) different cryptographic constructions, which may themselves be built from several cryptographic primitives. In order to support reasoning about such cryptographic systems, EasyCrypt features a module system which allows the construction of modular proofs that respect the layered and modular design of the cryptographic system. The module system is also useful for structuring large and complex proofs that involve a large number of game hops and perform reductions at different levels.

1.1 Outline

We first recall useful concepts and notations, before presenting a high-level, mathematical overview of the construction and proof developed in this tutorial (Section 2). The objective of the rest of this document is to illustrate our preferred way of specifying cryptographic systems and their proof sketches by constructing a pseudo-random generator (PRG) from a pseudo-random function (PRF). We start by specifying the construction and the desired security notions (Section 3). Finally, we prove formally that our construction is a secure PRG if it is applied to a secure PRF (Section 4).

1.2 Preliminaries

Types, operators and data structures. EasyCrypt's expression language is a higher-order strongly typed functional language. We often view *types* as (non-empty) sets and *operators* as mathematical functions, sometimes using these terms interchangeably. In addition to some basic types (*unit*, *bool*, *int*, *real*), EasyCrypt's libraries provide specifications for some more advanced data structures that can be used when specifying cryptographic systems or when proving their security. We only mention here the types and operators relevant to our formalization.

First, we consider inductive lists, that may be the empty list `[]`, or a value `x::xs` constructed inductively by prepending `x` to the list `xs`. We write `|xs|` to mean the length (or number of elements) of a list `xs`. We sometimes denote `[x]` the list `x::[]`. We define the boolean operator `unique`: α *list* $\rightarrow$ *bool* as the function that returns true if and only if its argument does not have any duplicates.

Our formalization also uses finite maps, that may be indexed by arbitrary types. We use mixfix notations for the map get (m[x]) and map set (m[x] = y) operations, denoting map0 the map that is everywhere undefined. We call the *domain* of a map m the (finite) set of indices on which m is defined.

Discrete probability sub-distributions. EasyCrypt features a type of *discrete probability sub-distributions* that is used to model probabilistic operations, including sampling from a distribution. Informally, a discrete probability distribution over a type A is a function $f : A \to \mathbb{R}$ such that: i. for every $a \in A$, $f\ a \geq 0$; ii. for every finite subset X of A, $\sum_{a \in X} f\ a \leq 1$; iii. the *support* of f, i.e. the set of elements a of A that have a non-zero probability (i.e. $f\ a > 0$) is discrete. Formally, we axiomatize discrete probability sub-distributions by defining for every type t a type t distr, and several operators, including an operator pr: α *distr* $\to (\alpha \to$ *bool*$) \to$ *real* that gives the probability of an event (modelled as a boolean-valued predicate over the carrier type). Moreover, we axiomatize various properties of discrete probability sub-distributions. We introduce some important properties of discrete probability sub-distributions:

- we call *full* sub-distributions whose support is the entire carrier type; conversely, we call *empty* sub-distributions whose support is the empty set,
- we call *lossless* sub-distributions in which the constantly true event has probability 1 (that is, proper distributions),
- we call *sub-uniform* sub-distributions that give the same probability to all elements in their support, using *uniform* to mean *lossless and sub-uniform.*

In the following, we often abuse terminology and use *distribution* to mean *discrete probability sub-distribution.*

2 High Level Description

We start by giving a high level description of the proof described in this tutorial. The idea is to prove that a concretely defined stateful random generator is a pseudo-random generator under the hypothesis that the underlying function is a pseudo-random function. We first introduce the construction, then the different security notions used in the proof.

A Stateful Random Generator. The stateful random generator we use in this chapter is a generic construction parameterized by a function Fc : seed $\to$ state $\to$ state $\times$ output The type seed represent the set of seeds, state is the set of states and output the set of the output returned by the random generator. The code of the construction is described in Figure 1. It is composed of two procedures: an initialization function that sample a seed and an initial state and a generator function generating an output. The generator uses Fc with the seed and the current state to obtain a new state, which is stored in place of the current, and an output which is returned to the caller. Concretely, one could, for example, instantiate the function Fc with AES, using appropriately-sized fixed-length

```
Game SRG =
  procedure init()
    s ←$ seed;
    st ←$ state;
  procedure next()
    (st, r) ← F_c s st;
    return r;
```

Fig. 1. Stateful random generator

bitstrings as seeds, states and outputs. The proof presented here would then directly apply to obtain a security result for this concrete instance.

We would like to prove that the concrete SRG construction is a secure pseudo-random generator (PRG) under reasonable assumptions on Fc. We now define the notion of PRG-security and formalize our assumption on Fc.

Pseudo-Random Generators (PRG). The notion of security for pseudo-random generators is expressed using games $\mathsf{Real}^{\mathrm{PRG}}_{\mathsf{Fc}}$ and $\mathsf{Rand}^{\mathrm{PRG}}_{\mathsf{output}}$ defined in Figure 2. Both games are parameterized by an adversary: a distinguisher D that, given oracle access to a next oracle, returns a bit representing its guess as to whether it is playing against the concrete PRG (game $\mathsf{Real}^{\mathrm{PRG}}_{\mathsf{Fc}}$) or the ideal random generator (game $\mathsf{Rand}^{\mathrm{PRG}}_{\mathsf{output}}$).

```
Game Real^PRG_Fc(D)                 Game Rand^PRG_output(D)
  procedure init()                    procedure init()
    s ←$ seed;
    st ←$ state;
  procedure next()                    procedure next()
    (st, r) ← Fc s st;                  r ←$ output;
    return r;                           return r;
  procedure main()                    procedure main()
    init();                             init();
    b ← D_next();                       b ← D_next();
    return b;                           return b;
```

Fig. 2. PRG security games

Definition 1 (PRG-advantage). *Let* $\mathsf{Fc} : \mathsf{seed} \to \mathsf{state} \to \mathsf{state} \times \mathsf{output}$ *be a function. Let* D *be a distinguisher with an oracle access to a function* next *and returning a bit. The PRG-advantage of* D *against* Fc *is defined as*

$$\mathsf{Adv}^{\mathrm{PRG}}_{Fc}(D) = \Pr\left[\mathsf{Real}^{\mathrm{PRG}}_{Fc}(D) : \mathsf{res}\right] - \Pr\left[\mathsf{Rand}^{\mathrm{PRG}}_{output}(D) : \mathsf{res}\right]$$

Intuitively, a function Fc yields a stateful random generator that is secure when, for all "reasonable" distinguisher D, $\mathsf{Adv}^{\mathrm{PRG}}_{\mathsf{Fc}}(\mathsf{D})$ is "small". Formally defining the notions of "reasonable" and "small" is not the objective of EasyCrypt: we

rather aim at proving *concrete* bounds that can be used to prove security with respect to chosen definitions (for example, parameterizing the system by a security parameter η, "reasonable" adversaries might be algorithms that are p.p.t. in η, and "small" advantages might be negligible as functions of η). However, our proofs still require some restrictions to be placed on the adversaries considered. In particular, we will consider adversaries that make a bounded number of queries to their oracles.

Pseudo-Random Functions (PRF). In our example, the bound for $\mathsf{Adv}^{\mathrm{PRG}}_{\mathsf{Fc}}(\mathsf{D})$ is expressed in terms of the security of Fc seen as a pseudo-random function. We now introduce this notion.

A *function family* is a function $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$, where $\mathcal{K}$ is the set (or type) of keys, $\mathcal{D}$ is the domain and $\mathcal{R}$ the range. We write $F_K(x)$ for $F(K, x)$. This allows us to view the function F as a family of functions from $\mathcal{D}$ to $\mathcal{R}$ indexed by $\mathcal{K}$.

A *pseudo-random function* is a function family that is computationally hard to distinguish from a random function when its key is chosen at random. Formally, this property is expressed using the games $\mathsf{Real}^{\mathrm{PRF}}$ and $\mathsf{Rand}^{\mathrm{PRF}}$ presented in Figure 3. Both games are parameterized by a distinguisher D which is given oracle access to a procedure Fn and returns a bit representing its guess as to which of the two games it is playing. In game $\mathsf{Real}^{\mathrm{PRF}}$, a key K is initially sampled in $\mathcal{K}$, and the procedure Fn is implemented using function F_K. In game $\mathsf{Rand}^{\mathrm{PRF}}$ the procedure Fn implements a lazily sampled random function: on each fresh query x a random value is sampled and stored into the (initially empty) map M, then the associated value is returned to the caller.

Game $\mathsf{Real}^{\mathrm{PRF}}_F(D)$	Game $\mathsf{Rand}^{\mathrm{PRF}}_{\mathcal{R}}(D)$
procedure init()	**procedure** init()
$K \xleftarrow{\$} \mathcal{K}$;	$M \leftarrow \emptyset$;
procedure Fn(x)	**procedure** Fn(x)
return $F_K(x)$;	**if** $M[x] = \bot$ **then** $M[x] \xleftarrow{\$} \mathcal{R}$;
	return $M[x]$;
procedure main()	**procedure** main()
init();	init();
$b \leftarrow D_{\mathsf{Fn}}()$;	$b \leftarrow D_{\mathsf{Fn}}()$;
return b;	**return** b;

Fig. 3. PRF security games

Definition 2 (PRF-advantage). *Let $F : \mathcal{K} \to D \to R$ be a function family. Let D be an adversary with oracle access to a procedure* Fn *and returning a bit. The PRF-advantage of D against F is defined as*

$$\mathsf{Adv}^{\mathrm{PRF}}_F(D) = \Pr\left[\mathsf{Real}^{\mathrm{PRF}}_F(D) : \mathsf{res}\right] - \Pr\left[\mathsf{Rand}^{\mathrm{PRF}}_F(D) : \mathsf{res}\right]$$

High level description of the security proof. The objective of the security proof is to bound, for all D in a certain class of algorithms, $\mathsf{Adv}^{\mathrm{PRG}}_{\mathsf{Fc}}(\mathsf{D})$ as a function of $\mathsf{Adv}^{\mathrm{PRF}}_{\mathsf{Fc}}(\mathsf{D}')$ for some adversary D' constructed from D. More concretely, in Section 4.1, we prove the following abstract probability bound.

Theorem 1 (Abstract Security of our SRG). *For all PRG-distinguisher D, we construct a PRF-distinguisher D^{PRF}_D such that*

$$\mathsf{Adv}^{\mathrm{PRG}}_{Fc}(D) \leq \mathsf{Adv}^{\mathrm{PRF}}_{Fc}(D^{\mathrm{PRF}}_D) + \Pr\left[\mathsf{Rand}^{\mathrm{PRF}}_{\mathcal{R}}(D^{\mathrm{PRF}}_D) : \exists x, 1 < x \# \mathcal{Q}\right],$$

where $\mathcal{Q}$ is the multiset of queries made by D^{PRF}_D to the PRF oracle, and $x \# \mathcal{X}$ is the number of occurrences of element x in multiset $\mathcal{X}$.

In practice, the class of distinguishers considered is restricted to adversaries with access to bounded resources, taking into account running-time and number of queries to the oracles, and limiting the adversary's access to the real and ideal system's memory spaces. In Subsection 4.2, we restrict the class of distinguishers under consideration and compute concrete probability and resource bounds.

Theorem 2 (Concrete Security of our SRG). *For all PRG-distinguisher D that makes at most q_n queries to its* next *oracle, the constructed adversary D^{PRF}_D from Theorem 1 makes at most q_n queries to its PRF oracle, and we have*

$$\Pr\left[\mathsf{Rand}^{\mathrm{PRF}}_{\mathcal{R}}(D^{\mathrm{PRF}}_D) : \exists x, 1 < x \# \mathcal{Q}\right] \leq \frac{q_n^2}{|\mathsf{state}|},$$

where $|\mathsf{state}|$ is the cardinal of the set state.

Remark. The bound from Theorem 2 could be made slightly tighter. We choose to keep this weaker result in this tutorial to keep the proof clear. This and other generalizations and improvements are discussed in the online version of this tutorial.

3 EasyCrypt Specification

In this Section, we formalize in EasyCrypt the definitions given in Section 2. We start by formalizing our concrete SRG construction. We then formalize what it means for an abstract SRG to be a secure PRG, and what it means for a function family to be be a secure PRF. Finally, we instantiate these abstract definitions to our concrete construction and state our security theorem.

3.1 A Stateful Random Generator

First of all we need to declare the types and distributions on which our construction relies. As discussed in Section 2, those are kept abstract throughout this document but can later be instantiated, for example with bitstrings of various fixed lengths, *without having to re-prove anything*, simply by proving that the concrete instantiations given to types and operators fulfill the axioms specified in our formalization. We give an example of such an instantiation (although we do so on other theories) in Section 3.4.

```
type seed.

op dseed: seed distr.
axiom dseed_ll: islossless dseed.

type state.

op dstate: state distr.
axiom dstate_uf: isuniform dstate.
axiom dstate_fu: isfull dstate.

type output.

op dout: output distr.
axiom dout_uf: isuniform dout.

op Fc: seed → state → state * output.
```

Listing 1.1. Core Declarations

The first line declares a new *abstract type* seed representing the set of seeds. The second line declares an abstract operator dseed: a sub-distribution over seed, that we further restrict to be lossless (i.e. a proper distribution) with an axiom dseed_ll. The next lines introduce the types state and output, and uniform distributions over them, also requiring the distribution dstate to be full. Note that this combination of axioms defines dstate uniquely given a finite instantiation for type state. Finally, we declare an operator Fc representing a function family from type state to type state * output and indexed by the type seed.

```
module SRG = {
  var s : seed
  var st : state

  proc init(): unit = {
    s <$- dseed;
    st <$- dstate;
  }

  proc next(): output = {
    var r;

    (st,r) = Fc s st;
    return r;
  }
}.
```

Listing 1.2. Our concrete Stateful Random Generator

Given these basic blocks, we define our stateful random generator as discussed in Section 2: during an initialization phase, a seed and an initial state are sampled

from the specified distributions. Each query for a new random output then simply uses the function Fc applied to the seed and the old state to produce a new state and some output. The new state is stored for use in the next query, and the output is returned.

These procedures are defined as part of a *module*, which also specifies a *memory space*, here composed of two global variables: s of type seed and st of type state. All procedures in a module may access the module's entire memory space and there is no need to pass the current state of global variables around through the return values and arguments of the procedures that use them. EasyCrypt modules are used to formalize schemes, constructions and oracles, but also concrete adversaries, games and security experiments.

We now give generic formalizations of PRG-security and PRF-security that are independent of our concrete construction, as they might appear in EasyCrypt's library of security notions. This library and the instantiation mechanism discussed in Section 3.4 often make it unnecessary to formalize security notions anew for each particular proof.

3.2 Pseudo-Random Generators

We first formalize PRG-security. To allow this notion to later be instantiated to the types declared in Listing 1.1 and our concrete SRG, we wrap the following definitions inside a *theory*: a collection of declarations and definitions, including types, operators, and modules, that can be restricted by axioms (assumptions) and extended with lemmas (derived from the axioms and the language's semantics).

```
theory PRG.
  type output.
  op dout: output distr.

  module type PRG = {
    proc init(): unit
    proc next(): output
  }.

  module type PRGA = { proc next(): output }.
```

Listing 1.3. Pseudo-Random Generators: Types

For any type output equipped with an arbitrary sub-distribution dout, we use a *module type* to define a random generator as a pair of algorithmes $\mathsf{G} = (\mathsf{init}, \mathsf{next})$. A module type specifies a set of procedures that are expected to be provided by a module implementing it. A module is said to *implement* a module type if it provides *at least* all the procedures specified in the type, with the correct types. In particular, our construction from Listing 1.2 implements the PRG module type, but also the module type PRGA, that hides the existence of the init oracle. Module types can be used to quantify over adversaries, or prove generic results on abstract cryptographic constructions before applying

them to concrete instances. In addition to quantification in lemmas, module types enable us to *parameterize* module definitions with abstract modules of a given type. Such module parameters can be used to define *generic constructions* of complex cryptographic schemes from abstract primitives, or to model that an adversary has *oracle access* to some procedure (that is, that it can query the procedure and get the corresponding reply, but cannot interfere with that procedure's internal state or its execution). For example, we consider adversaries that have only oracle access to the next algorithm, which can be formalized using the following set of definitions.

```
  module type Distinguisher(G:PRGA) = { proc distinguish(): bool }.

  module IND(G:PRG,D:Distinguisher) = {
    module D = D(G)

    proc main(): bool = {
      var b;

      G.init();
      b = D.distinguish();
      return b;
    }
  }.

  module PRGi:PRG,PRGA = {
    proc init(): unit = { }
    proc next(): output = { var r; r <$- dout; return r; }
  }.
end PRG.
```

Listing 1.4. Pseudo-Random Generators: Security

A PRG-distinguisher is an algorithm distinguish that, given no inputs, and oracle access to the next procedure of a PRG, returns a boolean. The module type Distinguisher is parameterized with an abstract module G implementing module type PRGA. This means that the implementation of its distinguish procedure may call the procedure G.next.

Given these module type definitions, we can now define an indistinguishability experiment as a module IND, parameterized by a PRG G and a PRG-distinguisher D. In this experiment, we first instantiate the distinguisher's module parameter, ensuring that any query it makes to next is answered using the implementation G.next, we then initialize G and run the distinguisher, returning its output. Security of a PRG G with respect to a given adversary D can then be defined using the standard notion of advantage. Formally, the advantage of an adversary D in distinguishing a PRG G from distribution dout in an initial memory m is written as follows:

$$\mathsf{Adv}^{\mathrm{PRG}}_{\mathsf{G}}(\mathsf{D},\mathsf{m}) = \Pr[\mathsf{IND(G,D).main()} \ @ \ \mathsf{m}: \mathsf{res}] - \Pr[\mathsf{IND(PRGi,D).main()} \ @ \ \mathsf{m}: \mathsf{res}].$$

Given a module M implementing procedure f, and an initial memory m, the expression Pr[M.f() @ m: res] is a real-valued expression whose value is the probability of procedure M.f() returning true when run in initial memory m. The formula appearing after the colon can be arbirary and may mention the global variables of any module currently in scope, as well as the special res variable, which is bound to the procedure's return value. In the rest of this document, we omit memories where irrelevant[2], and also omit the procedure name when it is main, simply writing, say, Pr[IND(G,D): res] for Pr[IND(G,D).main() @ m: res].

3.3 Pseudo-Random Functions

In EasyCrypt, we define pseudo-random functions using the following declarations, leading to the declaration of a function family F, and a module PRFr wrapping F so that it can be queried as an oracle, with a fixed key initially sampled in dK.

```
theory PRF.
  type D.

  type R.

  type K.

  op dK: K distr.
  axiom dK_ll: islossless dK.

  op F: K → D → R.

  module PRFr = {
    var k:K
    proc init(): unit = { k <$ dK; }
    proc f(x:D): R = { return F k x; }
  }.
```

Listing 1.5. Pseudo-Random Functions

The security of a PRF F: K → D → R is defined, as shown below, with respect to a random function from D to R. We write it as expected, using the uniform distribution uR on the full range R to sample output values. The standard definition of a random function from D to R as a function sampled uniformly at random in R^D can be recovered if the domain D is finite.

[2] Formally, these probabilities may in fact depend on the initial memory. In practice, it is always possible to make sure that advantage expressions are in fact independent from the initial memory by initializing all variables before use, and we slightly abuse notations by omitting initial memories.

```
  op uR:R distr.
  axiom uR_uf: is_uniform uR.

  module PRFi = {
    var m:(D,R) map

    proc init(): unit = { m = map0; }

    proc f (x:D): R = {
      if (x ∈ dom m) m[x] <$ uR;
      return (oget m[x]);
    }
  }.

  module type PRF = {
    proc init(): unit
    proc f(x:D): R
  }.

  module type PRFA = {
    proc f(x:D): R
  }.

  module type Distinguisher (F:PRFA) = {
    proc distinguish(): bool
  }.

  module IND(F:PRF,D:Distinguisher) = {
    module D = D(F)

    proc main(): bool = {
      var b;

      F.init();
      b = D.distinguish();
      return b;
    }
  }.
end PRF.
```

Listing 1.6. Pseudo-Random Functions: Security

The advantage of a given D in distinguishing the given PRF F (from Listing 1.5) from a random function in an initial memory m can be expressed as:

$$\mathsf{Adv}_{\mathsf{F}}^{\mathrm{PRF}}(\mathsf{D}) = \Pr[\mathsf{IND}(\mathsf{PRFr},\mathsf{D}).\,\mathsf{res}] - \Pr[\mathsf{IND}(\mathsf{PRFi},\mathsf{D}).\,\mathsf{res}].$$

3.4 Security of Our Stateful Random Generator

We now have enough definitions to properly express our desired security theorem. However, we first need to *instantiate* the abstract PRF and PRG theories with

the types and definitions used in our stateful random generator. We do so using EasyCrypt's *theory cloning* mechanism.

```
clone PRF as PRFa
with
  type D ← state,
  type R ← state * output,
  type K ← seed,
  op dK ← dseed,
  op F ← Fc,
  op uR ← dstate * dout (* product distribution *)
proof *
  (* Proofs omitted *).

module IND^PRF_P = PRFa.IND(P).
module PRFc = PRFa.PRFr.
module PRFi = PRFa.PRFi.
```

Listing 1.7. Security of Fc

The clone instruction creates a copy of the PRF theory defined in Section 3.3, renaming it PRFa, and *instantiating* some of its declared types and operators. For example, we instantiate the abstract operator F from the theory with the function family Fc used in the construction of our SRG, instantiating the domain, range and keyspace accordingly. In addition to instantiating types and operators, the cloning instruction allows us to discharge assumptions about them made in the theory. Here, we discharge all axioms, ensuring that any lemma existing in the PRF theory are *unconditional* lemmas of its PRFa instantiation. After cloning and instantiating the PRF definitions, we define some shorthand notations for its instantiated modules. In particular, we call PRFc the PRFr module where Fc is used. PRF advantage notations in the rest of the paper refer to the advantage in the $\mathsf{IND}^{\mathrm{PRF}}$ game rather than the uninstantiated IND game. Also note that parameterized modules can be *partially applied*: given a module P implementing module type PRF, the partially applied module $\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{P}} = \mathsf{IND}^{\mathrm{PRF}}(\mathsf{P})$ is such that, given a PRF-distinguisher D, $\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{P}}(\mathsf{D}) = \mathsf{IND}^{\mathrm{PRF}}(\mathsf{P},\mathsf{D})$. From now on, we often write the first parameter as an index when applying module expressions.

Similarly, we clone and instantiate the PRG theory with the types used in our construction to easily express the fact that PRGc is a secure PRG. Likewise, PRG advantage notations in the following refer to the instantiated $\mathsf{IND}^{\mathrm{PRG}}$ game.

```
clone PRG as PRGa
with
  type output ← output,
  op dout ← dout.

module IND^PRG_G = PRGa.IND(G).
module PRGi = PRGa.PRGi.
```

Listing 1.8. Security of PRGc

4 EasyCrypt Proof Sketch

4.1 Abstract Bounds for Arbitrary Distinguishers

We first bound the PRG advantage of D as abstract expressions that may not be very meaningful but hold for all D. This proof itself is done in two hops: i. the first hop transforms the SRG construction to make use of the random function PRFi to implement the next oracle instead of Fc; ii. the second hop shows that the PRFi-based implementation of the next oracle is equivalent, up to some well-defined failure event, to the ideal random generator PRGi. We now discuss both steps.

A simple reduction. We want to relate the probability of a distinguisher D to win the game $\mathsf{IND}^{\mathrm{PRG}}$ to the probability of another distinguisher $\mathsf{D}^{\mathsf{PRF}}$ to win the game $\mathsf{IND}^{\mathrm{PRF}}$. To do so we can simply use the game $\mathsf{IND}^{\mathrm{PRG}}$ itself as PRF distinguisher after rewriting SRG as a parameterized module PRGp that uses the PRF oracles instead of calling Fc directly. Anticipating on later proof steps, we also log the queries made by PRGp to the PRF oracle in a list $\mathsf{D}^{\mathrm{PRF}}$.log.

```
module D^PRF(D:PRGa.Distinguisher,F:PRFA) = {
  var log: state list

  module PRGp = {
    proc init(): unit = {
      SRG.st <$- dstate;
      log = [];
    }

    proc next(): output = {
      var r;

      log = SRG.st::log;
      (SRG.st,r) = F.f(SRG.st);
      return r;
    }
  }.

  proc distinguish = IND^PRG_PRGp(D).main
}.
```

Note that the module PRGp does not declare its own memory space, but simply hijacks our initial SRG's global variables. Although this is not necessary, it simplifies invariants slightly by reducing the number of proof artefacts to consider. The following fact is now easy to prove.

Fact 1 *For any PRG distinguisher D whose memory space is disjoint from that of SRG and PRFc, we have*

$$\Pr\left[\mathit{IND}^{\mathrm{PRG}}_{\mathit{SRG}}(D) : \mathsf{res}\right] = \Pr\left[\mathit{IND}^{\mathrm{PRF}}_{\mathit{PRFc}}(D^{\mathrm{PRF}}_{D}) : \mathsf{res}\right]$$

Proof (Sketch). In EasyCrypt, the proof of this lemma makes use of the following pRHL judgment:

$$\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{SRG}}(\mathsf{D}).\mathsf{main} \sim \mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFc}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}).\mathsf{main}: =\{\mathsf{glob\ D}\} \Longrightarrow =\{\mathsf{res}\}$$

The judgment itself is easily discharged by automated tactics after inlining all procedures. Indeed, the procedures are identical except for the fact that the program on the left uses variable SRG.s to store the seed whereas the one on the right uses PRFc.k. The main trick in this proof is to be able to prove the statement for all adversary D. Intuitively we have to compare two evaluations of D, the first uses the next function provided by the module SRG whereas the second uses the next function provided by the module PRGp. Assuming the memory space of D is equal in both evaluations, they can diverge only if: i. D can read values of variables SRG or PRFc but this is impossible due to memory restrictions; ii. the oracles return different results or dissimilar states even when called on identical arguments and in similar states. So, the only point that needs to be proved is that both oracles behave identically.

Once proved, this judgment can be used to prove the probability statement simply by using its semantic interpretation. □

Fact 1 is written as follows in its full EasyCrypt notation.

```
lemma SRG_PRGp (D <: PRGa.Distinguisher {SRG,PRFc}) &m:
  Pr[IND^PRG(SRG,D).main() @ &m: res] =
    Pr[IND^PRF(PRFc,D^PRF(D)).main() @ &m: res].
```

Note that the quantification over the PRG-distinguisher D is made explicit in the lemma, the only restrictions on it being that it implements module type PRGa.Distinguisher and that its memory space (denoted glob D in pRHL statements) is disjoint from those of SRG and PRFc. We also prove this lemma for any initial memory: the variable &m denotes a universally quantified memory. We do not list full EasyCrypt notations for other lemmas, but rather refer the reader to the formalization itself.

Continuing the proof, and using Fact 1 we show:

$$\begin{aligned}
\mathsf{Adv}^{\mathrm{PRG}}_{\mathsf{SRG}}(\mathsf{D}) &= \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{SRG}}(\mathsf{D}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}}(\mathsf{D}) : \mathsf{res}\right] && \text{by definition}\\
&= \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFc}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}}(\mathsf{D}) : \mathsf{res}\right] && \text{by Fact 1}\\
&= \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFc}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFi}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] +\\
&\quad \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFi}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}}(\mathsf{D}) : \mathsf{res}\right]\\
&= \mathsf{Adv}^{\mathrm{PRF}}_{\mathsf{Fc}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) + && \text{by definition}\\
&\quad \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFi}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}}(\mathsf{D}) : \mathsf{res}\right]
\end{aligned}$$

Considering failure events. We now wish to bound the last term

$$\Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFi}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}}(\mathsf{D}) : \mathsf{res}\right].$$

It is clear from their definitions that the two games will only show different behaviours if a duplicate query is made to the PRF by $\mathsf{D}_{\mathsf{D}}^{\mathrm{PRF}}$. Indeed, in this case, the value eventually returned to D is not sampled from dout but rather recalled from the random function's map, whereas the ideal random generator always samples its output freshly. We therefore expect the bound to be the probability of a duplicate query to the random function, which we expressed in Section 2 as $\Pr\left[\mathsf{IND}_{\mathsf{PRFi}}^{\mathrm{PRF}}(\mathsf{D}_{\mathsf{D}}^{\mathrm{PRF}}) : \exists x, 1 < x \# \mathcal{Q}\right]$. Moving slightly away from this high-level description, we use the inductive list $\mathsf{D}^{\mathrm{PRF}}$.log to model multiset $\mathcal{Q}$, using the unique predicate (or rather its negation) to capture the desired event.

However, having the failure event occur only in the first (left) game of the transition would not yield the expected inequality. Indeed directly applying the semantics of equivalence upto failure as presented in Section 1 would lead to a lower-bound rather than the desired upper-bound. Still, an upper-bound can be obtained in this case if the failure event is known to happen with the same probability on both sides of the transitions. Therefore, we also instrument the PRGi module to construct a log of "intermediate states" that it does not otherwise use. In addition, we also make sure that the intermediate state and the output are sampled in the product distribution dstate $*$ dout. This makes this complex game transition easier to prove, by separating two concerns.

```
module PRGi_log = {
  proc init(): unit = {
    SRG.st <$- dstate;
    D^PRF.log = [];
  }

  proc next(): output = {
    var r;

    D^PRF.log = SRG.st :: D^PRF.log;
    (SRG.st,r) <$- dstate * dout;
    return r;
  }
}.
```

First note that PRGi_{log} is indistinguishable from PRGi. Indeed, the additional log and state variables do not alter its control-flow and its output is the second component of a value sampled in the product distribution dstate $*$ dout. In EasyCrypt, we in fact prove that oracles PRGi_{log} and PRGi define the same distribution on the type output regardless of initial state by proving

$$\forall \&m1\ \&m2\ o,\quad \Pr\left[\mathsf{PRGi}_{log}.\mathsf{next}()@\ \&m1 : \mathsf{res} = o\right] = \Pr\left[\mathsf{PRGi}.\mathsf{next}()@\ \&m2 : \mathsf{res} = o\right].$$

This is in fact sufficient to prove that, for any PRG-distinguisher D, we have

$$\Pr\left[\mathsf{IND}_{\mathsf{PRGi}_{log}}^{\mathrm{PRG}}(D) : \mathsf{res}\right] = \Pr\left[\mathsf{IND}_{\mathsf{PRGi}}^{\mathrm{PRG}}(D) : \mathsf{res}\right]$$

and we now only have to prove the following Fact.

Fact 2 (Equivalence upto failure) *For all PRG-distinguisher D whose memory space is disjoint from that of D^{PRF} and SRG, we have the following bound*

$$\Pr\left[IND^{\mathrm{PRF}}_{PRFi}(D^{\mathrm{PRF}}_{D}) : \mathsf{res}\right] - \Pr\left[IND^{\mathrm{PRG}}_{PRGi_{log}}(D) : \mathsf{res}\right] \leq \Pr\left[IND^{\mathrm{PRF}}_{PRFi}(D^{\mathrm{PRF}}_{D}) : !unique\ D^{\mathrm{PRF}}.log\right]$$

Proof (Sketch). We prove in EasyCrypt a single pRHL statement:

$\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFi}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}).\mathsf{main} \sim \mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}_{log}}(\mathsf{D}).\mathsf{main}:$
$=\{\mathsf{glob\ D}\} \Longrightarrow$
$(\mathsf{unique\ D}^{\mathrm{PRF}}.\mathsf{log}\{1\} = \mathsf{unique\ D}^{\mathrm{PRF}}.\mathsf{log}\{2\}) \wedge (\mathsf{unique\ D}^{\mathrm{PRF}}.\mathsf{log}\{2\} \Rightarrow =\{\mathsf{res}\}).$

This pRHL judgement implies two distinct probability relations. The first conjunct in the postcondition implies that the probability of the failure event in both games is equal

$$\Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFi}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : !\mathsf{unique\ D}^{\mathrm{PRF}}.\mathsf{log}\right] = \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}_{log}}(\mathsf{D}) : !\mathsf{unique\ D}^{\mathrm{PRF}}.\mathsf{log}\right] \quad (1)$$

whereas the second conjunct implies the expected inequality

$$\Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFi}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}_{log}}(\mathsf{D}) : \mathsf{res}\right] \leq \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}_{log}}(\mathsf{D}) : !\mathsf{unique\ D}^{\mathrm{PRF}}.\mathsf{log}\right]$$

We then conclude easily. □

Combining Fact 2 and Fact 1's corollary concludes the proof of Theorem 1. Note that this Theorem does not directly imply security in the concrete sense: although we did take care to ensure that our constructed PRF-distinguisher $\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}$ did not have access to the internal memory of PRFc or PRFi (with which it could trivially distinguish the two constructions), we did not bound the number of oracle queries it makes, which may lead to large values of the PRF-advantage and to a large probability of the failure event occurring. We now bound these two quantitites more concretely.

4.2 Application to Resource-Bounded Adversaries

All proof steps seen so far hold regardless of the adversary's resource bounds (running time or number of oracle queries) and only place restrictions on the oracles the adversary can query (via its module type) and on the global variables it may access (via restrictions in the module quantification). However, computing probability and resource bounds requires us to restrict the adversary further, and in particular requires us to limit the number of oracle queries it can make.

Counting oracle queries. First, consider the following module wrappers, that simply count the number of queries made to the PRG ($\mathsf{C}^{\mathrm{PRG}}$) or PRF ($\mathsf{C}^{\mathrm{PRF}}$) oracles.

```
module C^PRG(G:PRG) = {
  var c:int

  proc init(): unit = {
    c = 0;
    G.init();
  }

  proc next(): output = {
    var r;

    r = G.next();
    c = c + 1;
    return r;
  }
}.
```

```
module C^PRF(F:PRF) = {
  var c:int

  proc init(): unit = {
    c = 0;
    F.init();
  }

  proc f(): output = {
    var r;

    r = F.f();
    c = c + 1;
    return r;
  }
}.
```

This wrapper allows us to easily restrict the number of queries made by an adversary to the oracle. In particular, all conditions of the form "D makes at most q queries to the PRG oracle" appearing below can be expressed in EasyCrypt using the following specification, which states that the distinguisher D playing the $\mathsf{IND}^{\mathrm{PRG}}$ game against *any* appropriate G (note the restrictions ensuring that such a D exists) has a probability 1 of making *at most* q queries to the oracle G.next.

$$\begin{array}{l}\forall\ (\mathsf{G} <: \mathsf{PRG}\ \{\mathsf{D},\mathsf{C}^{\mathrm{PRG}}\}),\\ \quad \mathsf{islossless\ G.init} \Rightarrow \mathsf{islossless\ G.next} \Rightarrow\\ \quad \Pr[\mathsf{IND}^{\mathrm{PRG}}(\mathsf{C}^{\mathrm{PRG}}_{\mathsf{G}},\mathsf{D}).\ \mathsf{C}^{\mathrm{PRG}}.\mathsf{c} \leq \mathsf{q}] = 1\end{array}$$

More complex conditions, for example relating the initial value of the counter to its final value even when $\mathsf{D}(\mathsf{C}^{\mathrm{PRG}}_{\mathsf{G}})$.distinguish is run independently of the experiment, can also be expressed in similar ways.

Bounding the failure event. We can now bound the probability of the failure event from Theorem 1 for any PRG-distinguisher D that makes at most q queries to its next oracle.

Lemma 1 (Probability of the failure event). *For all positive integer q, and all PRG-distinguisher D whose memory space is disjoint from those of* $\mathsf{C}^{\mathrm{PRG}}$, *PRFc, PRFi and* D^{PRF} *and that makes at most q queries to its* next *oracle, we have*

$$\Pr\left[IND^{\mathrm{PRF}}_{PRFi}(D^{\mathrm{PRF}}_{D}) : !unique\ D^{\mathrm{PRF}}.log\right] \leq \frac{q^2}{|state|},$$

where $|state|$ *is the cardinal of type* state.

Proof (Sketch). To prove this, we make use of the *failure event lemma*, that intuitively states that, if an event occurs with probability at most p, regardless of state and adversary inputs, during each execution of the oracle, and if the adversary may call this oracle at most q times, then the probability of the event occurring during a full run of the adversary is bounded by $p \cdot q$.

To ease the computation of the probability that the failure event is triggered during a given execution of the oracle, we simplify the computation in two ways: i. we first observe that, from equality (1) obtained during the proof of Fact 2, it is in fact sufficient to bound the probability $\Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}_{log}}(\mathsf{D}) : !\mathsf{unique}\ \mathsf{D}^{\mathrm{PRF}}.\mathsf{log}\right]$; and ii. we soundly approximate the failure event with a more general one, triggered as soon as a state that already appears in the log is sampled, rather than when it is added to the log.

```
proc next(): output = {
  var r;

  log = SRG.st::log;
  (SRG.st,r) ←$ dstate ∗ dout;
  bad = bad ∨ mem SRG.st log;
  return r;
}
```

Clearly, if bad is initially unset, we can prove that bad has to be set for the log to have duplicates. This implication is sufficient to prove, in pRHL:

$$\Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}_{log}}(\mathsf{D}) : !\mathsf{unique}\ \mathsf{D}^{\mathrm{PRF}}.\mathsf{log}\right] \leq \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{PRGi}_{log}}(\mathsf{D}) : \mathsf{bad}\right].$$

In this latest variant of the next oracle, it becomes clear that the probability of the bad event being triggered during a given query knowing that it was not true beforehand is exactly the probability that a freshly sampled state already appears in the log. Since the log until that point does not have duplicates and dstate is uniform and full, this probability is exactly $\frac{|\mathsf{log}|}{|\mathsf{state}|}$. Given that the log's size is bounded by q, we obtain the desired bound. □

Bounding the resources of the PRF-distinguisher. Finally, we prove that for any bounded PRG-distinguisher D with no access to the memory of the primitive modules, the generic construction $\mathsf{D}^{\mathrm{PRF}}$ and both $\mathsf{C}^{\mathrm{PRG}}$ modules, the constructed PRF-distinguisher $\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}$ makes at most as many queries to the PRF as D did to the PRG.

Lemma 2 (Number of PRF queries). *For all positive integer q, and all PRG-distinguisher D whose memory space is disjoint from those of $\mathsf{C}^{\mathrm{PRG}}$, $\mathsf{C}^{\mathrm{PRF}}$, PRFc, PRFi and SRG, and that makes at most q oracle queries, the constructed PRF-distinguisher $\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}$ makes at most q oracles queries.*

Proof (Sketch). Given an integer q and PRG-distinguisher D as constrained above, we prove the following equality, which states that (for all initial memory)

the probability that the constructed $\mathsf{D}_{\mathsf{D}}^{\mathrm{PRF}}$ makes at most q queries to $\mathsf{PRFc.f}$ during a run of the $\mathsf{IND}^{\mathrm{PRF}}$ experiment is 1.

$$\Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{C}^{\mathrm{PRF}}(\mathsf{PRFc})}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{C}^{\mathrm{PRF}}.\mathsf{c} \leq \mathsf{q}\right] = 1.$$

We do so by proving the following statement, inlining all functions and noting that the counters remain synchronized through the execution.

$$\Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{C}^{\mathrm{PRF}}(\mathsf{PRFc})}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{C}^{\mathrm{PRF}}.\mathsf{c} \leq \mathsf{q}\right] = \Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{C}^{\mathrm{PRG}}(\mathsf{SRG})}(D) : \mathsf{C}^{\mathrm{PRG}}.\mathsf{c} \leq \mathsf{q}\right]$$

We conclude by applying the assumption on D, rewriting the right-hand-side of this equality into 1. □

This concludes the proof of Theorem 2.

5 Further Reading and Concluding Remarks

EasyCrypt provides tool-assisted support for building and verifying machine-checked cryptographic proofs. Its foundations are based on a probabilistic relational Hoare logic, pRHL, that was first introduced in [5], and a verification condition generator that was first presented in [4]. Another key component of EasyCrypt is its module system, which supports the formalization of complex and layered proofs and has been used for instance to verify the security of protocols for secure function evaluation and verifiable computation [2]. One main motivation for the development of EasyCrypt is to close the gap between security proofs and implementations; an approach based on certified compilers is presented in [1]. Beyond EasyCrypt, it is possible to develop and apply fully automated verification techniques for analyzing the security of classes of cryptographic constructions; for instance, one can use customized logics to reason about the security of padding-based encryption schemes, i.e. public-key encryption schemes built from one-way trapdoor permutations and random oracles [3]. Deduction rules of the logic capture high-level reasoning principles that can be formalized in EasyCrypt, and contribute to building an extensive library of common reasoning patterns in cryptography. It is our hope that the development of the library will somewhat shift the focus of EasyCrypt proofs to reduce the emphasis on proving pRHL judgments and to bring them closer to the high level reasoning steps used by cryptographers.

References

1. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F.: Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In: ACM Communications and Computer Security (CCS), pp. 1217–1230. ACM (2013)

2. Almeida, J.B., Barbosa, M., Barthe, G., Davy, G., Dupressoir, F., Grégoire, B., Strub, P.-Y.: Verified implementations for secure and verifiable computation. Cryptology ePrint Archive, Report 2014/456 (2014), http://eprint.iacr.org/
3. Barthe, G., Crespo, J.M., Grégoire, B., Kunz, C., Lakhnech, Y., Schmidt, B., Béguelin, S.Z.: Fully automated analysis of padding-based encryption in the computational model. In: ACM Communications and Computer Security (CCS), pp. 1247–1260. ACM (2013)
4. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
5. Barthe, G., Grégoire, B., Zanella-Béguelin, S.: Formal certification of code-based cryptographic proofs. In: ACM Principles of Programming Languages (POPL), pp. 90–101. ACM (2009)