

# EasyCrypt: A Tutorial\*

Gilles Barthe<sup>1</sup>, François Dupressoir<sup>1</sup>, Benjamin Grégoire<sup>2</sup>, César Kunz<sup>3\*\*</sup>,  
Benedikt Schmidt<sup>1</sup>, and Pierre-Yves Strub<sup>1</sup>

<sup>1</sup> IMDEA Software Institute, Madrid, Spain

{gilles.barthe, francois.dupressoir, benedikt.schmidt, pierre-yves.strub}@imdea.org

<sup>2</sup> INRIA Sophia-Antipolis Méditerranée, France

benjamin.gregoire@inria.fr

<sup>3</sup> FireEye

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, that is, 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 supporting direct reasoning in the computational model commonly used by cryptographers.

EasyCrypt<sup>4</sup> 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.

<sup>4</sup> See <https://www.easycrypt.info>.

as probabilistic programs (called *experiments* or *games*) with unspecified adversarial code. `EasyCrypt` uses formal tools 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 transitions 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 judgments of the form

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

where  $c_1$  and  $c_2$  are probabilistic programs, and  $\Phi$  and  $\Psi$  are relational assertions, that is, 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.<sup>5</sup> 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  $\Phi$  and events  $E$  and  $F$ .<sup>6</sup> Clearly, `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  $\Phi$  and event  $A$  that only depends on  $\{x_1, \dots, x_n\}$ .

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  $\Phi$  and event  $A$  that only depends on  $\{x_1, \dots, x_n\}$ , 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

<sup>5</sup> Formal details were described by Barthe, Grégoire and Zanella-Béguelin [5].

<sup>6</sup> Where  $\Pr [c, m : E]$  denotes the probability of  $E$  in the distribution (on memories) produced by running  $c$  on initial memory  $m$ .

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,  $\varsigma$  and  $\varphi$  are (non-relational) assertions,  $\diamond$  is a comparison operator ( $\leq$ ,  $\geq$ ,  $<$ ,  $>$ , or  $=$ ) 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 automatically discharge low-level 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 proof steps 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, and invite the reader to explore the standard library distributed with `EasyCrypt` (in `theories/datatypes`).

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 `[x1;...;xn]` the list `x1::...::xn::[]`. We define the boolean operator `uniq`:  $\alpha \text{ list} \rightarrow \text{bool}$  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 `fmap0` the map that is everywhere undefined. We call the *domain* of a map `m` the (finite) set of indices `dom m` 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. A discrete probability distribution over a type  $A$  is a function  $f : A \rightarrow \mathbb{R}$  such that: i. for every  $a \in A$ ,  $0 \leq f a$ ; and ii. for every finite subset  $X$  of  $A$ ,  $\sum_{x \in X} f x \leq 1$ . Formally, we equip every type  $A$  with the type  $A$  `distr` of discrete sub-distributions over  $A$ , namely the subtype of  $A \rightarrow \mathbb{R}$  defined above. Given  $d \in A$  `distr`, we denote  $\mu_x d$  the probability mass function of  $d$  (when  $d$  is an injection of some valid probability mass function  $f$  into the subtype, we have  $\mu_x d = f$ ), and define the probability of an event (or predicate)  $E \in A \rightarrow \text{bool}$  in  $d$  as  $\mu d E = \sum_{\{a \in A \mid E a\}} \mu_x d a$ . 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 the *empty* sub-distribution for carrier type  $\tau$  the sub-distribution on  $\tau$  whose support is the empty set,
- we call *lossless* sub-distributions in which the constantly true event has probability 1 (that is, proper distributions), and
- we call *sub-uniform* sub-distributions that give the same mass 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 cannot be computationally distinguished from a true random generator provided it is constructed from a secure 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  $Fc : \text{seed} \rightarrow \text{state} \rightarrow \text{state} \times \text{output}$ . 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

```

Game SRG =
  procedure init()
     $s \xleftarrow{\$}$  seed;
     $st \xleftarrow{\$}$  state;
  procedure next()
     $(st, r) \leftarrow \text{Fc } s \ st;$ 
    return  $r$ ;

```

**Fig. 1.** Stateful random generator

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  $\text{Fc}$  with AES, using appropriately-sized fixed-length 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  $\text{Fc}$ . We now define the notion of PRG-security and formalize our assumption on  $\text{Fc}$ .

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

<pre> Game <math>\text{Real}_{\text{Fc}}^{\text{PRG}}(D)</math>   <u>procedure</u> init()     <math>s \xleftarrow{\\$}</math> seed;     <math>st \xleftarrow{\\$}</math> state;   <u>procedure</u> next()     <math>(st, r) \leftarrow \text{Fc } s \ st;</math>     return <math>r</math>;   <u>procedure</u> main()     init();     <math>b \leftarrow D_{\text{next}}();</math>     return <math>b</math>; </pre>	<pre> Game <math>\text{Rand}_{\text{output}}^{\text{PRG}}(D)</math>   <u>procedure</u> init()    <u>procedure</u> next()     <math>r \xleftarrow{\\$}</math> output;     return <math>r</math>;   <u>procedure</u> main()     init();     <math>b \leftarrow D_{\text{next}}();</math>     return <math>b</math>; </pre>
--	--

**Fig. 2.** PRG security games

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

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

Intuitively, a function  $F_c$  yields a stateful random generator that is secure when, for all “reasonable” distinguisher  $D$ ,  $\text{Adv}_{F_c}^{\text{PRG}}(D)$  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  $\eta$ , “reasonable” adversaries might be algorithms that are p.p.t. in  $\eta$ , and “small” advantages might be negligible as functions of  $\eta$ ). 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  $\text{Adv}_{F_c}^{\text{PRG}}(D)$  is expressed in terms of the PRF security of  $F_c$ . We now introduce this notion.

A *function family from  $\mathcal{D}$  to  $\mathcal{R}$  indexed by  $\mathcal{K}$*  is a function  $F : \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{R}$ , 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  $F(K, x)$ .

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  $\text{Real}^{\text{PRF}}$  and  $\text{Rand}^{\text{PRF}}$  presented in Figure 3. Both games are parameterized by a distinguisher  $D$  which is given oracle access to a procedure  $\text{Fn}$  and returns a bit representing its guess as to which of the two games it is playing. In game  $\text{Real}^{\text{PRF}}$ , a key  $K$  is initially sampled in  $\mathcal{K}$ , and the procedure  $\text{Fn}$  is implemented using function  $F_K$ . In game  $\text{Rand}^{\text{PRF}}$  the procedure  $\text{Fn}$  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.

<p>Game <math>\text{Real}_F^{\text{PRF}}(D)</math></p> <pre> <u>procedure</u> init()   <math>K \leftarrow \mathcal{K}</math>; <u>procedure</u> Fn(<math>x</math>)   return <math>F_K(x)</math>;  <u>procedure</u> main()   init();   <math>b \leftarrow D_{\text{Fn}}()</math>;   return <math>b</math>; </pre>	<p>Game <math>\text{Rand}_F^{\text{PRF}}(D)</math></p> <pre> <u>procedure</u> init()   <math>M \leftarrow \emptyset</math>; <u>procedure</u> Fn(<math>x</math>)   if <math>M[x] = \perp</math> then <math>M[x] \leftarrow \mathcal{R}</math>;   return <math>M[x]</math>; <u>procedure</u> main()   init();   <math>b \leftarrow D_{\text{Fn}}()</math>;   return <math>b</math>; </pre>
---	--

**Fig. 3.** PRF security games

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

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

*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,  $\text{Adv}_{\mathcal{F}_c}^{\text{PRG}}(D)$  as a function of  $\text{Adv}_{\mathcal{F}_c}^{\text{PRF}}(D')$  for some explicitly constructed adversary  $D'$ . 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  $D_D^{\text{PRF}}$  that uses  $D$  as a sub-procedure and such that*

$$\text{Adv}_{\mathcal{F}_c}^{\text{PRG}}(D) \leq \text{Adv}_{\mathcal{F}_c}^{\text{PRF}}(D_D^{\text{PRF}}) + \Pr [\text{Rand}_{\mathcal{R}}^{\text{PRF}}(D_D^{\text{PRF}}) : \exists x, 1 < x \# \mathcal{Q}],$$

where  $\mathcal{Q}$  is the multiset of queries made by  $D_D^{\text{PRF}}$  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  $D_D^{\text{PRF}}$  from Theorem 1 makes at most  $q_n$  queries to its PRF oracle, and we have*

$$\Pr [\text{Rand}_{\mathcal{R}}^{\text{PRF}}(D_D^{\text{PRF}}) : \exists x, 1 < x \# \mathcal{Q}] \leq \frac{(q_n + 1) \cdot q_n}{2 \cdot |\text{state}|},$$

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

*Remark.* The bound from Theorem 2 could be made *very slightly* tighter (with the bound becoming  $\frac{q_n \cdot (q_n - 1)}{2 \cdot |\text{state}|}$ ). We choose to keep the weaker result from Theorem 2 in this tutorial to keep the proof clear. The associated proof file hints at a way of tightening the bound, left as an exercise to the advanced reader.

### 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 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 | is_lossless dseed } as dseed_ll.

type state.
op dstate: { state distr | is_uniform dstate  $\wedge$  is_full dstate } as dstate_uffu.

type output.
op dout: { output distr | isuniform dout } as dout_uf.

op Fc: seed  $\rightarrow$  state  $\rightarrow$  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 (that is, 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 and restricts type `state` to being finite. Finally, we declare an operator `Fc` representing a function family from type `state` to type `state * output` and indexed by the type `seed`.

Given these abstract symbols, 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.

---

```

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

  proc init(): unit = {
    s  $\stackrel{\$}{\leftarrow}$  dseed;
    st  $\stackrel{\$}{\leftarrow}$  dstate;
  }

  proc next(): output = {
    var r;

    (st,r)  $\leftarrow$  Fc s st;
    return r;
  }
}.

```



---

**Listing 1.2.** Our concrete Stateful Random Generator

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.<sup>7</sup> This library and the instantiation mechanism discussed in Section 3.4 often makes it unnecessary to formalize security notions anew for each particular proof. In this document, we define PRG and PRF as subtheories of the top-level working environment. In the practical development, they are defined as separated files (*PRF\_eca* and *PRG\_eca*) that are *required* (or loaded) into the EasyCrypt proof environment.

### 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).

---

```
abstract theory PRG.  
  type output.  
  op dout: output distr.  
  
  module type RG = {  
    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 algorithms  $G = (\text{init}, \text{next})$ . 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

---

<sup>7</sup> The definitions shown here are not from the EasyCrypt library, although this may change once it stabilizes.

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 may not interfere with that procedure’s internal state or its execution). For example, we consider adversaries that have only oracle access to the next algorithm. They can be formalized using the following set of definitions.

---

```

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

module IND(G:PRG,D:Distinguisher) = {
  proc main(): bool = {
    var b;

    G.init();
    b  $\stackrel{\textcircled{D}}{\leftarrow}$  D(G).distinguish();
    return b;
  }
}.

module PRGi:PRG,PRGA = {
  proc init(): unit = { }
  proc next(): output = { var r; r  $\stackrel{\textcircled{S}}{\leftarrow}$  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:

$$\text{Adv}_G^{\text{PRG}}(D, m) = \Pr[\text{IND}(G, D).main() \text{ @ } m: \text{res}] - \Pr[\text{IND}(\text{PRG}_i, D).main() \text{ @ } m: \text{res}].$$

Given a module  $M$  implementing procedure  $f$ , and an initial memory  $m$ , the expression  $\Pr[M.f() \text{ @ } m: \text{res}]$  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 arbitrary and may mention the global variables of any module currently in scope, as well as the special  $\text{res}$  variable, which is bound to the procedure's return value. In the rest of this document, we omit memories where irrelevant<sup>8</sup>, and also omit the procedure name when it is `main`, simply writing, say,  $\Pr[\text{IND}(G, D): \text{res}]$  for  $\Pr[\text{IND}(G, D).main() \text{ @ } m: \text{res}]$ .

### 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  $dK$ .

---

```

abstract theory PRF.
  type D, R, K.

  op dK: { K distr | is_lossless dK } as dK_ll.

  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  $F: K \rightarrow D \rightarrow R$  is defined, as shown below, with respect to a random function from  $D$  to  $R$ . We write it as expected, using a uniform distribution  $uR$  on some subset of  $R$  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.

---

```

op uR: { R distr | is_uniform uR } as uR_uf.

```

---

<sup>8</sup> 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.

```

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) = {
  proc main(): bool = {
    var b;

    F.init();
    b ←ⓐ D(F).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:

$$\text{Adv}_F^{\text{PRF}}(D) = \Pr[\text{IND}(\text{PRFr}, D): \text{res}] - \Pr[\text{IND}(\text{PRFi}, D): \text{res}].$$

### 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  $\leftarrow$  state,
  type R  $\leftarrow$  state * output,
  type K  $\leftarrow$  seed,
  op dK  $\leftarrow$  dseed,
  op F  $\leftarrow$  Fc,
  op uR  $\leftarrow$  dstate '*' dout (* product distribution *)
proof *.
(* Proofs omitted *)

module INDPPRF = 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 IND<sup>PRG</sup> 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 IND<sub>P</sub><sup>PRF</sup> = IND<sup>PRF</sup>(P) is such that, given a PRF-distinguisher D, IND<sub>P</sub><sup>PRF</sup>(D) = IND<sup>PRF</sup>(P,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 IND<sup>PRG</sup> game.

```

clone PRG as PRGa
with
  type output  $\leftarrow$  output,
  op dout  $\leftarrow$  dout.

module INDGPRG = 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  $F_c$ ; 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  $\text{IND}^{\text{PRG}}$  to the probability of another distinguisher  $D^{\text{PRF}}$  to win the game  $\text{IND}^{\text{PRF}}$ . To do so we can simply use the game  $\text{IND}^{\text{PRG}}$  itself as PRF distinguisher after rewriting SRG as a parameterized module PRGp that uses the PRF oracles instead of calling  $F_c$  directly. Anticipating on later proof steps, we also log the queries made by PRGp to the PRF oracle in a list  $D^{\text{PRF}}.\text{log}$ .

---

```
module  $D^{\text{PRF}}$ ( $D$ :PRGa.Distinguisher, $F$ :PRFA) = {  
  var log: state list  
  
  module PRGp = {  
    proc init(): unit = {  
      SRG.st  $\stackrel{\$}{\leftarrow}$  dstate;  
      log  $\leftarrow$  [];  
    }  
  
    proc next(): output = {  
      var r;  
  
      log  $\leftarrow$  SRG.st::log;  
      (SRG.st,r)  $\stackrel{\textcircled{\$}}{\leftarrow}$  F.f(SRG.st);  
      return r;  
    }  
  }  
  
  proc distinguish =  $\text{IND}_{\text{PRGp}}^{\text{PRG}}(D).\text{main}$   
}
```

---

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*

$$\Pr [\text{IND}_{\text{SRG}}^{\text{PRG}}(D) : \text{res}] = \Pr [\text{IND}_{\text{PRFc}}^{\text{PRF}}(D_D^{\text{PRF}}) : \text{res}]$$

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

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

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 that 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 restriction; ii. the oracles return different results or dissimilar states even when called on identical arguments and in similar states. So, the only point that need to be proved is that both oracle behave identical.

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

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

$$\begin{aligned} \text{lemma } \text{SRG\_PRGp } (\text{D} <: \text{PRGa.Distinguisher } \{\text{SRG,PRFc}\}) \ \&\text{m}: \\ \text{Pr}[\text{IND}_{\text{SRG}}^{\text{PRG}}(\text{SRG},\text{D}).\text{main}() \ @ \ \&\text{m} : \text{res}] = \\ \text{Pr}[\text{IND}_{\text{PRFc}}^{\text{PRF}}(\text{PRFc},\text{D}_D^{\text{PRF}}).\text{main}() \ @ \ \&\text{m} : \text{res}]. \end{aligned}$$

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, which is made available from the web page corresponding to this tutorial.<sup>9</sup>

Continuing the proof, and using Fact 1 we show:

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

<sup>9</sup> <https://www.easycrypt.info/trac/Tutorial>

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

$$\Pr [\text{IND}_{\text{PRFi}}^{\text{PRF}}(\text{D}_D^{\text{PRF}}) : \text{res}] - \Pr [\text{IND}_{\text{PRGi}}^{\text{PRG}}(\text{D}) : \text{res}] .$$

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  $\text{D}_D^{\text{PRF}}$ . Indeed, in this case, the value eventually returned to  $\text{D}$  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  $\Pr [\text{IND}_{\text{PRFi}}^{\text{PRF}}(\text{D}_D^{\text{PRF}}) : \exists x, 1 < x \# \mathcal{Q}]$ . Moving slightly away from this high-level description, we use the inductive list  $\text{D}^{\text{PRF}}.\text{log}$  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 PRGilog = {
  proc init(): unit = {
    SRG.st  $\stackrel{\$}{\leftarrow}$  dstate;
    DPRF.log  $\leftarrow$  [];
  }

  proc next(): output = {
    var r;

    DPRF.log  $\leftarrow$  SRG.st :: DPRF.log;
    (SRG.st,r)  $\stackrel{\$}{\leftarrow}$  dstate * dout;
    return r;
  }
}

```

---

First note that  $\text{PRGi}_{\text{log}}$  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  $\text{PRGi}_{\text{log}}$  and PRGi define the same distribution on the type `output` regardless of initial state by proving

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



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

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

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  $D^{\text{PRF}}$  and SRG, we have the following bound*

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

*Proof.* We prove in EasyCrypt a single pRHL statement:

$$\begin{aligned} \text{IND}_{\text{PRFi}}^{\text{PRF}}(D_D^{\text{PRF}}).\text{main} &\sim \text{IND}_{\text{PRGi}_{\text{log}}}^{\text{PRG}}(D).\text{main} \\ &= \{\text{glob } D\} \implies \\ &(\text{uniq } D^{\text{PRF}}.\text{log}\{1\} = \text{uniq } D^{\text{PRF}}.\text{log}\{2\}) \wedge (\text{uniq } D^{\text{PRF}}.\text{log}\{2\} \Rightarrow =\{\text{res}\}). \end{aligned}$$

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

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

whereas the second conjunct implies the expected inequality

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

We then conclude easily.  $\square$

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  $D_D^{\text{PRF}}$  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 quantities 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 ( $C^{\text{PRG}}$ ) or PRF ( $C^{\text{PRF}}$ ) oracles.

<pre> <b>module</b> <math>C^{\text{PRG}}</math>(G:PRG) = {   <b>var</b> c:int    <b>proc</b> init(): <i>unit</i> = {     c <math>\leftarrow</math> 0;     G.init();   }    <b>proc</b> next(): output = {     <b>var</b> r;      r <math>\stackrel{\textcircled{a}}</math> G.next();     c <math>\leftarrow</math> c + 1;     <b>return</b> r;   } }. </pre>	<pre> <b>module</b> <math>C^{\text{PRF}}</math>(F:PRF) = {   <b>var</b> c:int    <b>proc</b> init(): <i>unit</i> = {     c <math>\leftarrow</math> 0;     F.init();   }    <b>proc</b> f(): output = {     <b>var</b> r;      r <math>\stackrel{\textcircled{a}}</math> F.f();     c <math>\leftarrow</math> c + 1;     <b>return</b> r;   } }. </pre>
--	--

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  $\text{IND}^{\text{PRG}}$  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  $G.\text{next}$ .

$$\forall (G <: \text{PRG } \{D, C^{\text{PRG}}\}),$$

$$\text{islossless } G.\text{init} \Rightarrow \text{islossless } G.\text{next} \Rightarrow$$

$$\Pr[\text{IND}^{\text{PRG}}(C_G^{\text{PRG}}, D): C^{\text{PRG}}.c \leq q] = 1$$

More complex conditions, for example relating the initial value of the counter to its final value even when  $D(C_G^{\text{PRG}}).\text{distinguish}$  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  $C^{\text{PRG}}$ ,  $\text{PRFc}$ ,  $\text{PRFi}$  and  $D^{\text{PRF}}$  and that makes at most  $q$  queries to its next oracle, we have*

$$\Pr \left[ \text{IND}_{\text{PRFi}}^{\text{PRF}}(D_D^{\text{PRF}}) : \text{!uniq } D^{\text{PRF}}.\text{log} \right] \leq \frac{(q+1) \cdot q}{2 \cdot |\text{state}|},$$

where  $|\text{state}|$  is the cardinal of type state.

*Proof.* To prove this, we make use of the *failure event lemma*, that intuitively states that, if an event occurs with probability at most  $p_i$ , regardless of state and adversary inputs, during the  $i$ th oracle query, 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  $\sum_{0 \leq i < q} p_i$ .

To ease the computation of the probability that the failure event is triggered during a given execution of the oracle, we first overapproximate it, triggering a failure as soon as a colliding response is sampled, even if it is not used. This weakens the bound, since a collision in the  $q$ th response would not give any distinguishing advantage, but makes the formal proof slightly easier. Formally, we prove the following::

$$\Pr \left[ \text{IND}_{\text{PRGi}_{\text{og}}}^{\text{PRG}}(D) : \text{!uniq } D^{\text{PRF}}.\text{log} \right] \leq \Pr \left[ \text{IND}_{\text{PRGi}_{\text{og}}}^{\text{PRG}}(D) : \text{!uniq } (\text{SRG.st}::D^{\text{PRF}}.\text{log}) \right].$$

The probability of this generalized failure event during the  $i$ th query is much easier to compute, as it is simply the probability that a freshly sampled output already appears in a duplicate-free list of size  $i$ . Since  $\text{dstate}$  is uniform and full, the probability of the generalized failure event being triggered during the  $i$ th query is exactly  $\frac{i}{|\text{state}|}$ .  $\square$

*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  $D^{\text{PRF}}$  and both  $C^{\text{PRG}}$  modules, the constructed PRF-distinguisher  $D_D^{\text{PRF}}$  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  $C^{\text{PRG}}$ ,  $C^{\text{PRF}}$ ,  $\text{PRFc}$ ,  $\text{PRFi}$  and  $\text{SRG}$ , and that makes at most  $q$  oracle queries, the constructed PRF-distinguisher  $D_D^{\text{PRF}}$  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  $D_D^{\text{PRF}}$  makes at most  $q$  queries to  $\text{PRF.c}$  during a run of the  $\text{IND}^{\text{PRF}}$  experiment is 1.

$$\Pr \left[ \text{IND}_{C^{\text{PRF}}(\text{PRF.c})}^{\text{PRF}}(D_D^{\text{PRF}}) : C^{\text{PRF}}.c \leq q \right] = 1.$$

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

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

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

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. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *ACM Communications and Computer Security (CCS)*, pages 1217–1230. ACM, 2013. 20

2. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Guillaume Davy, François Dupressoir, Benjamin Grégoire, and Pierre-Yves Strub. Verified implementations for secure and verifiable computation. Cryptology ePrint Archive, Report 2014/456, 2014. <http://eprint.iacr.org/>. 20
3. Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, Yassine Lakhnech, Benedikt Schmidt, and Santiago Zanella Béguelin. Fully automated analysis of padding-based encryption in the computational model. In *ACM Communications and Computer Security (CCS)*, pages 1247–1260. ACM, 2013. 20
4. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology — CRYPTO*, volume 6841 of *Lectures Notes in Computer Science*, pages 71–90. Springer-Verlag, 2011. 20
5. Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *ACM Principles of Programming Languages (POPL)*, pages 90–101. ACM, 2009. 2, 20