

# SoK: Security Models for Pseudo-Random Number Generators

Sylvain Ruhault

Oppida, 6 avenue du Vieil Etang, 78180 Montigny Le Bretonneux, France

[sylvain.ruhault@oppida.fr](mailto:sylvain.ruhault@oppida.fr)

**Abstract.** Randomness plays an important role in multiple applications in cryptography. It is required in fundamental tasks such as key generation, masking and hiding values, nonces and initialization vectors generation. Pseudo-random number generators have been studied by numerous authors, either to propose clear security notions and associated constructions or to point out potential vulnerabilities. In this systematization of knowledge paper, we present the three notions of generators that have been successively formalized: standard generators, stateful generators and generators with input. For each notion, we present expected security properties, where adversaries have increasing capabilities (including access to partial information on the internal variables) and we propose secure and efficient constructions, all based on the block cipher AES. In our description of generators with input, we revisit the notions of accumulator and extractor and we point out that security crucially relies on the independence between the randomness source and the seeds of the accumulator and the extractor. To illustrate this requirement, we identify a potential vulnerability of the NIST standard CTR\_DRBG.

**Keywords:** Pseudo-random number generation · Security Models · Entropy

## 1 Introduction

### 1.1 Security Notions

The first simple notion is for a **standard pseudo-random number generator** (standard PRNG). A secure standard pseudo-random number generator is formalized as an extending function, that on input a random bit string (called a *seed*), outputs a longer bit string which is indistinguishable from random. The output of the algorithm cannot be perfectly random, as there are fewer seeds than possible outputs, so one can define a security objective for this algorithm as follows: no computationally-bounded adversary, which does not know the seed, can distinguish an output from the uniform. The model is described in Section 2.

The generation of a random seed can be amortized allowing the computation of several outputs with the same seed. As the algorithm is deterministic, this implies that the algorithm also modifies the seed for each output. This class of algorithm can also be defined precisely with a formal security game and is referred to as a **stateful pseudo-random number generator**. Its security is formalized by the indistinguishability from random of all the outputs generated from a secret seed. In this situation, as the seed is reused, the generator needs to store it between the generation of two outputs. This design has been implemented in a large number of systems, including hardware security modules. Several attacks have been mounted against some generators, that rely on the predictability of the seed or on the potential leakage of the memory of the generator. The memory of the generator (usually named its *internal state*) is its most critical part, as an adversary that has access to it can predict the future outputs of the generator. Bellare

and Yee [BY03] proposed a dedicated security model to assess *Forward Security*: it should be infeasible to recover any information on previous states or previous output blocks from the compromise of the current state. In addition, several works studied the extension of the [BY03] model in the presence of **leakage**. In this paper, we present two constructions that extend the original construction of [BY03], satisfying different notions of leakage and reaching different security levels. All these models and associated constructions are described in Section 3.

A second solution to amortize the use of a random seed is to allow the algorithm to continuously collect new inputs in addition to the seed and produce outputs that depend on the previous inputs. This class of algorithm is referred to as a **pseudo-random number generator with input**. In this situation, the idea is to use the largest amount of possible events from the environment of the generator, gather them together in the internal state  $S$  of the generator and produce outputs that are indistinguishable from random. An expected property of the generator is that it *accumulates* the successive inputs properly, so that each new input is taken into account. The compromise of the internal state is still critical in this situation, however, as new inputs are collected continuously, the generator may *recover* from a compromise if enough inputs are collected. Moreover, as inputs may be influenced by an adversary, a second expected property is that the generator *preserves* its state against such inputs.

The formalization of the expected security properties of a pseudo-random number generator with input has been a challenging task. We present in Section 4 the successive models for pseudo-random number generators with inputs that have been proposed. A major contribution of these security model is the formalization of both these *recovering* and *preserving* properties.

Eventually, in Section 5, we present recent extensions of the robustness model that have been proposed to capture different classes of adversaries. First we present an extension related to *premature next attacks*, applicable in situations in which the state of the pseudo-random number generator with input has not accumulated a sufficient amount of entropy and is asked to produce some outputs. Then we present an extension related to *memory attacks*, which refers to situations where an adversary can recover or modify a significant fraction of the secret stored in memory, even if those secrets have never been involved in any computation, contrary to the class of attacks that rely on computation. Finally we present an extension related to *leakage security*, which contrary to memory attacks leads to leakage of sensitive information because measurements can be made during generator operations.

## 1.2 PRNG Models

Gutmann [Gut98], and Kelsey *et al.* [KSWH98] gave useful guidelines for the design of secure pseudo-random number generators with input. In these guidelines, they all considered a generator as a pair of algorithms, one to collect inputs and a second one to generate outputs. Desai, Hevia and Yin [DHY02], modelled secure pseudo-random number generators with input as a pair of algorithms: the *Seed Generation* algorithm and the *Output Generation* algorithm. This model assumes the existence of an entropy pool, different from the internal state, in which randomness is accumulated, that is used to refresh the internal state of the generator. Viega [Vie03] analyzed the use of AES in counter mode to build a secure generator and the issue of entropy accumulation. Barak, Shaltiel and Tromer [BST03] proposed a security model where an adversary can have some control on the randomness source. This model explicitly explains the importance of a *randomness extractor* as a core component of a generator and proposes an analysis of the public parameter *seed* which is inherent to this component. An elegant and remarkable work by Barak and Halevi [BH05] modelled pseudo-random number generators with input as a pair of algorithms (*refresh*, *next*) and defined a new security property called *robustness*

based on the design guidelines of [KSWH98]: this property assesses the behavior of a generator after the compromise of its internal state, but fails to capture the small and gradual entropy accumulation present in most real-life implementations. In [DPR<sup>+</sup>13], Dodis *et al.* extended the work of [BH05] and formalized the accumulation process of a pseudo-random number generator with input. They introduced the notion of a controlled *Distribution Sampler*, that allows an adversary to control the distribution of the inputs that are collected by a generator and a new property of *entropy accumulation*. Three extensions of the previous model have been then proposed. In [DSSW14], Dodis *et al.* extended the model to capture the premature next attack, in which the generator has accumulated an insufficient amount of entropy and is asked to produce some outputs. In [CR14], Cornejo and Ruhault extended the model to capture memory attacks and in [ABP<sup>+</sup>15], Abdalla *et al.* extended the model to capture leakage.

### 1.3 Potential Weaknesses

The lack of assurance about the generated random numbers can cause serious damages in cryptographic protocols, and vulnerabilities can be exploited by adversaries to mount concrete attacks. One striking example is the failure in the Debian Linux distribution [CVE], where a commented code in the OpenSSL generator led to insufficient entropy gathering and allowed an adversary to conduct brute force guessing attacks against cryptographic keys.

Concerning system generators, an analysis of Linux generators `dev/random` and `dev/urandom` was done in 2006 by Gutterman, Pinkas and Reinman in [GPR06], where they presented an attack for which a fix has been published. Lenstra *et al.* [LHA<sup>+</sup>12] showed that a non-negligible percentage of RSA keys share prime factors. Heninger *et al.* [HDWH12] presented an analysis of the behavior of Linux generators that explains the generation of low entropy keys when these keys are generated at boot time and the findings of Lenstra *et al.* The Windows pseudo-random number generator with input `CryptGenRandom` was analyzed in 2006 by Dorrendorf, Gutterman and Pinkas in [DGP07]; the authors showed an attack on the forward security of the generator implemented in Windows 2000, for which a fix has been published. In [DPR<sup>+</sup>13], Dodis *et al.* gave a precise assessment of the security of the two Linux pseudo-random number generators with input: `/dev/random` and `/dev/urandom`. In particular, they showed several attacks proving that these generators are not robust because they do not accumulate entropy properly. These attacks are due to the vulnerabilities of the entropy estimator and the internal mixing function of the generators.

Concerning application generators, Argyros and Kiayias [AK12] showed practical attacks on web applications exploiting randomness vulnerabilities in PHP applications. Michaelis, Meyer and Schwenk [MMS13] described and analyzed several Java implementations; they have also identified some weaknesses. More recently, a flaw in the Android pseudo-random number generator, identified by Kim, Han and Lee in [KHL13], has been actively exploited against Android-based Bitcoin wallets [SEC]. In [CR14], Cornejo and Ruhault gave an in-depth analysis of generator implementations from widely used providers in real-life applications: OpenSSL, OpenJDK, Android, Bouncycastle and IBM. Their analysis revealed new vulnerabilities of these generators due to the implementation of their internal state in several fields that are not updated securely. In [ST15], Shrimpton and Terashima gave a complete analysis of the Intel Secure Key hardware generator ISK-RNG, which has been included in Intel processors since late 2011. Their analysis provides concrete security bounds for the *forward security* and the *backward security* of the two implemented instructions `RDRAND` and `RDSEED`. Finally, in [GT16], Gazi and Tessaro proposed an extension of the original sponge-based generator of [BDPV10] in the model of [DPR<sup>+</sup>13].

Concerning standards, the previous version of the NIST specification [BK15] contained

a pseudo-random number generator named `Dual_EC_DRBG` that has been known to admit a serious potential back door in the event that an attacker generates the standard algorithm parameters, a potential vulnerability that has been announced by Shumow and Ferguson at the Crypto rump session in 2007. In [CFN<sup>+</sup>14], Checkoway *et al.* mounted practical attacks against TLS/SSL connections established by software libraries implementing this generator using the back door, assuming that an attacker knows a trapdoor for the parameters. In December 2015, Juniper Networks announced that unknown attackers had added unauthorized code to ScreenOS, the operating system for their NetScreen VPN routers, related to an implementation of `Dual_EC_DRBG` (see [CMG<sup>+</sup>16] for details). Recent works study the use of potentially backdoored generators ([DGG<sup>+</sup>15, DPSW16]).

This illustrates the need for precise evaluation of pseudo-random number generators based on clear security requirements. Several notions and associated security models exist, which we detail in this paper.

## 1.4 Formalization

Let  $X$  be a random variable over a sample set  $S$ . Then  $X$  defines a probability distribution  $P_X : S \rightarrow [0, 1]$ , where  $P_X(x) := \Pr[X = x]$  called the **distribution** of the random variable  $X$ . The random variable is also called a **source** on  $S$ . We denote by  $X$  both the random variable  $X$  and the distribution of the random variable  $X$ . Let  $n > 0$  be an integer, the **uniform distribution** over the sample set  $\{0, 1\}^n$  is denoted  $\mathcal{U}_n$ . We denote  $X \stackrel{\$}{\leftarrow} \{0, 1\}^n$  when  $X$  is uniformly distributed over  $\{0, 1\}^n$ .

Let  $X$  and  $Y$  be two random variables. Then  $X$  and  $Y$  are **independent** if for all  $x$  and  $y$ ,  $\Pr[(X = x) \text{ and } (Y = y)] = \Pr[X = x] \cdot \Pr[Y = y]$ . Let  $n > 0$  be an integer and let  $X$  and  $Y$  be two random variables over the sample set  $\{0, 1\}^n$ . The **statistical distance** between  $X$  and  $Y$  is equal to:  $\mathbf{SD}(X, Y) = \frac{1}{2} \sum_x |\Pr[X = x] - \Pr[Y = x]|$ . The random variables  $X$  and  $Y$  are said  $\varepsilon$ -**close** if  $\mathbf{SD}(X, Y) \leq \varepsilon$ .

Let  $X$  be a source on  $S$ . The **min-entropy** of  $X$  is  $\mathbf{H}_\infty(X) = \min_{x \in S} \{-\log \Pr[X = x]\}$ . A source  $X$  is a  $k$ -**source** if  $\mathbf{H}_\infty(X) \geq k$ . A  $k$ -source of length  $n$  has an **entropy rate** equal to  $k/n$ . Let  $Z$  be a source on  $S$ . The **worst-case min-entropy of  $X$  conditioned on  $Z$**  is  $\mathbf{H}_\infty(X|Z) = -\log([\max_{x,z} \Pr[X = x|Z = z]])$ .

Let  $p$  and  $n$  be integers, such that  $p > n$ . A **hash function** is a function  $h : \{0, 1\}^p \rightarrow \{0, 1\}^n$ . A hash functions family  $\mathcal{H} = \{h_X : \{0, 1\}^p \rightarrow \{0, 1\}^n, X \in \{0, 1\}^s\}$  is  $\varepsilon$ -**universal** if for any inputs  $x_1 \neq x_2 \in \{0, 1\}^p$  we have:  $\Pr_{X \stackrel{\$}{\leftarrow} \{0, 1\}^s} [h_X(x_1) = h_X(x_2)] \leq \varepsilon$ . Let  $H = \{h_X : I \rightarrow [X \cdot I]_n\}$ , where all operations are in  $F_{2^p}$  and  $[y]_n$  denotes the first  $n$  bits of  $y$ . Then  $H$  is  $1/2^n$ -universal [BST03]. Let  $\bar{I} := (I_{d-1}, \dots, I_0)$  be the concatenation of  $d$  samples in  $\{0, 1\}^p$  and let  $H' = \{h'_X : \bar{I} \rightarrow \sum_{j=0}^{d-1} I_j \cdot X^j\}$ , where all operations are in  $F_{2^p}$ . Then  $H'$  is  $(d/2^n)$ -universal [DPR<sup>+</sup>13].

In this paper, we describe security models in the code-based game playing framework of [BR06]. In this framework, a security game involves a challenger and an adversary, denoted  $\mathcal{A}$ . The challenge of the adversary is to distinguish between two experiments, which are both indexed by a Boolean bit  $b$ . Interactions between the challenger and the adversary are modeled with *procedures*. A security game **GAME** has an *initialize* procedure, procedures to respond to adversary oracle queries, and a *finalize* procedure. A security game **GAME** is executed with an adversary  $\mathcal{A}$  as follows. First, the challenger executes procedure *initialize*, and its outputs are given as inputs to  $\mathcal{A}$ . Then  $\mathcal{A}$  executes, its oracle queries being answered by the corresponding procedures of **GAME**. When  $\mathcal{A}$  terminates, its output becomes the input to the *finalize* procedure. The output of the *finalize* procedure is called the output of the security game **GAME**, and we denote the output of the adversary as  $\text{GAME}^{\mathcal{A}}$ . Finally we denote the event that this output takes value  $y$  as  $\text{GAME}^{\mathcal{A}} \Rightarrow y$  and we define the *advantage* of  $\mathcal{A}$  in **GAME** as  $\text{Adv}_{\mathcal{A}}^{\text{GAME}} = 2 \times \Pr[\text{GAME}^{\mathcal{A}} \Rightarrow 1] - 1$ .

## 1.5 Accumulators and Extractors

Randomness is concretely generated from sources which are potentially biased, where the only known information is that they may contain some amount of randomness, or they are  $k$ -sources. We therefore need two family of maps: (a) on the one hand, maps that given a set of sources, accumulates the randomness that these sources collectively contain; these maps are named **accumulators** and (b) on the second hand, maps that extracts the randomness that is actually contained in these sources, and produce an output which is close to uniform. These maps are named **extractors**.

Extractors and accumulators are special cases of a more general notion, named **condensers**. Informally, a  $(k, k')$ -condenser is a function that given a  $k$ -source as input, outputs a  $k'$ -source: when the output is close to uniform, it is an **extractor** and when  $k' = k$ , it is an **accumulator**. The design of the generators ISAAC in [Jen96] and Fortuna in [FSK10] uses the notion of accumulator, however without a clear formalization. In addition, an accumulator is implicitly contained in a large family of generators, including Linux `dev/random`, Windows `CryptGenRandom`, and the OpenSSL generator. Definition 1 is adapted from [DRV12] and [Che09]. In this paper, we focus on situations for which independence between the public parameter `seed` and the randomness source is guaranteed and where extraction and accumulation are possible for all  $k$ -sources and for all adversaries (usually called **strong extractors** and **strong accumulators**). In Section 6, we discuss this choice, as it has practical impacts: in particular, we show that the generator described in [BK15], named `CTR_DRBG`, and proposed as a standard by the NIST, can be broken because of potential correlation between one public parameter of the specification and the randomness source.

**Definition 1** ([DRV12, Che09]). A function  $\text{Cond} : \{0, 1\}^p \times \{0, 1\}^s \rightarrow \{0, 1\}^n$  is a  $(k, k', \varepsilon)$ -condenser if for all probabilistic adversaries  $\mathcal{A}$  which sample a distribution  $X$  of entropy  $\mathbf{H}_\infty(X) \geq k$ , the distributions  $(\text{seed}, \text{Cond}(X, \text{seed}))$  is  $\varepsilon$ -close to some  $(\text{seed}, R)$ , where  $\mathbf{H}_\infty(R|\text{seed}) \geq k'$ ,  $\text{seed} \stackrel{\$}{\leftarrow} \{0, 1\}^s$  and  $X$  is independent of `seed`. If  $k' = k$ , it is called a  $(k, \varepsilon)$ -accumulator and if  $k' = n$ , it is called a  $(k, \varepsilon)$ -extractor.

Definition 1 can be expressed in terms of a family of universal hash functions family (Sect. 1.4). The hash function family  $H$  is a  $(k, \varepsilon)$ -extractor if for any random variable  $I$  over  $\{0, 1\}^p$  with  $\mathbf{H}_\infty(I) \geq k$ , the distributions  $(X, h_X(I))$  and  $(X, \mathcal{U}_n)$  are  $\varepsilon$ -close for all  $X$ , or a  $(k, \varepsilon)$ -accumulator if for any random variable  $I$  over  $\{0, 1\}^p$  with  $\mathbf{H}_\infty(I) \geq k$ , the distribution  $h_X(I)$  is  $\varepsilon$ -close to some  $R$ , where  $\mathbf{H}_\infty(R|X) \geq k$ .

The Leftover Hash Lemma<sup>1</sup> constructs extractors and accumulators from universal hash functions families. This lemma was first formally stated in [HILL99]. Note that the usual version of Lemma 1 presents the extraction; here we give a more general version that extends to accumulation, adapted from [DRV12] and [Che09].

**Lemma 1** ([HILL99, DRV12, Che09]). Assume that the hash function family  $H = \{h : \{0, 1\}^p \rightarrow \{0, 1\}^n\}$  is  $\rho$ -universal where  $\rho = (1 + \alpha)2^{-n}$  for some  $\alpha \geq 0$ . Then, for any  $k, k' \geq 0$ , it is also a  $(k, k', \varepsilon)$ -condenser, where  $\varepsilon = \frac{1}{2}\sqrt{2^{k'-k} + 2^{k'-n}(1 + \alpha)} - 1$ . In particular, for  $k' = k$  and  $k < n$ , it is a  $(k, \varepsilon)$ -accumulator, where  $\varepsilon = \frac{1}{2}\sqrt{2^{k-n}(1 + \alpha)}$  and for  $k' = n$  and  $k > n$ , it is a  $(k, \varepsilon)$ -extractor, where  $\varepsilon = \frac{1}{2}\sqrt{2^{n-k} + \alpha}$ .

Let  $H = \{h_X : I \rightarrow [X \cdot I]_n\}$ , where all operations are in  $F_{2^p}$  and  $[y]_n$  denotes the first  $n$  bits of  $y$ . Then  $H$  is  $1/2^n$ -universal and a  $(k, \varepsilon)$ -extractor, where  $\varepsilon = \frac{1}{2}\sqrt{2^{n-k}}$  [BST03]. Let  $\bar{I} := (I_{d-1}, \dots, I_0)$  be the concatenation of  $d$  samples in  $\{0, 1\}^p$  and let  $H' = \{h'_X : \bar{I} \rightarrow \sum_{j=0}^{d-1} I_j \cdot X^j\}$ , where all operations are in  $F_{2^p}$ . Then  $H'$  is  $(d/2^n)$ -universal and a  $(k, \varepsilon)$ -accumulator, where  $\varepsilon = \frac{1}{2} \cdot \sqrt{d2^{k-n}}$  [DGMP92, dB93, DPR<sup>+</sup>13].

<sup>1</sup>For completeness, we give the complete proof in Appendix B.

## 2 Standard Pseudo-Random Number Generators

Blum and Micali [BM82] and Yao [Yao82] defined security for a standard pseudo-random number generator. Based on this seminal work, Bellare and Yee [BY03] formalized security.

**Definition 2** ([BM82, Yao82, BY03]). Let  $s$  and  $\ell$  be integers such that  $\ell > s$ . A  $(s, \ell)$ -standard pseudo-random number generator is a function  $\mathbf{G} : \{0, 1\}^s \rightarrow \{0, 1\}^\ell$ , that takes as input a bit string  $S$  of length  $s$  and outputs a bit string  $R$  of length  $\ell$ .

Consider the security game PR described in Fig. 1. In this security game, the challenger generates a random secret input  $K$  and challenges the adversary  $\mathcal{A}$  on its capacity to distinguish the output of the pseudo-random number generator from random.

proc. initialize	proc. next-ror	proc. finalize( $b^*$ )
$S \xleftarrow{\$} \{0, 1\}^n;$	$R_0 \leftarrow \mathbf{G}(S)$	IF $b = b^*$ RETURN 1
$b \xleftarrow{\$} \{0, 1\};$	$R_1 \xleftarrow{\$} \{0, 1\}^\ell$	ELSE RETURN 0
	RETURN $R_b$	

**Figure 1:** Procedures in Security Game PR

**Definition 3** ([BM82, Yao82, BY03]). Let  $n$  and  $\ell$  be integers such that  $\ell > n$ . A  $(n, \ell)$ -standard pseudo-random number generator is  $(t, \varepsilon)$ -secure if for any adversary  $\mathcal{A}$  running in time at most  $t$ , the advantage of  $\mathcal{A}$  in game PR is at most  $\varepsilon$ .

Recall that a block cipher is a function  $E : \{0, 1\}^n \times \{0, 1\}^\ell \rightarrow \{0, 1\}^\ell$  such that for each  $S \in \{0, 1\}^n$  and  $M \in \{0, 1\}^\ell$ , the function  $E_S : \{0, 1\}^\ell \rightarrow \{0, 1\}^\ell$  defined by  $E_S(M) = E(S, M)$  is a permutation on  $\{0, 1\}^\ell$ . In this paper, we will propose constructions based on the block cipher AES [AES01] and prove the security of these constructions by reduction to the security of the following pseudo-random number generator. Let  $S \xleftarrow{\$} \{0, 1\}^n$ , where  $n \in \{128, 256\}$  and  $q \geq 1$ . Consider the  $(n, qn)$ -pseudo-random number generator  $\mathbf{G}$  defined by  $\mathbf{G}(S) = \text{AES}_S(1) \parallel \dots \parallel \text{AES}_S(q)$ . Then, following the ‘PRF/PRP Switching Lemma’<sup>2</sup> of Bellare *et al.* [BDJR97], we have that  $\mathbf{G}$  is a  $(t, \frac{q(q-1)}{2^{n+1}})$ -secure pseudo-random generator, under the assumption that the block cipher AES is indistinguishable from a pseudo-random permutation.

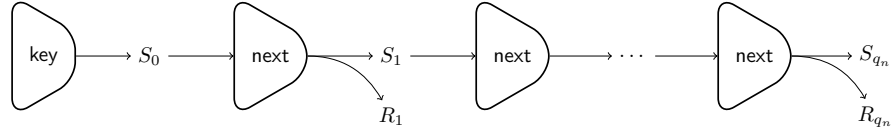
Note that in this paper, we present constructions in different security models that all rely on the use on the block cipher AES. The proposed constructions could be adapted to another block cipher, however as security bounds are obtained by reduction to the security of AES as a secure standard pseudo-random number generator, these security bounds shall be adapted. We chose the block cipher AES because it is widely implemented and provides hardware and software efficiency.

## 3 Stateful Generators

Bellare and Yee [BY03] proposed a notion of *stateful pseudo-random number generators* where the maximal number of outputs the pseudo-random number generator is allowed to produce (named  $q_n$  hereafter) is a parameter of the generator. This notion is illustrated in Fig. 2 and formalized in Definition 4.

**Definition 4** ([BY03]). A *stateful pseudo-random number generator* is a pair of algorithms (key, next) and an integer  $q_n$ , where key is a probabilistic algorithm which takes no input and outputs an initial state  $S \in \{0, 1\}^n$ , next is a deterministic algorithm which, given the current state  $S$ , outputs a pair  $(S', R) \leftarrow \text{next}(S)$  where  $S'$  is the new state and

<sup>2</sup>For completeness, this lemma is presented in Appendix A.



**Figure 2:** Stateful Pseudo-Random Number Generator [BY03]

$R \in \{0, 1\}^\ell$  is the output and  $q_n$  is the maximal number of outputs the pseudo-random number generator is allowed to produce.

Consider the security game SPR described in Fig. 3. In this security game, the challenger generates a random initial secret  $S$  and challenges the adversary  $\mathcal{A}$  on its capacity to distinguish the real output of the pseudo-random number generator from random. The difference with game PR is that here successive calls to next-ror will produce different outputs, that should all be indistinguishable from random.

proc. initialize	proc. next-ror	proc. finalize( $b^*$ )
$S \xleftarrow{\$} \text{key};$	$(S, R_0) \leftarrow \text{next}(S)$	IF $b = b^*$ RETURN 1
$b \xleftarrow{\$} \{0, 1\}$	$R_1 \xleftarrow{\$} \{0, 1\}^\ell$	ELSE RETURN 0
	OUTPUT $R_b$	

**Figure 3:** Procedures in Security Game SPR

**Definition 5** ([Pie09, JP14]). A stateful pseudo-random number generator  $\mathbf{G} = (\text{key}, \text{next}, q_n)$  is called  $(t, q_n, \varepsilon)$ -secure, if for any adversary  $\mathcal{A}$  running in time at most  $t$ , making  $q_n$  calls to next-ror, the advantage of  $\mathcal{A}$  in game SPR is at most  $\varepsilon$ .

In [BY03] Bellare and Yee proposed an extension of the previous model, where a stateful pseudo-random number generator should be designed so that it is infeasible to recover any information on previous states or previous outputs from the compromise of the current state. To formalize this property, they proposed a dedicated security model where an adversary  $\mathcal{A}$  chooses dynamically when to compromise the current state  $S$ . After this compromise, all *future* outputs are compromised, as they all deterministically depend on the compromised state, however, the expected security property (named **Forward Security**) is that the *past* outputs are computationally indistinguishable from random.

proc. initialize	proc. get-state	proc. next-ror	proc. finalize( $b^*$ )
$S \xleftarrow{\$} \text{key};$	OUTPUT $S$	$(S, R_0) \leftarrow \text{next}(S)$	IF $b = b^*$ RETURN 1
$b \xleftarrow{\$} \{0, 1\}$		$R_1 \xleftarrow{\$} \{0, 1\}^\ell$	ELSE RETURN 0
		OUTPUT $R_b$	

**Figure 4:** Procedures in Security Game FWD

Consider the security game FWD described in Fig. 4. In this security game, the challenger generates a random initial secret input  $S$  and challenges the adversary  $\mathcal{A}$  on its capacity to distinguish the real output of the pseudo-random number generator from random. In addition to the usual procedures, the adversary  $\mathcal{A}$  has access to a procedure get-state in which  $\mathcal{A}$  has access to the current value of the internal state  $S$ .

**Definition 6** ([BY03]). A stateful pseudo-random number generator  $\mathbf{G} = (\text{key}, \text{next}, q_n)$  is called  $(t, q_n, \varepsilon)$ -forward-secure, if for any adversary  $\mathcal{A}$  running in time at most  $t$ , making at most  $q_n$  calls to next-ror, followed by one call to get-state, which is the last call  $\mathcal{A}$  is allowed to make, the advantage of  $\mathcal{A}$  in game FWD is at most  $\varepsilon$ .

key	next
<b>Require:</b> $\emptyset$	<b>Require:</b> $S$
<b>Ensure:</b> $S$	<b>Ensure:</b> $S', R$
1: $S \xleftarrow{\$} \{0,1\}^{128}$	1: $S' = \text{AES}_S(1)$
2: <b>return</b> $S$	2: $R = \text{AES}_S(2)$
	3: <b>return</b> $(S', R)$

**Figure 5:** FWD: A Secure Construction

Consider the stateful pseudo-random number generator  $\mathcal{G}$ , defined with algorithms (key, next) described in Fig. 5. It uses the  $(t, 2^{-128})$ -secure standard generator  $\mathbf{G}$  defined by  $\mathbf{G}(S) = \text{AES}_S(1) \parallel \text{AES}_S(2)$ . Following [BY03], we have that  $\mathcal{G}$  is a  $(t, 2^{64}, 2^{-128})$ -forward secure stateful pseudo-random number generator.

An important assumption for the security of stateful pseudo-random number generators is the secrecy of the state. However, in practice there are situations where the state can be compromised. For example, side-channel attacks exploit the fact that every cryptographic algorithm is ultimately implemented on a physical device and an implementation enable observations which can be made and measured, such as the amount of power consumption or the time taken. To capture this, **leakage security** of a stateful pseudo-random number generator (key, next) has been defined. The main idea is to model the leakage of information with a *leakage function*, that is named  $f$ . It is important to note that without restrictions on the leakage function, no security can be guaranteed (one simple attack would be to leak the complete state or the next output of the generator). Yet a fundamental issue in the context of leakage-resilient cryptography is to define reasonable restrictions on the leakage functions. The following assumptions are considered:

- **(1) Only computation leaks:** only the data being manipulated in a computation can leak during this computation [MR04].
- **(2) Bounded leakage per iteration:** the length of the output of the leakage functions, expressed in bits and taken globally, is bounded with a parameter  $\lambda$  [Pie09, YSPY10, FPS12, YS13].
- **(3) Non-Adaptive Leakage:** leakage functions are not adaptively chosen by the adversary before each invocation, but are fixed [YSPY10, FPS12, YS13].
- **(4) Simulatable leakage:** if an adversary cannot tell the difference between the real leakage function and a simulated leakage function (from a simulator that does not know the secret state of the generator) then the real leakage function does not reveal any information about the state [SPY13]. Note that as pointed out in [SPY13], this assumption implies that the leakage function is unbounded, as every new measurement gives more information.

To illustrate this, we present below two constructions, that are extensions of the previous forward-secure construction from Fig. 5. The first construction from [YSPY10] satisfies assumptions 1, 2 and 3 and has a low security level, while the second construction from [LMO<sup>+</sup>14] satisfies in addition assumption 4 and has a high security level.

Consider the security game  $\text{LPR}(f)$  described in Fig. 6. The objective of the adversary  $\mathcal{A}$  is to distinguish the output of the generator at one round from a uniformly distributed random value, given the successive outputs and leakages for the previous rounds. Formally, the security game extends SPR (Fig. 3), with the additional procedure leaknext.

**Definition 7.** Let  $f$  be a leakage function of output length  $\lambda$ . A stateful pseudo-random number generator  $\mathbf{G} = (\text{key}, \text{next})$  is  $(t, q_n, \varepsilon, f)$ -leakage resilient for the leakage function  $f$  if for any attacker  $\mathcal{A}$  running in time at most  $t$ , making  $q_n$  calls to next-ror/ leaknext, the advantage of  $\mathcal{A}$  in game  $\text{LPR}(f)$  is at most  $\varepsilon$ .



<b>proc. initialize</b> $S \xleftarrow{\$} \text{key};$ $b \xleftarrow{\$} \{0, 1\}$	<b>proc. leaknext</b> $\left\{ \begin{array}{l} L \leftarrow f(S) \\ (S, R) \leftarrow \text{next}(S) \end{array} \right\}$ OUTPUT $(L, R)$	<b>proc. next-ror</b> $(S, R_0) \leftarrow \text{next}(S)$ $R_1 \xleftarrow{\$} \{0, 1\}^\ell$ RETURN $R_b$	<b>proc. finalize(<math>b^*</math>)</b> IF $b = b^*$ RETURN 1 ELSE RETURN 0
--	---	--	---

**Figure 6:** Procedures in Security Game  $\text{LPR}(f)$

<b>key</b> <b>Require:</b> $\emptyset$ <b>Ensure:</b> $S_0, P_0, P_1$ 1: $S_0 \xleftarrow{\$} \{0, 1\}^{128}$ 2: $P_0, P_1 \xleftarrow{\$} \{0, 1\}^{256}$ 3: <b>return</b> $S_0, P_0, P_1$	<b>next</b> <b>Require:</b> $S_i$ <b>Ensure:</b> $S_{i+1}, R_{i+1}$ 1: $S_{i+1} = \text{AES}_{S_i}(P_{\rho(i)}^\ell)$ 2: $R_{i+1} = \text{AES}_{S_i}(P_{\rho(i)}^r)$ 3: <b>return</b> $(S_{i+1}, R_{i+1})$
--	---

**Figure 7:**  $\text{LPR}(f)$ : Construction from [YSPY10]

Consider the stateful pseudo-random number generator  $\mathcal{G}$  described in Fig. 7. Algorithm **key** outputs a secret  $S_0 \in \{0, 1\}^{128}$ , which is kept secret, and two public values  $P_0 \in \{0, 1\}^{256}$  and  $P_1 \in \{0, 1\}^{256}$ . The two public values are used in an alternative way to update the internal state and to produce output, we denote  $\rho(i) = i \bmod 2$  to indicate which public value is used. We also decompose  $P_0 \in \{0, 1\}^{256}$  (resp.  $P_1 \in \{0, 1\}^{256}$ ) in two 128-bits blocks, denoted  $P_0^\ell, P_0^r$  (resp.  $P_1^\ell, P_1^r$ ). The generator uses the  $(t, 2^{-128})$ -secure standard generator  $\mathbf{G}$  defined by  $\mathbf{G}(S_i) = \text{AES}_{S_i}(P_{\rho(i)}^\ell || \text{AES}_{S_i}(P_{\rho(i)}^\ell))$ , seen as a  $(t, 2, 2^{-128})$ -weak pseudo-random function<sup>3</sup>. Note that each AES operation is done either on the leftmost part or on the rightmost part of each public value (hence the notation  $P_0^\ell$  or  $P_0^r$ ). In [YSPY10], Yu *et al.* proved that under assumptions (1) (2) and (3) above, the security of  $\mathcal{G}$  is measured by  $\varepsilon = q_n \varepsilon_{\mathbf{G}}^{1/12}$  and  $\lambda \leq \log(\varepsilon_{\mathbf{G}}^{-1/6})$ , which depend on the security of  $\mathbf{G}$  as a  $(t, 2, 2^{-128})$ -weak pseudo-random function ( $\varepsilon_{\mathbf{G}}$ ) and on the maximal number of calls to **next** ( $q_n$ ). Hence  $\mathcal{G}$  is a  $(t, 4, 2^{-8}, f)$ -leakage resilient stateful pseudo-random number generator, where  $\lambda = 21$ .

<b>key</b> <b>Require:</b> $\emptyset$ <b>Ensure:</b> $S_0, P_0, P_1, P_2$ 1: $S_0 \xleftarrow{\$} \{0, 1\}^{128}$ 2: $P_0, P_1, P_2 \xleftarrow{\$} \{0, 1\}^{384}$ 3: <b>return</b> $S_0, P_0, P_1, P_2$	<b>next</b> <b>Require:</b> $S_i$ <b>Ensure:</b> $S_{i+1}, R_{i+1}$ 1: $S_{i+1} = \text{AES}_{S_i}(P_{\rho(i)}^\ell)$ 2: $\alpha_i = \text{AES}_{S_i}(P_{\rho(i)}^m)$ 3: $R_{i+1} = \text{AES}_{\alpha_i}(P_{\rho(i)}^r)$ 4: <b>return</b> $(S_{i+1}, R_{i+1})$
---	---

**Figure 8:**  $\text{LPR}(f)$ : Construction from [LMO<sup>+</sup>14]

In [SPY13] Standaert, Pereira and Yu considered the security of the generator described in Fig. 7 in the context of *simulatable* leakage (assumption (4) above). They obtained tighter bounds for this construction, namely, they proved that the security degrades linearly with the number of rounds. However, in [LMO<sup>+</sup>14], Longo Galea *et al.* exhibited a distinguisher against the simulator of Standaert *et al.* and propose to enforce the above construction with an extra call to AES and an extra public value. Their construction is described in Fig. 8. Algorithm **key** outputs a secret  $S_0 \in \{0, 1\}^{128}$ , and three public values  $P_0 \in \{0, 1\}^{384}$ ,  $P_1 \in \{0, 1\}^{384}$  and  $P_2 \in \{0, 1\}^{384}$ . The three public values are used in an alternative way to update the internal state and to produce output, we denote  $\rho(i) = i \bmod 3$  to indicate which public value is used. As before, we decompose  $P_0 \in \{0, 1\}^{384}$  (resp.  $P_1 \in \{0, 1\}^{384}$ ,  $P_2 \in \{0, 1\}^{384}$ ) into three 128-bits blocks, denoted  $P_0^\ell, P_0^m, P_0^r$  (resp.  $P_1^\ell, P_1^m, P_1^r$  and  $P_2^\ell, P_2^m, P_2^r$ ). As before, each AES operation is done on the leftmost part or on the middle

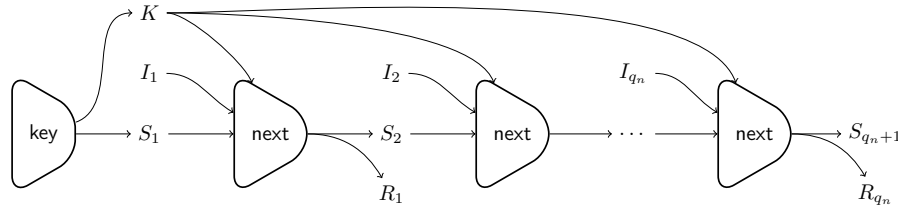
<sup>3</sup>For completeness, the definition of a weak pseudo-random function is given in Appendix A.

part or on the rightmost part of each public value (hence the notation  $P_0^\ell$ ,  $P_0^m$  or  $P_0^r$ ). In [LMO<sup>+</sup>14], Longo Galea *et al.* proved that this construction is a  $(t, 2^{64}, 2^{-63}, f)$ -leakage resilient stateful pseudo-random number generator, where  $f$  is simulatable and unbounded.

## 4 Generators with Input

### 4.1 Model of Desai, Hevia and Yin

Desai, Hevia and Yin [DHY02] proposed a security model for pseudo-random number generators with input where the internal state is split into two parts: a first part named  $K$  (the *key*) and second part named  $S$  (the *state*). In their model, a pseudo-random number generator with input is a stateful and iterative algorithm, which at each invocation produces some output bits as a function of the current value of  $K$  and  $S$  and an *auxiliary* input  $I$  and which updates the state  $S$ . They proposed several security properties, which capture the potential compromise of the state  $S$ , the key  $K$  or the auxiliary input  $I$ . The generator operations are illustrated in Fig. 9, in accordance with Definition 8.



**Figure 9:** Pseudo-Random Number Generator with Input [DHY02]

**Definition 8** ([DHY02]). A *pseudo-random number generator with input* is a pair of algorithms ( $\text{key}, \text{next}$ ), where  $\text{key}$  is a probabilistic algorithm that takes no input and outputs a *key*  $K \in \{0, 1\}^n$  and an initial state  $S \in \{0, 1\}^n$ ,  $\text{next}$  is a deterministic algorithm that, given the current state  $S$ , the key  $K$  and an auxiliary input  $I \in \{0, 1\}^p$ , outputs a pair  $(S', R) \leftarrow \text{next}(S, K, I)$ , where  $S' \in \{0, 1\}^n$  is the new state and  $R \in \{0, 1\}^\ell$  is the output.

They denoted their attacks as CIA, for Chosen-Input Attack, CSA, for Chosen-State Attack and KKA, for Known-Key Attack. Under CIA, the key is hidden, the states are known, but not chosen, and the auxiliary input may be chosen by the adversary. The attack CSA is similar, except that the auxiliary inputs are not allowed to be chosen while the states may now be chosen. The attack KKA is different: it allows the key to be known. However, under the attack KKA, the states are hidden and the auxiliary inputs are not allowed to be chosen. All security games are described in Fig. 10.

<b>proc. initialize</b> $(K, S) \stackrel{\$}{\leftarrow} \text{key};$ $(I_1, \dots, I_{q_n}) \stackrel{\$}{\leftarrow} (\{0, 1\}^p)^{q_n};$ $i \leftarrow 1;$ $b \stackrel{\$}{\leftarrow} \{0, 1\};$	<b>proc. getinput</b> OUTPUT $I_i$	<b>proc. get-state</b> OUTPUT $S$	<b>proc. next-ror</b> $(S, R_0) \leftarrow \text{next}(S, K, I_i)$ $R_1 \stackrel{\$}{\leftarrow} \{0, 1\}^\ell$
<b>proc. finalize</b> ( $b^*$ ) IF $b = b^*$ RETURN 1 ELSE RETURN 0	<b>proc. setinput</b> ( $I^*$ ) $I_i \leftarrow I^*$	<b>proc. set-state</b> ( $S^*$ ) $S \leftarrow S^*$	$i \leftarrow i + 1$ OUTPUT $R_b$
	<b>proc. get-key</b> OUTPUT $K$		

**Figure 10:** Procedures in Security Games CIA, CSA and KKA

**Definition 9** ([DHY02]). A pseudo-random number generator with input  $(\text{key}, \text{next})$  is called  $(t, q_n, \varepsilon)$ -secure against Chosen Input Attack (resp. Chosen State Attack or Known Key Attack), if for any adversary  $\mathcal{A}$  running in time at most  $t$ , which generates at most  $q_n$  outputs, the advantage of  $\mathcal{A}$  in game CIA, (resp. CSA, KKA) is at most  $\varepsilon$ , where  $\mathcal{A}$  can call `getinput` and `next-ror`, in CIA,  $\mathcal{A}$  cannot call `get-key` or `set-state` and can call `get-state` and `setinput`, in CSA,  $\mathcal{A}$  cannot call `get-key` or `setinput` and can call `get-state` and `set-state` and in KKA,  $\mathcal{A}$  cannot call `get-state`, `set-state` or `setinput` and can call `get-key`.

setup	key	next
<b>Require:</b> $\emptyset$	<b>Require:</b> $\emptyset$	<b>Require:</b> $X, K, S, I$
<b>Ensure:</b> $X$	<b>Ensure:</b> $K, S$	<b>Ensure:</b> $S', R$
1: $X \xleftarrow{\$} \{0, 1\}^{128}$	1: $K \xleftarrow{\$} \{0, 1\}^{128}$	1: $U = K \cdot X^2 + S \cdot X + I$
2: <b>return</b> $X$	2: $S \xleftarrow{\$} \{0, 1\}^{128}$	2: $S' = \text{AES}_U(1)$
	3: <b>return</b> $K, S$	3: $R = \text{AES}_U(2)$
		4: <b>return</b> $(S', R)$

**Figure 11:** CIA, CSA, KKA: A Secure Construction

In [DHY02], Desai *et al.* proposed constructions secure against CSA, CIA and KKA, which are based on existing standard specifications (ANSI X9.17 [ANS85] and FIPS [DSS00]). However, to prove security, they rely on the capability of the adversary to ‘cause collisions in the inputs to the functions computing the outputs or the next states’. However, the independence between successive inputs can not be completely guaranteed and we are not confident with the bounds presented in these constructions. Therefore we present a new construction, based on an accumulator (Def. 1), secure against CSA, CIA and KKA. Consider the pseudo-random number generator  $\mathcal{G} = (\text{setup}, \text{key}, \text{next})$  defined in Fig. 11 where the input, key, state lengths are equal (128 bits). The generator requires that in addition to the key and next algorithms, a third algorithm named `setup` is defined, to generate a random public parameter  $X \in \{0, 1\}^{128}$ , which is the seed of an accumulator (Def. 1). Note that this algorithm is not required in Definition 8, but since the generator uses an accumulator, it is necessary to complete the description. The generator uses the function  $(K, S, I) \rightarrow K \cdot X^2 + S \cdot X + I$  as a  $(128, 2^{-128})$ -accumulator of seed  $X$  and the  $(t, 2^{-128})$ -secure standard generator  $\mathbf{G}$  defined by  $\mathbf{G}(U) = \text{AES}_U(1) \parallel \text{AES}_U(2)$ . The security of  $\mathcal{G}$  is measured by  $\varepsilon = q_n(\varepsilon_{\mathbf{G}} + \varepsilon_H)$ , where  $\varepsilon_{\mathbf{G}}$  measures the security of  $\mathbf{G}$  and  $\varepsilon_H$  measures the security of the accumulator. Hence with  $q_n = 2^{64}$ ,  $\mathcal{G}$  is  $(t, 2^{64}, 2^{-63})$  secure against CIA, CSA and KKA.

## 4.2 Model of Barak, Shaltiel and Tromer

Barak, Shaltiel and Tromer [BST03] modeled the following scenario: a manufacturer designs a device whose output is supposed to be a randomness source. Ideally, one would like the adversary not to be able to influence the distribution of the randomness source at all. However, in a realistic setting an adversary can have some control over the environment in which the device operates (temperature, voltage, frequency, timing, etc.), and it is possible that changes in this environment affect the source. In their model, they assumed that the adversary can define a set of  $k$ -sources that will provide the inputs. This model also considers that the generator definition shall describe the underlying randomness extractor, as formalized in Definition 1 and therefore an algorithm `setup` is used, to provide the public parameter `seed` associated with the extractor.

**Definition 10** ([BST03]). A *pseudo-random number generator with input* is a pair of algorithms  $\mathcal{G} = (\text{setup}, \text{next})$ , where `setup` is a probabilistic algorithm that outputs a public parameter `seed`  $\in \{0, 1\}^s$  and `next` is a deterministic algorithm that, given `seed` and an input  $I \in \{0, 1\}^p$ , outputs  $R \leftarrow \text{next}(\text{seed}, I) \in \{0, 1\}^\ell$ .

Consider the security game described in Fig. 12. The game is parametrized by a family of distributions  $\mathcal{F}$ , that is adversarially provided. During procedure `initialize`, the challenger parses  $\mathcal{F}$  as  $\{\mathcal{D}_j, j \in J\}$  and calls `setup` to generate `seed`, which is given to  $\mathcal{A}$ . During procedure `next-ror`,  $\mathcal{A}$  chooses a distribution  $\mathcal{D}_j \in \mathcal{F}$ , the challenger samples an input  $I$  of distribution  $\mathcal{D}_j$ , generates the real output ( $R_0 = \text{next}(\text{seed}, I)$ ), picks a random string ( $R_1$ ) and returns the challenge  $R_b$  to  $\mathcal{A}$ . Definition 11 formalizes security.

proc. initialize	proc. next-ror( $j$ )	proc. finalize( $b^*$ )
seed $\stackrel{\$}{\leftarrow}$ setup;	$I \stackrel{\$}{\leftarrow} \mathcal{D}_j$	IF $b = b^*$ RETURN 1
parse $\mathcal{F}$ as $\{\mathcal{D}_j, j \in J\}$	$R_0 \leftarrow \text{next}(\text{seed}, I)$	ELSE RETURN 0
$b \stackrel{\$}{\leftarrow} \{0, 1\}$ ;	$R_1 \stackrel{\$}{\leftarrow} \{0, 1\}^n$	
OUTPUT seed	RETURN $R_b$	

**Figure 12:** Procedures in Security Game  $\text{RES}(\mathcal{F})$

**Definition 11** ([BST03]). A pseudo-random number generator with input  $\mathcal{G} : (\text{setup}, \text{next})$  is  $(t, q_n, \varepsilon)$ -resilient for the family  $\mathcal{F}$  if for any adversary  $\mathcal{A}$  running in time  $t$ , which generates at most  $q_n$  outputs, the advantage of  $\mathcal{A}$  in game  $\text{RES}(\mathcal{F})$  is at most  $\varepsilon$ .

setup	next
<b>Require:</b> $r$	<b>Require:</b> $X, I$
<b>Ensure:</b> $X$	<b>Ensure:</b> $R$
1: $X \stackrel{\$}{\leftarrow} \{0, 1\}^{\frac{512}{r}}$	1: $R = [X \cdot I]_{128}$
2: return $X$	2: return $R$

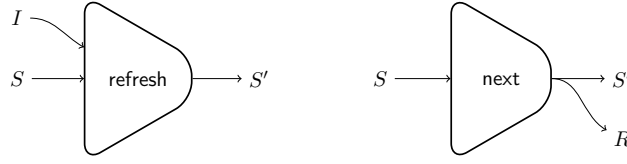
**Figure 13:**  $\text{RES}(\mathcal{F})$ : A Secure Construction

Consider the pseudo-random number generator with input  $\mathcal{G}$  defined in Fig. 13, where  $r$  is the entropy rate of the entropy source. It uses the  $2^{-128}$ -universal hash function family  $\mathcal{F} : \{h_X : I \rightarrow [X \cdot I]_{128}\}$  as a  $(k, \varepsilon_H)$ -extractor, where  $\varepsilon_H = \frac{1}{2}\sqrt{2^{128-k}}$ . In [BST03] Barak *et al.* demonstrated that the security of  $\mathcal{G}$  is measured by  $\varepsilon = q_n \varepsilon_H$ . Hence with  $q_n = 2^{64}$  and a source containing at least 512 bits of entropy we obtain that  $\mathcal{G}$  is  $(t, 2^{-64})$ -resilient for the family  $\mathcal{F}$ . As explained in [BST03], the length of the public parameter  $X$  shall be estimated depending on the environment and the entropy rate: for a low entropy rate (e.g. equal to  $\frac{1}{4}$ ), we need to set  $p = s = 2048$ , while for a high entropy rate (e.g. equal to  $\frac{3}{4}$ ), we can set  $p = s = 683$ . Hence the drawback of this model is that one has to identify an entropy source and estimate its entropy rate.

### 4.3 Robustness

Barak and Halevi [BH05] proposed a security model for pseudo-random number generators with input that clearly states that the *entropy extraction* process and the *output generation* process are completely different in nature, where entropy extraction is information-theoretic and generation is cryptographic. As a consequence, these two operations should be separated and analyzed independently. The generator operations are illustrated in Fig. 14, in accordance with Definition 12.

**Definition 12** ([BH05]). A *pseudo-random number generator with input* is a pair of algorithms (`refresh`, `next`) where `refresh` is a deterministic algorithm that, given the current state  $S \in \{0, 1\}^n$  and an input  $I \in \{0, 1\}^p$ , outputs a new state  $S' \leftarrow \text{refresh}(S, I)$  where  $S' \in \{0, 1\}^n$  is the new state and `next` is a deterministic algorithm that, given the current state  $S$ , outputs a pair  $(S', R) \leftarrow \text{next}(S)$  where  $S' \in \{0, 1\}^n$  is the new state and  $R \in \{0, 1\}^\ell$  is the output of the generator.



**Figure 14:** Pseudo-Random Number Generator with Input [BH05]

In their security model (described in Fig. 15), they captured the potential compromise of the internal state  $S$  and of the inputs used to refresh the internal state. They considered an adversary  $\mathcal{A}$  that has access to the system where the generator is run, and can (a) get the output of the generator, (b) modify the data that is used to refresh the internal state of the generator and (c) have access to and modify the internal state of the generator. The adversary  $\mathcal{A}$  has always two choices to refresh the generator, either with an input with high entropy, or with an input that  $\mathcal{A}$  totally controls. In the first case,  $\mathcal{A}$  uses procedure **good-refresh**:  $\mathcal{A}$  chooses the distribution from the family  $\mathcal{F}$ , and generates an input of the chosen distribution and finally the challenger applies algorithm **refresh** with the previously generated input. In the second case,  $\mathcal{A}$  uses procedure **bad-refresh**:  $\mathcal{A}$  chooses an input that is directly used with algorithm **refresh**. The security model uses a new important Boolean parameter, named **corrupt**, which is set to **true** when the generator is compromised and set to **false** otherwise. This parameter is part of the security game and is not a component of the generator. Note that the **next-ror** procedure differs from the equivalent procedure in the previous security models. Here, as the challenger maintains the flag **corrupt**, a challenge between the real output and a random one is sent to  $\mathcal{A}$  only if **corrupt** = **false**. If **corrupt** = **true**, the adversary can mount an attack on the real output, so  $\mathcal{A}$  will certainly distinguish it from a random one. Definition 13 formalizes security.

<b>proc. initialize</b> seed $\stackrel{\$}{\leftarrow}$ setup; $S \leftarrow 0^n$ ; corrupt $\leftarrow$ true; $b \stackrel{\$}{\leftarrow} \{0, 1\}$ ; parse $\mathcal{F}$ as $\{\mathcal{D}_j, j \in J\}$ OUTPUT seed  <b>proc. finalize</b> ( $b^*$ ) IF $b = b^*$ RETURN 1 ELSE RETURN 0	<b>proc. good-refresh</b> ( $j$ ) $I \stackrel{\$}{\leftarrow} \mathcal{D}_j$ ; $S \leftarrow$ refresh( $S, I$ ); corrupt $\leftarrow$ false;  <b>proc. bad-refresh</b> ( $I$ ) IF corrupt = true $S \leftarrow$ refresh( $S, I$ ); ELSE $\perp$	<b>proc. set-state</b> ( $S^*$ ) corrupt $\leftarrow$ true $S \leftarrow S^*$ OUTPUT $R_0$	<b>proc. next-ror</b> $(S, R_0) \leftarrow$ next( $S$ ) IF corrupt = true, RETURN $R_0$ ELSE $R_1 \stackrel{\$}{\leftarrow} \{0, 1\}^\ell$ OUTPUT $R_b$
---	--	---	---

**Figure 15:** Procedures in Security Game  $\text{ROB}(\mathcal{F})$

**Definition 13** ([BH05]). A pseudo-random number generator with input  $\mathcal{G} : (\text{refresh}, \text{next})$  is  $(t, q_n, q_r, \varepsilon)$ -robust for the family  $\mathcal{F}$  if for any adversary running in time  $t$ , making at most  $q_r$  calls to  $\mathcal{D}$ -refresh and  $q_n$  calls to next-ror, the advantage of  $\mathcal{A}$  in game  $\text{ROB}(\mathcal{F})$  is at most  $\varepsilon$ .

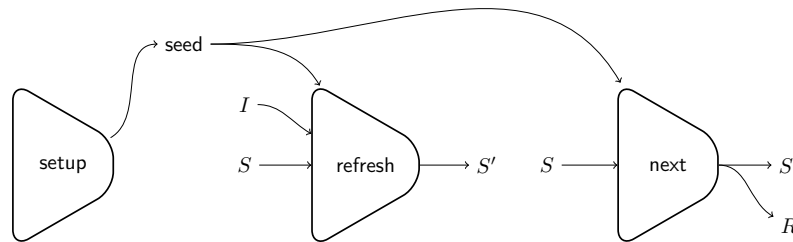
<b>setup</b> <b>Require:</b> $r$ <b>Ensure:</b> $X$ 1: $X \stackrel{\$}{\leftarrow} \{0, 1\}^{\frac{512}{r}}$ 2: return $X$	<b>refresh</b> <b>Require:</b> $X, I, S$ <b>Ensure:</b> $S'$ 1: $U = [X \cdot I]_{128}$ 2: $S' = S \oplus U$ 3: return $S'$	<b>next</b> <b>Require:</b> $S$ <b>Ensure:</b> $S', R$ 1: $S' = \text{AES}_S(1)$ 2: $R = \text{AES}_S(2)$ 3: return $(S', R)$
---	--	--

**Figure 16:**  $\text{ROB}(\mathcal{F})$ : A Secure Construction

Consider the pseudo-random number generator with input  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  defined in Fig. 16. The generator requires that in addition to the refresh and next algorithms, a third algorithm **setup** is defined, to generate a random public parameter  $X$  that is the seed of an extractor. Note that this parameter is not mandatory in Definition 12, but as we will explain in Section 4.4, this parameter is necessary to provide a complete description of the generator. We denote  $r$  the entropy rate of the entropy source. It uses the  $2^{-128}$ -universal hash function family  $\{h_X : I \rightarrow [X \cdot I]_{128}\}$  as a  $(k, \varepsilon_H)$ -extractor, where  $\varepsilon_H = \frac{1}{2}\sqrt{2^{128-k}}$  and the  $(t, 2^{-128})$ -secure standard generator  $\mathbf{G}$  defined by  $\mathbf{G}(S) = \text{AES}_S(1) \parallel \text{AES}_S(2)$ . In [BH05] Barak and Halevi demonstrated that the security of  $\mathcal{G}$  is measured by  $\varepsilon = q_r \varepsilon_H + q_n \varepsilon_{\mathbf{G}}$ , where  $\varepsilon_{\mathbf{G}} = 2^{-128}$  is the security of  $\mathbf{G}$  and  $\varepsilon_H = 1/2\sqrt{2^{128-k}}$  is the extractor security. With powerful adversaries ( $q_r = q_n = 2^{64}$ ) and a source containing at least 512 bits of entropy,  $\mathcal{G}$  is  $(t, 2^{64}, 2^{64}, 2^{-63})$ -robust. Hence we face the same issue as for  $\text{RES}(\mathcal{F})$  regarding the entropy rate of the entropy source.

#### 4.4 Robustness, Preserving and Recovering Security

Dodis *et al.* [DPR<sup>+</sup>13] proposed a security model in which the definition of a pseudo-random number generator with input requires that, in addition to **refresh** and **next** algorithms, an algorithm named **setup** is set. This algorithm generates a public parameter **seed**, which will be used to provide a seed for an extractor and a seed for an accumulator (Definition 1). This is illustrated in Fig. 17 and formalized in Definition 14.



**Figure 17:** PRNG with Input [DPR<sup>+</sup>13]

**Definition 14** ([DPR<sup>+</sup>13]). A *pseudo-random number generator with input* is a triple of algorithms  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  where **setup** is a probabilistic algorithm that outputs a public parameter  $\text{seed} \in \{0, 1\}^s$ , **refresh** is a deterministic algorithm that, given  $\text{seed} \in \{0, 1\}^s$ , a state  $S \in \{0, 1\}^n$  and an input  $I \in \{0, 1\}^p$ , outputs a new state  $S' \leftarrow \text{refresh}(\text{seed}, S, I) \in \{0, 1\}^n$  and **next** is a deterministic algorithm that, given  $\text{seed} \in \{0, 1\}^s$  and a state  $S \in \{0, 1\}^n$ , outputs a pair  $(S', R) \leftarrow \text{next}(\text{seed}, S)$  where  $S' \in \{0, 1\}^n$  is the new state and  $R \in \{0, 1\}^\ell$  is the output.

To define security, Dodis *et al.* split the adversary into two entities: an *adversary*  $\mathcal{A}$  whose task is (intuitively) to distinguish the outputs of the generator from random, and a *distribution sampler*  $\mathcal{D}$  whose task is to produce inputs  $I_1, I_2, \dots$ , which have high entropy *collectively*, but help  $\mathcal{A}$  in breaking the security of the generator. The distribution sampler aims at modeling potentially adversarial environment where the generator operates. To ensure independence of the randomness sources with **seed**, they require that the distribution sampler is set independently of **seed** and once  $\mathcal{D}$  is set, the adversary  $\mathcal{A}$  has access to **seed**. This separation between  $\mathcal{A}$  and  $\mathcal{D}$  clarifies the requirement of independence between the adversary and **seed**: as independence is only required between **seed** and the randomness source to build a strong randomness extractor, they enforce independence between **seed**

and the ‘part’ of the adversary that has control over the randomness source and they let the ‘other part’ have access to `seed`. The above discussion justifies Definition 15.

**Definition 15** (Distribution Sampler). Let  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  be a pseudo-random number generator with input. A distribution sampler  $\mathcal{D}$  for  $\mathcal{G}$  is a *stateful and probabilistic* algorithm which, given the current state  $\sigma$ , outputs a tuple  $(\sigma', I, \gamma, z)$  where:  $\sigma'$  is the new state for  $\mathcal{D}$ ,  $I \in \{0, 1\}^p$  is the next input for the refresh algorithm,  $\gamma$  is some *fresh entropy estimation* of  $I$ ,  $z$  is the *leakage* about  $I$  given to the adversary  $\mathcal{A}$ . Let denote by  $q_r$  the upper bound on number of executions of  $\mathcal{D}$  in the security games. Then  $\mathcal{D}$  is *legitimate* if  $\mathbf{H}_\infty(I_k \mid I_1, \dots, I_{k-1}, I_{k+1}, \dots, I_{q_r}, z_1, \dots, z_{q_r}, \gamma_1, \dots, \gamma_{q_r}) \geq \gamma_k$  for all  $k \in \{1, \dots, q_r\}$  where  $(\sigma_k, I_k, \gamma_k, z_k) = \mathcal{D}(\sigma_{k-1})$  for  $k \in \{1, \dots, q_r\}$  and  $\sigma_0 = 0$ .

proc. initialize( $\mathcal{D}$ )	proc. $\mathcal{D}$ -refresh	proc. get-state	proc. next-ror
$\text{seed} \stackrel{\$}{\leftarrow} \text{setup};$	$(\sigma, I, \gamma, z) \stackrel{\$}{\leftarrow} \mathcal{D}(\sigma)$	$c \leftarrow 0$	$(S, R_0) \leftarrow \text{next}(\text{seed}, S)$
$\sigma \leftarrow 0;$	$S \leftarrow \text{refresh}(\text{seed}, S, I)$	OUTPUT $S$	$R_1 \stackrel{\$}{\leftarrow} \{0, 1\}^\ell$
$S \leftarrow 0^n;$	$c \leftarrow c + \gamma$		IF $c < \gamma^*$ THEN $c \leftarrow 0$
$c \leftarrow 0;$	IF $c < \gamma^*$		ELSE OUTPUT $R_b$
$b \stackrel{\$}{\leftarrow} \{0, 1\};$	$c \leftarrow \min(c + \gamma, n)$	proc. set-state( $S^*$ )	
OUTPUT $\text{seed}$	OUTPUT $(\gamma, z)$	$c \leftarrow 0$	
		$S \leftarrow S^*$	
proc. finalize( $b^*$ )			
IF $b = b^*$ RETURN 1			
ELSE RETURN 0			

**Figure 18:** Procedures in Security Game  $\text{ROB}(\gamma^*)$

Dodis *et al.* [DPR<sup>+</sup>13] formalized three security properties: *recovering security*, *preserving security* and *robustness*. Informally, this new robustness property captures that entropy can be accumulated at a low pace in the internal state of the generator to reach a complete recovery (contrary to the previous robustness model where a full recovery is done with a high entropy input). To model this property, they decomposed it into two simpler properties: recovering security ensures that entropy is properly accumulated and preserving security ensures that the accumulated entropy does not decrease. The fundamental result of [DPR<sup>+</sup>13] is that taken together, recovering and preserving security imply robustness<sup>4</sup>.

The robustness security game uses procedures described in Fig. 18. The game is parametrized by  $\gamma^*$ , that measures the minimal entropy that the pseudo-random number generator shall have accumulated when security is expected. It uses  $\mathcal{D}$ -refresh procedure where the distribution sampler  $\mathcal{D}$  is run, and its output  $I$  is used to refresh the current state  $S$ , the amount of fresh entropy  $\gamma$  is added to the entropy counter  $c$  and the values of  $\gamma$  and the leakage  $z$  are also returned to  $\mathcal{A}$ ; next-ror procedure, where it provides  $\mathcal{A}$  with either the real-or-random challenge (provided that  $c \geq \gamma^*$ ) or the true generator output, a ‘premature’ call before  $c$  crosses the  $\gamma^*$  resets the counter  $c$  to 0, since then  $\mathcal{A}$  might learn something about the (low-entropy) state  $S$  in this case; get-state/set-state procedures, that provide  $\mathcal{A}$  with the ability to either learn the current state  $S$ , or set it to any value  $S^*$  and in either case  $c$  is reset to 0. Definition 16 formalizes security.

**Definition 16** ([DPR<sup>+</sup>13]). A pseudo-random number generator with input  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  is called  $(t, q_r, q_n, q_s, \gamma^*, \varepsilon)$ -robust, if for any adversary  $\mathcal{A}$  running in time at most  $t$ , making at most  $q_r$  calls to  $\mathcal{D}$ -refresh,  $q_n$  calls to next-ror/get-next and  $q_s$  calls to get-state/set-state, and any legitimate distribution sampler  $\mathcal{D}$  inside the  $\mathcal{D}$ -refresh procedure, the advantage of  $\mathcal{A}$  in game  $\text{ROB}(\gamma^*)$  is at most  $\varepsilon$ .

<sup>4</sup>For completeness, this result is fully described in Appendix C.

setup	refresh	next
<b>Require:</b> $\emptyset$	<b>Require:</b> $X, I, S$	<b>Require:</b> $S, X'$
<b>Ensure:</b> $X, X'$	<b>Ensure:</b> $S'$	<b>Ensure:</b> $S', R$
1: $X \xleftarrow{\$} \{0, 1\}^{1024}$	1: $S' = S \cdot X + I$	1: $U = [X' \cdot S]_{256}$
2: $X' \xleftarrow{\$} \{0, 1\}^{1024}$	2: <b>return</b> $S'$	2: $S' = \text{AES}_U(1)    \dots    \text{AES}_U(8)$
3: <b>return</b> $X, X'$		3: $R = \text{AES}_U(9)$
		4: <b>return</b> $(S', R)$

**Figure 19:** ROB: A Secure Construction

Let  $q > 0$ ,  $m > 0$ ,  $n > 0$  and  $\gamma^* > 0$ . Consider:

- The  $(t, \varepsilon_{\mathbf{G}})$ -secure pseudo-random generator  $\mathbf{G}$  defined by  $\mathbf{G}(U) = \text{AES}_U(1) || \dots || \text{AES}_U(q)$ , where  $U \in \{0, 1\}^m$  and  $\varepsilon_{\mathbf{G}} = \frac{q(q-1)}{2^{m+1}}$ .
- The  $(m, \varepsilon_{H'})$ -accumulator, where  $\varepsilon_{H'} = \frac{1}{2} \cdot \sqrt{q_r \cdot 2^{m-n}}$  defined by the  $(q_r/2^n)$ -universal hash function family  $\{h'_X : \bar{I} \rightarrow \sum_{j=0}^{q_r-1} I_j \cdot X^j\}$ .
- The  $(\gamma^*, \varepsilon_H)$ -extractor, where  $\varepsilon_H = \frac{1}{2} \sqrt{2^{m-\gamma^*}}$ , defined by the  $2^{-n}$ -universal hash function family  $\{h_X : I \rightarrow [X \cdot I]_n\}$ .
- The pseudo-random number generator  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  defined in Fig. 19.

The security of  $\mathcal{G}$  is obtained by [DPR<sup>+</sup>13], where Dodis *et al.* demonstrated that it is  $((t', q_r, q_n, q_s), \gamma^*, \varepsilon)$ -robust, where  $\varepsilon = q_n(2\varepsilon_{\mathbf{G}} + q_r^2(\varepsilon_H + \varepsilon_{H'}) + 2^{-n+1})$ . Taking into account powerful adversaries ( $q_n = q_r = q_s = 2^{64}$ ) we obtain  $m = 256$ ,  $n = 1024$ ,  $\gamma^* = 900$  and  $q = 9$ . Hence  $\mathcal{G}$  is  $((t, 2^{64}, 2^{64}, 2^{64}), 900, 2^{-128})$ -robust:  $\mathcal{G}$  offers 128 bits security provided 900 bits have been accumulated in its internal state, of length 1024 bits. Note that we considered an adversary with high capabilities ( $2^{64}$ ), therefore the obtained bounds are higher than the one from [DPR<sup>+</sup>13].

## 5 Extensions of the Robustness Model

### 5.1 Security Against Premature Next

In [DSSW14], Dodis *et al.* extended the robustness model to address the *premature next attack*. This general attack, first explicitly mentioned in [KSWH98] is applicable in situations in which the state of the pseudo-random number generator with input has not accumulated a sufficient amount of entropy and is asked to produce some outputs  $R$  via legitimate next calls. Inspired by the design of the Fortuna pseudo-random number generator [FSK10], they partition the incoming entropy into multiple entropy pools and then use these pools at vastly different rates when producing outputs, in order to guarantee that at least one pool will eventually accumulate enough entropy to guarantee security before it is emptied via a premature next call. They complement the construction presented in Fig. 19 with a *secure scheduler*, whose task is to fill the different pools of entropy to ensure that at least one pool contains enough entropy to defeat this attack.

The model extension is described in Fig. 20. In this model extension, a premature next call is *not* considered an unrecoverable state corruption, and premature calls do *not* reset the entropy counter. The price for this is a new parameter named  $\beta \geq 1$ . In particular, in the modified game, the game does not immediately declare the state to be uncorrupted when the entropy counter  $c$  goes above the threshold  $\gamma^*$ . Instead, the game keeps a new counter  $T$  that records the number of calls previously done to  $\mathcal{D}$ -refresh since the last *set-state* or *get-state* call (or since the start of the game). When  $c$  goes beyond  $\gamma^*$ , it sets  $T^* \leftarrow T$  and the state becomes uncorrupted only after  $T \geq \beta \cdot T^*$  (of course, provided  $\mathcal{A}$  made no additional calls to *set-state* or *get-state*). In [DSSW14], Dodis *et al.* introduced a



<b>proc.</b> initialize( $\mathcal{D}$ )	<b>proc.</b> $\mathcal{D}$ -refresh	<b>proc.</b> get-state	<b>proc.</b> next-ror
seed $\stackrel{\$}{\leftarrow}$ setup;	$(\sigma, I, \gamma, z) \stackrel{\$}{\leftarrow} \mathcal{D}(\sigma)$	$c \leftarrow 0, \text{corrupt} \leftarrow \text{true}; (S, R_0) \leftarrow \text{next}(\text{seed}, S)$	
$\sigma \leftarrow 0;$	$S \leftarrow \text{refresh}(\text{seed}, S, I)$	$T \leftarrow 0, T^* \leftarrow 0$	$R_1 \stackrel{\$}{\leftarrow} \{0, 1\}^\ell$
$S \leftarrow 0^n;$	$c \leftarrow c + \gamma$	OUTPUT S	IF $c < \gamma^*$ ,
$c \leftarrow 0;$	$T \leftarrow T + 1$		RETURN $R_0$
$b \stackrel{\$}{\leftarrow} \{0, 1\};$	IF $c \geq \gamma^*$		ELSE
corrupt $\leftarrow \text{true}$	IF $T^* = 0$		OUTPUT $R_b$
OUTPUT seed	$T^* \leftarrow T$	<b>proc.</b> set-state( $S^*$ )	
	IF $T \geq \beta \cdot T^*$	$c \leftarrow 0, \text{corrupt} \leftarrow \text{true}$	
<b>proc.</b> finalize( $b^*$ )	corrupt $\leftarrow \text{false}$	$T \leftarrow 0, T^* \leftarrow 0$	
IF $b = b^*$ RETURN 1	OUTPUT $(\gamma, z)$	$S \leftarrow S^*$	
ELSE RETURN 0			

**Figure 20:** Procedures in Security Game NROB( $\gamma^*, \beta$ )

new parameter named  $w_{max}$ , that is an upper bound on the collected entropy for each input. In our description, we omit this parameter, because as mentioned in [DSSW14], it is mainly useful for the security proof.

**Definition 17** (Security of PRNG with input against Premature Next [DSSW14]). A pseudo-random number generator with input  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  is called  $(T = (t, q_r, q_n, q_s), \gamma^*, \varepsilon, \beta)$ -premature next robust, if for any attacker  $\mathcal{A}$  running in time at most  $t$ , making at most  $q_r$  calls to  $\mathcal{D}$ -refresh,  $q_n$  calls to next-ror/get-next and  $q_s$  calls to get-state/set-state, and any legitimate distribution sampler  $\mathcal{D}$  inside the  $\mathcal{D}$ -refresh procedure, the advantage of  $\mathcal{A}$  in game NROB( $\gamma^*, \beta$ ) is at most  $\varepsilon$

$SC$
<b>Require:</b> $skey, \tau$
<b>Ensure:</b> $\tau', in, out$
1: IF $\tau \neq 0 \bmod 512$ , THEN $out \leftarrow \perp$
2: ELSE $out \leftarrow \max\{out : \tau = 0 \bmod 2^{\tau+9}\}$
3: $in \leftarrow \text{int}([\text{AES}_{skey}(\tau)]_5)$
4: $\tau' \leftarrow \tau + 1 \bmod 2^{32}$
5: OUTPUT $(\tau', in, out)$

**Figure 21:** NROB: A Secure Scheduler

In [DSSW14], Dodis *et al.* proposed a construction inspired by the design of the Fortuna pseudo-random number generator [FSK10]. The idea is to use a pool of 32 robust pseudo-random number generators, two of whom are chosen pseudo-randomly, one to collect entropy and the other one to generate outputs. The choice of the two generators is done through the function  $SC$  (for *secure scheduler*), which uses the block cipher AES as a pseudo-random function and *public* parameters  $skey$  and  $\tau$ . For the exact formalization of a scheduler, we refer to [DSSW14]; in this paper, we focus on the instantiation with common primitives. The secure scheduler  $SC$  is described in Fig. 21. We denote  $\text{int}([X]_5)$  the function that converts the five leftmost bits of  $X$  to an integer in  $\{0, \dots, 31\}$ .

Let  $i$  be an integer. Consider the pseudo-random number generator family  $\mathcal{G}_i = (\text{setup}_i, \text{refresh}_i, \text{next}_i)$  described in Fig. 22 (top). Each generator is exactly the same as the one described in Fig. 19, except that the output of  $\mathcal{G}_i$  has length 256 bits (instead of 128 bits for the construction of Fig. 19), all generators have the same **setup** function, denoted  $\text{setup}_{\mathcal{G}}$ . Consider finally the pseudo-random number generator  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  described in Fig. 22 (bottom). It uses 32 different generators  $\mathcal{G}_i, i = 1, \dots, 32$ , the scheduler  $SC$ , has output length equal to 256 bits. Its internal state is equal to the concatenation of the 32 internal states of the generators  $\mathcal{G}_i (S_i)$ , each of length 1024 bits, a block  $S_\rho$  of length 256 bits, an integer  $\tau$  of length 32 bits, two integers  $in, out \in \{0 \dots 31\}$ . Parameters  $X, X'$  and  $skey$  are public. In [DSSW14], Dodis *et al.* proved that the pseudo-random

setup $\mathcal{G}$	refresh $_i$	next $_i$
<b>Require:</b> $\emptyset$	<b>Require:</b> $X, I, S_i$	<b>Require:</b> $S_i, X'$
<b>Ensure:</b> $X, X'$	<b>Ensure:</b> $S'$	<b>Ensure:</b> $S'_i, R$
1: $X \xleftarrow{\$} \{0, 1\}^{1024}$	1: $S'_i = S_i \cdot X + I$	1: $U = [X' \cdot S_i]_{256}$
2: $X' \xleftarrow{\$} \{0, 1\}^{1024}$	2: <b>return</b> $S'_i$	2: $S'_i = \text{AES}_U(0) \parallel \dots \parallel \text{AES}_U(7)$
3: <b>return</b> $X, X'$		3: $R = \text{AES}_U(8) \parallel \text{AES}_U(9)$
		4: <b>return</b> $(S'_i, R)$

setup	refresh	next
<b>Require:</b> $\emptyset$	<b>Require:</b> $X, I, S$	<b>Require:</b> $S$
<b>Ensure:</b> $X, X', skey$	<b>Ensure:</b> $S'$	<b>Ensure:</b> $S', R$
1: $X, X' \leftarrow \text{setup}_{\mathcal{G}}$	1: parse seed as $(key, seed)$	1: parse $S$ as $(\tau, S_\rho, (S_i)_{i=0}^{31})$
2: $skey \xleftarrow{\$} \{0, 1\}^{128}$	2: parse $S$ as $(\tau, S_\rho, (S_i)_{i=0}^{31})$	2: $S_\rho = \text{AES}_{S_\tau}(1) \parallel \text{AES}_{S_\rho}(2)$
3: <b>return</b> $X, X', skey$	3: $(\tau, in, out) \leftarrow SC(key, \tau)$	3: $R = \text{AES}_{S_\rho}(3) \parallel \text{AES}_{S_\rho}(4)$
	4: $S_{in} \leftarrow \text{refresh}_{in}(X, S_{in}, I)$	4: <b>return</b> $(S', R)$
	5: $(S_{out}, R) \leftarrow \text{next}_{out}(X', S_{out})$	
	6: $S_\rho \leftarrow S_\rho \oplus R$	
	7: <b>return</b> $S' = (\tau, S_\rho, (S_i)_{i=0}^{31})$	

**Figure 22:** NROB: A Secure Construction  $(\mathcal{G}_i, \mathcal{G})$

number generator with input  $\mathcal{G}$  defined above is  $((t, 2^{32}, 2^{32}, 2^{32}), 900, 2^{-128}, 4)$ -premature next robust.

## 5.2 Security against Memory Attacks

In [CR14], Cornejo and Ruhault proposed a modification of the *robustness* security model to identify exactly the part of  $S$  that an adversary needs to compromise to attack a pseudo-random number generator with input. To capture this idea, they considered the internal state as a concatenation of several binary strings (named hereafter its *decomposition*). They modelled the adversarial capability of an adversary  $\mathcal{A}$  with two new functions named  $\mathcal{M}$ -get and  $\mathcal{M}$ -set that allow  $\mathcal{A}$  to set or get a part of the internal state of the pseudo-random generator with input defined with a *mask*  $\mathcal{M}$ . They assumed that the adversary  $\mathcal{A}$  knows the *decomposition* of  $S$  and is able to choose  $\mathcal{M}$  adaptively. The only differences between their security game and the original game ROB is that they replaced the procedures get-state and set-state, with new procedures  $\mathcal{M}$ -get-state and  $\mathcal{M}$ -set-state, allowing the adversary to get/set a part the internal state identified by the mask.

<b>proc.</b> initialize( $\mathcal{D}$ )	<b>proc.</b> $\mathcal{D}$ -refresh	<b>proc.</b> $\mathcal{M}$ -set-state( $S, \mathcal{M}, J$ )	<b>proc.</b> next-ror
seed $\xleftarrow{\$}$ setup;	$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$	$S \leftarrow \mathcal{M}\text{-set}(S, \mathcal{M}, J)$	$(S, R_0) \leftarrow \text{next}(S)$
$\sigma \leftarrow 0$ ;	$S \leftarrow \text{refresh}(S, I)$	$c \leftarrow \max(0, c - \lambda)$	IF $c < \gamma^*$ ,
$S \xleftarrow{\$} \{0, 1\}^n$ ;	IF $c < \gamma^*$	IF $c < \gamma^*$ ,	$c \leftarrow 0$
$c \leftarrow n$ ;	$c \leftarrow \min(c + \gamma, n)$	$c \leftarrow 0$	OUTPUT $R_0$
corrupt $\leftarrow$ true;	OUTPUT $(\gamma, z)$	<b>proc.</b> $\mathcal{M}$ -get-state( $S, J$ )	ELSE
$b \xleftarrow{\$} \{0, 1\}$ ;		$c \leftarrow \max(0, c - \lambda)$	$R_1 \xleftarrow{\$} \{0, 1\}^\ell$
OUTPUT seed		IF $c < \gamma^*$ ,	OUTPUT $R_b$
		$c \leftarrow 0$	
<b>proc.</b> finalize( $b^*$ )		OUTPUT $\mathcal{M}\text{-get}(S, J)$	
IF $b = b^*$ RETURN 1			
ELSE RETURN 0			

**Figure 23:** Procedures in Security Game MROB( $\gamma^*, \lambda$ )

**Definition 18** (Decomposition). A decomposition of a binary string  $S \in \{0, 1\}^n$  is a sequence of disjoint binary strings  $(S_1, \dots, S_k)$ , such that  $S = [S_1 \parallel \dots \parallel S_k]$ . Two binary strings  $S$  and  $M$  have the *same* decomposition if  $M = [M_1 \parallel \dots \parallel M_k]$  and  $|S_i| = |M_i|$  for  $i \in \{1, \dots, k\}$ .

**Definition 19** ( $\mathcal{M}$ -get /  $\mathcal{M}$ -set). Function  $\mathcal{M}$ -get takes as input a couple  $(S, J)$ , where  $S = [S_1 || \dots || S_k]$  and  $J \subset \{1, \dots, k\}$ , then  $\mathcal{M}$ -get $(S, J) = (S_j)_{j \in J}$ . Function  $\mathcal{M}$ -set takes as input a triple  $(S, M, J)$ , where  $S, M \in \{0, 1\}^n$  have the same decomposition  $S = [S_1 || \dots || S_k], M = [M_1 || \dots || M_k]$  and  $J \subset \{1, \dots, k\}$ , then  $\mathcal{M}$ -set $(S, M, J) = S$ , where  $S_j = M_j$ , for  $j \in J$ .

These functions are adversarially provided, and their goal is to let  $\mathcal{A}$  choose the mask  $\mathcal{M}$  over the internal state. Note that if the mask is too large (so that  $\mathcal{G}$  becomes insecure), the security game will require that new input is collected. These procedures model the memory attacks against the generator.

The security model is adapted from the security game  $\text{ROB}(\gamma^*)$  from Sect. 4.4. The integer  $\gamma^*$  defines the minimum entropy that is required in  $S$  for the generator to be secure. Integer  $c$  estimates the amount of collected entropy. The integer  $\lambda \leq n$  defines the size of the mask  $\mathcal{M}$ . Boolean flag **corrupt** is set to **true** if  $c < \gamma^*$  and **false** otherwise. Boolean  $b$  is used to challenge the adversary  $\mathcal{A}$ . The security game uses procedures described in Fig. 23. The procedure **initialize** sets the parameter **seed** with a call to algorithm **setup**, the internal state  $S$  of the generator, as well as parameters  $c$  and  $b$ . Note that they initially set  $c$  to  $n$  and  $S$  to a random value, to avoid giving any knowledge of  $S$  to the adversary  $\mathcal{A}$ . After all oracle queries,  $\mathcal{A}$  outputs a bit  $b^*$ , given as input to the procedure **finalize**, which compares the response of  $\mathcal{A}$  to the challenge bit  $b$ . Procedure  **$\mathcal{D}$ -refresh** is the same as in the security game  $\text{ROB}(\gamma^*)$ , procedure  **$\mathcal{M}$ -set-state** is used by  $\mathcal{A}$  to set a part of  $S$ . First  $\mathcal{A}$  calls function  **$\mathcal{M}$ -set** to update a part of the internal state. Then the counter value  $c$  is decreased by  $\lambda$ , the size of the mask  $\mathcal{M}$  ( $c \leftarrow c - \lambda$ ) and as in the initial **set-state** procedure, if  $c < \gamma^*$ ,  $c$  is reset to 0. Procedure  **$\mathcal{M}$ -get-state** is used by  $\mathcal{A}$  to get a part of  $S$ . First  $\mathcal{A}$  calls the function  **$\mathcal{M}$ -get**. Then the counter value  $c$  is decreased by  $\lambda$ , the size of the mask  $\mathcal{M}$  ( $c \leftarrow c - \lambda$ ) and as in the initial **get-state** procedure, if  $c < \gamma^*$ ,  $c$  is reset to 0. Procedure **next-ror** is the same as in the security game  $\text{ROB}(\gamma^*)$ .

**Definition 20** (Security of a Pseudo-Random Number Generator with Input against Memory Attacks [CR14]). A pseudo-random number generator with input  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  is called  $(T = (t, q_r, q_n, q_s), \gamma^*, \varepsilon)$ -robust against memory attacks, if for any adversary  $\mathcal{A}$  running in time at most  $t$ , the advantage of  $\mathcal{A}$  in game  $\text{MROB}(\gamma^*, \lambda)$  is at most  $\varepsilon$ .

It is possible to construct a robust pseudo-random number generator with input (Definition 16) that never resists a single bit corruption. Consider  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  a robust pseudo-random number generator with input and denote  $S$  its internal state. Consider  $\mathcal{G}' = (\text{setup}', \text{refresh}', \text{next}')$  a second pseudo-random number generator with input. The internal state of  $\mathcal{G}'$  is defined with  $S' = S || b$  where  $b$  is a single bit. The generator  $\mathcal{G}'$  is defined with the following algorithms:

- $\text{refresh}'(S', I) = \text{refresh}(S, I) || 1$  (i.e.  $S \leftarrow \text{refresh}(S, I)$  and  $b \leftarrow 1$ )
- $\text{next}'(S') = \text{next}(S)$  if  $b = 1$ ,  $\text{next}'(S') = 0$  if  $b = 0$

Then generator  $\mathcal{G}'$  is robust since, as soon as one **refresh** procedure is executed the bit  $b$  is set to 1 and the generator  $\mathcal{G}'$  works exactly as  $\mathcal{G}$  does when the internal state is not compromised. However, it is obviously not secure under a corruption of the single bit  $b$ . Note that a similar attack can be applied against the premature next robust construction presented in Fig. 22 as a compromise of the part of the internal state named  $S_\tau$  makes the output of the generator predictable.

In [CR14], Cornejo and Ruhault proposed an extension of the construction described in Fig. 19. They proved that if the length of the internal state is increased with  $\lambda$ , then the generator is robust against memory attacks. However, note that this simple extension works because the internal state is composed of one full block that cannot be divided into

smaller blocks. An interesting work would be to extend the construction described in Fig. 22 because this construction explicitly requires a specific decomposition of the internal state between blocks.

### 5.3 Leakage Security

In [ABP<sup>+</sup>15], Abdalla *et al.* proposed a modification of the *robustness* security model to capture the potential leakage of sensitive information. In the robustness security model ROB, the distribution sampler  $\mathcal{D}$  generates the external inputs used to refresh the generator and already gives the adversary  $\mathcal{A}$  some information about how the environment of the generator leaks when it generates these inputs. This information is modelled by  $z$ . In order to model information leakage during the executions of the algorithms `refresh` and `next`, they give the adversary the choice of the leakage functions, that they globally name  $f$ , associated to each algorithm, or even each small block. Since they restrict our model to non-adaptive leakage, they ask the adversary to choose them beforehand. So the leakage functions are provided as input to the `initialize` procedure by the adversary (see Fig. 24). Then, each leakage function will be implicitly used by two new procedures named `leak-refresh` and `leaknext` that, in addition to the usual outputs, also provide some leakage  $L$  about the manipulated data.

The procedures `leak-refresh` and `leaknext` are detailed below:

- The procedure `leak-refresh` runs the `refresh` algorithm but additionally provides some information leakage  $L$  on the input  $(S, I)$  and `seed`. As for the `next-ror-queries`, the leakage can reveal information about a insecure internal state before the effectiveness of the refresh, and then  $c$  is reduced by  $\lambda$  bits. And if  $c$  drops below the threshold  $\gamma^*$ , it is reset to 0. Note that if the  $\mathcal{D}$ -refresh algorithm is complex, several leakage functions can be defined at every step, but the global leakage is limited to  $\lambda$  bits, hence the notation  $\{\dots\}$ , since they can be interleaved.
- The procedure `leaknext` runs the `next` algorithm but additionally provides some information leakage  $L$  on the input  $S$  and `seed`, according to the leakage function  $f$  provided to the `initialize` procedure. If the generator is in a secure state, then the new entropy estimate  $c$  is set to  $\alpha$ , otherwise, it is reset to 0 (as for the `next-ror`). If the `next` algorithm is complex, several leakage functions can be defined at each step, but the global leakage is limited to  $\lambda$  bits (hence the notation  $\{\dots\}$ ).

We thus have two new parameters:  $\lambda$ , that bounds the output length of the leakage function and  $\alpha$ , that models the minimal expected entropy of  $S$  after a `leaknext` (`next` with leakage) call, in a secure case, when the entropy of the internal state was assumed greater than  $\gamma^*$ .

As in the security game ROB, attackers have two parts: a distribution sampler and a classical attacker with the former only used to generate seed-independent inputs (potentially *partially* biased) from device activities. The threshold  $\gamma^*$  has to be slightly higher in this model, because for a similar `next` algorithm, we need to accumulate a bit more of entropy to maintain security even in presence of leakage. Typically, it has to be increased by  $\lambda$ . After detailing the new security game, we can define the notion of leakage-resilient robustness of a pseudo-random number generator with input.

**Definition 21** (Leakage-Resilient Robustness of Pseudo-Random Number Generator with Input [ABP<sup>+</sup>15]). A pseudo-random number generator with input  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  is called  $(t, q_r, q_n, q_s, \gamma^*, \lambda, \varepsilon)$ -leakage-resilient robust, if for any adversary  $\mathcal{A}$  running in time  $t$ , that first generates a legitimate distribution sampler  $\mathcal{D}$  (for the  $\mathcal{D}$ -refresh/leak-refresh procedure), that subsequently makes at most  $q_r$  calls to  $\mathcal{D}$ -refresh/leak-refresh,  $q_n$  calls to `next-ror/leaknext`, and  $q_s$  calls to `get-state/set-state` with a leakage bounded by  $\lambda$ , the advantage of  $\mathcal{A}$  in game  $\text{LROB}(\gamma^*, \lambda)$  is at most  $\varepsilon$ .

<b>proc. initialize</b> ( $\mathcal{D}, f$ ) $\text{seed} \stackrel{\$}{\leftarrow} \text{setup}$ $\sigma \leftarrow 0;$ $S \leftarrow 0;$ $c \leftarrow 0;$ $b \stackrel{\$}{\leftarrow} \{0, 1\}$ <b>OUTPUT</b> $\text{seed}$	<b>proc. <math>\mathcal{D}</math>-refresh</b> $(\sigma, I, \gamma, z) \stackrel{\$}{\leftarrow} \mathcal{D}(\sigma)$ $S \leftarrow \text{refresh}(S, I)$ <b>IF</b> compromised $c \leftarrow \min(c + \gamma, n)$ <b>OUTPUT</b> $(\gamma, z)$	<b>proc. get-state</b> $c \leftarrow 0;$ <b>OUTPUT</b> $S$  <b>proc. set-state</b> ( $S^*$ ) $c \leftarrow 0;$ $S \leftarrow S^*$	<b>proc. next-ror</b> $(S, R_0) \leftarrow \text{next}(S)$ <b>IF</b> $c\gamma^*$ , $c \leftarrow 0$ <b>RETURN</b> $R_0$ <b>ELSE</b> $R_1 \stackrel{\$}{\leftarrow} \{0, 1\}^\ell$ <b>RETURN</b> $R_b$
<b>proc. finalize</b> ( $b^*$ ) <b>IF</b> $b = b^*$ <b>RETURN</b> 1 <b>ELSE</b> <b>RETURN</b> 0	<b>proc. leak-refresh</b> $(\sigma, I, \gamma, z) \stackrel{\$}{\leftarrow} \mathcal{D}(\sigma)$ $\left\{ \begin{array}{l} L \leftarrow f(S, I, \text{seed}) \\ S \leftarrow \text{refresh}(S, I; \text{seed}) \end{array} \right\}$ $c \leftarrow \max\{0, c - \lambda\}$ <b>IF</b> $c\gamma^*$ $c \leftarrow 0$ <b>OUTPUT</b> $(L, \gamma, z)$	<b>proc. leaknext</b> $\left\{ \begin{array}{l} L \leftarrow f(S, \text{seed}) \\ (S, R) \leftarrow \text{next}(S; \text{seed}) \end{array} \right\}$ <b>IF</b> $c\gamma^*$ $c \leftarrow 0$ <b>ELSE</b> $c \leftarrow \alpha$ <b>OUTPUT</b> $(L, R)$	

**Figure 24:** Procedures in the Security Game  $\text{LROB}(\gamma^*, \lambda)$

Unfortunately, even a secure standard pseudo-random generator is not enough to resist information leakage. As shown in [MOP07] and later in [BGS15], several calls to AES with known inputs and one single secret key may lead to very efficient side-channel attacks that can help to recover the secret key. Because of the numerous executions of AES with the same key, one can perform a differential power analysis (DPA) attack. Then, for the construction of Fig. 19, during a `leaknext`, even with a safe state, the DPA can reveal the secret key of the internal AES, that is also used to generate the new internal state from public plaintexts. This internal state, after the `leaknext`, can thus be recovered, whereas it is considered as safe in the security game. A `next-ror` challenge can then be easily broken.

Therefore Abdalla *et al.* [ABP<sup>+</sup>15] pointed out that one needs a stronger notion of security than the usual security of a standard pseudo-random number generator for  $\mathbf{G}$ , namely a *leakage-resilient and secure standard pseudo-random number generator* (Definition 22): it takes as input a perfectly random  $m$ -bit string  $U$ , and generates an  $(n + \ell)$ -bit output  $T = (S, R)$  that looks random. Even in case of leakage,  $S$  should have enough entropy.

**Definition 22** (Leakage-Resilient and Secure Standard Pseudo-Random Number Generator [ABP<sup>+</sup>15]). A standard pseudo-random number generator  $\mathbf{G} : \{0, 1\}^m \rightarrow \{0, 1\}^N$  is  $(\alpha, \lambda)$ -leakage-resilient and  $(t, \varepsilon)$ -secure if it is first a  $(t, \varepsilon)$ -secure standard pseudo-random number generator, but in addition, for any adversary  $\mathcal{A}$ , running within time  $t$ , that first outputs a leakage  $f$  with  $\lambda$ -bit outputs, there exists a source that outputs pairs  $(L, T) \in \{0, 1\}^\lambda \times \{0, 1\}^N$ , so that the entropy of  $T$ , conditioned on  $L$  being greater than  $\alpha$ , and the advantage with which  $\mathcal{A}$  can distinguish  $(f(U), \mathbf{G}(U))$  from  $(L, T)$  is bounded by  $\varepsilon$ .

With this new property, they proposed three extensions of the construction of Fig. 19 that are leakage-resilient robust. An interesting work would be to extend the construction described in Fig. 22 because this construction explicitly requires several computations of AES that may be vulnerable to side-channel attacks.

## 6 Discussion: (In)dependence of seed

### 6.1 Impossibility Results

The robustness model from [DPR<sup>+</sup>13] has a limitation: the seed dependence of the distribution used to generate inputs. The proposed constructions crucially rely on the

independence between the distribution sampler and seed, and impossibility results show that full seed dependence is impossible.

There is a direct attack when independence between the randomness source and the seed is not guaranteed for the secure robust construction described in Fig. 19. In the secure construction, seed is composed of two parts  $(X, X')$ , where  $X, X' \in \mathbb{F}_{2^n}$ , the input  $I \in \mathbb{F}_{2^n}$  and the state  $S \in \mathbb{F}_{2^n}$ . Consider the distribution sampler  $\mathcal{D}$  where  $I_j$  is sampled uniformly from  $\{0, X^{j-q_r}\}$ . Let us consider an adversary  $\mathcal{A}$  against the security of the generator that chooses the distribution  $\mathcal{D}$ , and that makes the following oracle queries in the security game ROB: one call to `set-state(0)`,  $q_r$  calls to  `$\mathcal{D}$ -refresh`, one call to `next-ror`. After  $q_r$  calls to  `$\mathcal{D}$ -refresh`, the state of the generator is equal to:  $S = X^{q_r-1}I_1 + X^{q_r-2}I_2 + \dots + I_{q_r}$ . Then, as each term  $X^{q_r-j}I_j$  can only be equal to 0 or 1 the state  $S$  can only be equal to 0 or 1, although the inputs  $I_1, \dots, I_{q_r}$  collectively contain  $q_r$  bits of entropy. Hence the adversary  $\mathcal{A}$  breaks the robustness of the generator. One may argue that this kind of attack is only possible because this construction does not use cryptographic primitives, however, as we show in the following impossibility result, it does not suffice to build a `refresh` algorithm upon cryptographic primitives (as opposed to the polynomial hash function) to be secure against such attack. Indeed, an explicit impossibility result can also be pointed out for the generator described in [BK15], named CTR\_DRBG, and proposed as a standard by the NIST. As before, if we allow the distribution sampler to depend on seed, the adversary can mount an attack against the robustness of the generator. Here the critical point is that the parameter seed is not defined in the specification [BK15], hence an assumption shall be made on its definition. A careful analysis of the specification shows that a public parameter  $K = 0x00010203040506070809101112131415$  is defined in the specification, which is used exactly for randomness extraction. If we allow the distribution sampler  $\mathcal{D}$  to sample an input that depends on  $K$ , the adversary  $\mathcal{A}$  can mount an attack against the robustness of the generator. For completeness, as this attack is new, it is fully described in Appendix D.

## 6.2 Potential Solutions

Two promising solutions concern (a) the restriction of the capabilities of the adversary and (b) the allowance of a certain level of dependence of seed.

The first solution is to restrict the capabilities of the adversary  $\mathcal{A}$  to force its running time to be less than the running time of the extractor `Extract`. This idea was formalized by Trevisan and Vadhan in [TV00]. In this work, they show how seed-dependent randomness extraction is possible from a *sampleable* distribution, provided that the complexity of the extractor is larger than the complexity of the adversary  $\mathcal{A}$  that generates the source  $X$ . In line with this, Dodis, Ristenpart and Vadhan [DRV12] showed that sufficiently strong collision-resistant hash functions are seed-dependent condensers.

The second solution is to allow a certain level of dependence between the randomness source and seed. In [DSSW14], Dodis *et al.* introduced a realistic model that effectively allows a certain level of seed dependence. They complemented the robustness model allowing the attacker  $\mathcal{A}$  and the distribution sampler  $\mathcal{D}$  to define a new distribution sampler  $\mathcal{D}'$  correlated with seed. The idea is that the inputs that are accumulated in the internal state of the generator shall be generated by the sampler  $\mathcal{D}$ , while the inputs that are controlled by the adversary can be correlated and therefore use the second sampler  $\mathcal{D}'$ . They proved that the original construction of [DPR<sup>+</sup>13] can be extended in this model with the same parameters.

## 7 Conclusion

In this systematization of knowledge, we presented the main security models that have been formalized to define and assess the security of pseudo-random number generators. We presented three notions of generators, standard generators: stateful generators and generators with input; for each notion, we presented expected security properties. We proposed for each security model a secure and efficient construction based on AES and a polynomial hash function. We explained the link with the notions of *accumulator* and *extractor* that are used to collect entropy sources and to generate outputs. Security notions presented rely on the independence between the randomness source and the seeds of the accumulator and the extractor. To illustrate this requirement, we presented an attack against one NIST specification under the assumption that this independence is not ensured. We finally presented potential solutions where some level of dependence is accepted. We presented extensions of the robustness model, a promising work would be to extend the premature next robust construction in the context of memory attacks and leakage.

Our paper is summarized in Table 1. For each model, we provide (a) the definition (b) the attacker capabilities (c) the dependence on a standard pseudo-random number generator, a randomness extractor, a randomness accumulator or a secure scheduler and (d) the operations that are necessary in each secure construction.

## Acknowledgments

The author would like to thank the anonymous reviewers and Bart Preneel for their valuable comments and suggestions to improve the quality of the paper.

**Table 1:** Security Properties of Pseudo-Random Number Generators

Ref.	Definition	Property	Attacker Capabilities	Construction				Operations (*)	
				G	Ext	Acc	SC	refresh	next
[BY03]	1 : $S \leftarrow \text{key}$ 2 : $(S', R) \leftarrow \text{next}(S)$	FWD	next-ror, get-state	✗					AES (2)
[LMO <sup>+</sup> 14]	1 : $S \leftarrow \text{key}$ 2 : $(S', R) \leftarrow \text{next}(S)$	LPR( $f$ )	next-ror, leaknext	✗					AES (3)
[DHY02]	1 : $(K, S) \leftarrow \text{key}$ 2 : $(S', R) \leftarrow \text{next}(S, K, I)$	CIA	getinput, get-state, setinput	✗		✗			+ (3), × (2), AES (2)
		CSA	getinput, get-state, set-state						
		KKA	getinput, get-key						
[BST03]	1 : seed $\leftarrow$ setup 2 : $R \leftarrow \text{next}(\text{seed}, I)$	RES( $\mathcal{F}$ )	next-ror		✗				× (1), [ ] (1)
[BH05]	1 : $S' \leftarrow \text{refresh}(S, I)$ 2 : $(S', R) \leftarrow \text{next}(S)$	ROB( $\mathcal{F}$ )	good-refresh, bad-refresh, get-state, next-ror	✗	✗			× (1), [ ] (1), ⊕ (1)	AES (2)
[DPR <sup>+</sup> 13]	1 : seed $\leftarrow$ setup 2 : $S' \leftarrow \text{refresh}(\text{seed}, S, I)$ 3 : $(S', R) \leftarrow \text{next}(\text{seed}, S)$	ROB( $\gamma^*$ )	$\mathcal{D}$ -refresh, set-state, get-state next-ror	✗	✗	✗		× (1), + (1)	× (1), [ ] (1), AES (9)
[DSSW14]	1 : seed $\leftarrow$ setup 2 : $S' \leftarrow \text{refresh}(\text{seed}, S, I)$ 3 : $(S', R) \leftarrow \text{next}(\text{seed}, S)$	NROB( $\gamma^*, \beta$ )	$\mathcal{D}$ -refresh, set-state, get-state next-ror	✗	✗	✗	✗	+ (1), × (2), ⊕ (1), [ ] (2), AES (11)	AES (4)

✗: The construction involves a secure standard generator  $\mathbf{G}$ , a strong extractor Ext, a strong accumulator Acc or a secure scheduler  $SC$ .

\*: Secure constructions for each property involves multiplications (×), additions (+), XOR (⊕), truncations ([ ]), AES computations (AES).



## References

- [ABP<sup>+</sup>15] Michel Abdalla, Sonia Belaïd, David Pointcheval, Sylvain Ruhault, and Damien Vergnaud. Robust pseudo-random number generators with input secure against side-channel attacks. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15: 13th International Conference on Applied Cryptography and Network Security*, volume 9092 of *Lecture Notes in Computer Science*, pages 635–654, New York, NY, USA, June 2–5, 2015. Springer, Heidelberg, Germany.
- [AES01] Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.
- [AK12] George Argyros and Aggelos Kiayias. I forgot your password: randomness attacks against php applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.
- [ANS85] ANSI X9.17 (revised). American National Standard for Financial Institution Key Management (Wholesale), American Bankers Association, 1985.
- [BDJR97] Mihir Bellare, Anand Desai, E. Jorjipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 394–403. IEEE Computer Society, 1997.
- [BDPV10] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge-based pseudo-random number generators. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 33–47, Santa Barbara, CA, USA, August 17–20, 2010. Springer, Heidelberg, Germany.
- [BGS15] Sonia Belaïd, Vincent Grosso, and François-Xavier Standaert. Masking and leakage-resilient primitives: One, the other(s) or both? *Cryptography and Communications*, 7(1):163–184, 2015.
- [BH05] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM CCS 05: 12th Conference on Computer and Communications Security*, pages 203–212, Alexandria, Virginia, USA, November 7–11, 2005. ACM Press.
- [BK15] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A, 2015.
- [BKR94] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO'94*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358, Santa Barbara, CA, USA, August 21–25, 1994. Springer, Heidelberg, Germany.
- [BM82] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo random bits. In *23rd Annual Symposium on Foundations of Computer Science*, pages 112–117, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.

- [BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.
- [BST03] Boaz Barak, Ronen Shaltiel, and Eran Tromer. True random number generators secure in a changing environment. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 166–180, Cologne, Germany, September 8–10, 2003. Springer, Heidelberg, Germany.
- [BY03] Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18, San Francisco, CA, USA, April 13–17, 2003. Springer, Heidelberg, Germany.
- [CFN<sup>+</sup>14] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, and Hovav Shacham. On the practical exploitability of Dual EC in TLS implementations. In *Proceedings of the 23rd USENIX Conference on Security Symposium, SEC’14*, pages 319–335, Berkeley, CA, USA, 2014. USENIX Association.
- [Che09] Mahdi Cheraghchi. Capacity achieving codes from randomness conductors. *CoRR*, abs/0901.1866, 2009.
- [CMG<sup>+</sup>16] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohneney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the Juniper Dual EC incident. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16: 23rd Conference on Computer and Communications Security*, pages 468–479, Vienna, Austria, October 24–28, 2016. ACM Press.
- [CR14] Mario Cornejo and Sylvain Ruhault. Characterization of real-life PRNGs under partial state corruption. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14: 21st Conference on Computer and Communications Security*, pages 1004–1015, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.
- [CVE] <https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0166>.
- [dB93] Bert den Boer. A simple and key-economical unconditional authentication scheme. *Journal of Computer Security*, volume 2, pages 65–72, 1993. IOS Press, Amsterdam, The Netherlands.
- [DGG<sup>+</sup>15] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. A formal treatment of backdoored pseudorandom generators. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 101–126, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.
- [DGMP92] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In Werner Kuich,

- editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1992.
- [DGP07] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the Windows random number generator. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 07: 14th Conference on Computer and Communications Security*, pages 476–485, Alexandria, Virginia, USA, October 28–31, 2007. ACM Press.
- [DHY02] Anand Desai, Alejandro Hevia, and Yiqun Lisa Yin. A practice-oriented treatment of pseudorandom number generators. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 368–383, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.
- [DPR<sup>+</sup>13] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergnaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input: `/dev/random` is not robust. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 647–658, Berlin, Germany, November 4–8, 2013. ACM Press.
- [DPSW16] Jean Paul Degabriele, Kenneth G. Paterson, Jacob C. N. Schuldt, and Joanne Woodage. Backdoors in pseudorandom number generators: Possibility and impossibility results. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 403–432, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.
- [DRV12] Yevgeniy Dodis, Thomas Ristenpart, and Salil P. Vadhan. Randomness condensers for efficiently samplable, seed-dependent sources. In Ronald Cramer, editor, *TCC 2012: 9th Theory of Cryptography Conference*, volume 7194 of *Lecture Notes in Computer Science*, pages 618–635, Taormina, Sicily, Italy, March 19–21, 2012. Springer, Heidelberg, Germany.
- [DSS00] Digital signature standard (dss), fips pub 186-2 with change notice. National Institute of Standards and Technology (NIST), FIPS PUB 186-2, U.S. Department of Commerce, January 2000.
- [DSSW14] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. How to eat your entropy and have it too - optimal recovery strategies for compromised RNGs. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 37–54, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.
- [FPS12] Sebastian Faust, Krzysztof Pietrzak, and Joachim Schipper. Practical leakage-resilient symmetric cryptography. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 213–232, Leuven, Belgium, September 9–12, 2012. Springer, Heidelberg, Germany.
- [FSK10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

- [GB01] Shafi Goldwasser and Mihir Bellare. Lecture notes on cryptography, 2001.
- [GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986.
- [GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the Linux random number generator. In *2006 IEEE Symposium on Security and Privacy*, pages 371–385, Berkeley, CA, USA, May 21–24, 2006. IEEE Computer Society Press.
- [GT16] Peter Gazi and Stefano Tessaro. Provably robust sponge-based PRNGs and KDFs. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 87–116, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.
- [Gut98] Peter Gutmann. Software generation of practically strong random numbers. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM’98, pages 19–19, Berkeley, CA, USA, 1998. USENIX Association.
- [HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security’12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.
- [HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.
- [Jen96] Robert J. Jenkins Jr. ISAAC. In Dieter Gollmann, editor, *Fast Software Encryption – FSE’96*, volume 1039 of *Lecture Notes in Computer Science*, pages 41–49, Cambridge, UK, February 21–23, 1996. Springer, Heidelberg, Germany.
- [JP14] Dimitar Jetchev and Krzysztof Pietrzak. How to fake auxiliary input. In Yehuda Lindell, editor, *TCC 2014: 11th Theory of Cryptography Conference*, volume 8349 of *Lecture Notes in Computer Science*, pages 566–590, San Diego, CA, USA, February 24–26, 2014. Springer, Heidelberg, Germany.
- [KHL13] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of Android OpenSSL’s pseudo random number generator. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 659–668, Berlin, Germany, November 4–8, 2013. ACM Press.
- [KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In Serge Vaudenay, editor, *Fast Software Encryption – FSE’98*, volume 1372 of *Lecture Notes in Computer Science*, pages 168–188, Paris, France, March 23–25, 1998. Springer, Heidelberg, Germany.
- [LHA<sup>+</sup>12] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 626–642, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

- [LMO<sup>+</sup>14] Jake Longo, Daniel P. Martin, Elisabeth Oswald, Daniel Page, Martijn Stam, and Michael Tunstall. Simulatable leakage: Analysis, pitfalls, and new constructions. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 223–242, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.
- [MMS13] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. Randomly failed! The state of randomness in current Java implementations. In Ed Dawson, editor, *Topics in Cryptology – CT-RSA 2013*, volume 7779 of *Lecture Notes in Computer Science*, pages 129–144, San Francisco, CA, USA, February 25 – March 1, 2013. Springer, Heidelberg, Germany.
- [MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.
- [MR04] Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In Moni Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer Science*, pages 278–296, Cambridge, MA, USA, February 19–21, 2004. Springer, Heidelberg, Germany.
- [Pie09] Krzysztof Pietrzak. A leakage-resilient mode of operation. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 462–482, Cologne, Germany, April 26–30, 2009. Springer, Heidelberg, Germany.
- [SEC] <http://www.securitytracker.com/id/1028916>.
- [SPY13] François-Xavier Standaert, Olivier Pereira, and Yu Yu. Leakage-resilient symmetric cryptography under empirically verifiable assumptions. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 335–352, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.
- [ST15] Thomas Shrimpton and R. Seth Terashima. A provable-security analysis of Intel’s secure key RNG. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 77–100, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.
- [TV00] Luca Trevisan and Salil P. Vadhan. Extracting randomness from samplable distributions. In *41st Annual Symposium on Foundations of Computer Science*, pages 32–42, Redondo Beach, CA, USA, November 12–14, 2000. IEEE Computer Society Press.
- [Vie03] John Viega. Practical random number generation in software. In *19th Annual Computer Security Applications Conference (ACSAC 2003), 8-12 December 2003, Las Vegas, NV, USA*, pages 129–140. IEEE Computer Society, 2003.
- [Yao82] Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.
- [YS13] Yu Yu and François-Xavier Standaert. Practical leakage-resilient pseudorandom objects with minimum public randomness. In Ed Dawson, editor, *Topics in Cryptology – CT-RSA 2013*, volume 7779 of *Lecture Notes in Computer*

---

*Science*, pages 223–238, San Francisco, CA, USA, February 25 – March 1, 2013. Springer, Heidelberg, Germany.

- [YSPY10] Yu Yu, François-Xavier Standaert, Olivier Pereira, and Moti Yung. Practical leakage-resilient pseudorandom generators. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10: 17th Conference on Computer and Communications Security*, pages 141–151, Chicago, Illinois, USA, October 4–8, 2010. ACM Press.

## A Pseudo-Random Functions and Permutations

We recall the definitions of a pseudo-random function from [BKR94]. A pseudo-random function is a family of functions such that no adversary can computationally distinguish the input/output behavior of a random instance from this family from the input/output behavior of a random function.

Hence in this security model the adversary can give inputs to the function and gets the corresponding output in a black-box way. Intuitively, as explained in [BKR94], the pseudo-randomness of a function family is its ‘distance’ from the ensemble of the family of all functions. This notion was originally proposed by Goldreich, Goldwasser and Micali [GGM86]. They explain the notion with the following intuitive example. Consider the set  $\mathcal{F}^k$  of all functions from  $\{0, 1\}^k$  to  $\{0, 1\}^k$ . This set has cardinality  $2^{k \cdot 2^k}$ , hence to describe a (random) function from this set, we would need  $k \cdot 2^k$  bits, which is impractical. Suppose now that we select a set of cardinality  $2^k$ , denoted  $\hat{\mathcal{F}}^k$  and such that  $\hat{\mathcal{F}}^k \subset \mathcal{F}^k$ . This allows to build a family of functions, where each function is indexed with a unique index in  $\{0, 1\}^k$ . The family  $\hat{\mathcal{F}}^k$  is pseudo-random if no adversary can computationally distinguish the functions from  $\hat{\mathcal{F}}^k$  from the functions in  $\mathcal{F}^k$ . Let first formalize the notion of Keyed Family of Functions in Definition 23.

**Definition 23** (Keyed Family of Functions). A keyed family of functions is a map  $F : \{0, 1\}^s \times \{0, 1\}^\ell \rightarrow \{0, 1\}^L$ , where (a)  $\{0, 1\}^s$  is the *key space* of  $F$  and  $s$  is the key length (b)  $\{0, 1\}^\ell$  is the *domain* of  $F$  and  $\ell$  is the input length and (c)  $\{0, 1\}^L$  is the *range* of  $F$  and  $L$  is the output length

Hence in a Keyed Family of Functions, each function is specified by a short, random key. As explained, the security objective we give is that the function behaves like a random one, in the sense that a computationally bounded adversary that is given the key cannot distinguish the input-output behavior of the function from a random function. This property is formalized with the security game PRF described in Fig. 25.

<b>proc. initialize</b>	<b>proc. funct-ror(<math>x</math>)</b>
$K \xleftarrow{\$} \{0, 1\}^s;$	$R_0 \leftarrow F(x, K)$
$\text{funtab} \leftarrow \emptyset$	<b>IF</b> $\text{funtab}[x] = \perp,$
$b \xleftarrow{\$} \{0, 1\};$	$\text{funtab}[x] \xleftarrow{\$} \{0, 1\}^L$
	$y \leftarrow \text{funtab}[x]$
<b>proc. finalize(<math>b^*</math>)</b>	$R_1 \leftarrow y$
<b>IF</b> $b = b^*$ <b>RETURN</b> 1	<b>RETURN</b> $R_b$
<b>ELSE RETURN</b> 0	

**Figure 25:** Procedures in Security Game PRF

In this security game, the challenger first generates a random key  $K \xleftarrow{\$} \{0, 1\}^s$  and a bit  $b \xleftarrow{\$} \{0, 1\}$ , then the adversary  $\mathcal{A}$  uses procedure `funct-ror` with chosen inputs. For each input, the challenger generates a real output with function  $F$  or a random output and challenges  $\mathcal{A}$  on its capability to distinguish the output of  $F$  from random. Note that the challenger constructs a lookup table `funtab` for the random outputs to ensure that the evaluation of equal inputs gives equal outputs: `funtab` is first initialized with  $\emptyset$ ; then at each oracle call, if the value does not exist in the lookup table `funtab`, it is randomly created, otherwise it is directly given as a random output.

**Definition 24** (Pseudo-Random Function). A keyed family of functions  $F : \{0, 1\}^s \times \{0, 1\}^\ell \rightarrow \{0, 1\}^L$  is a  $(t, q, \varepsilon)$ -pseudo-random function if for any adversary  $\mathcal{A}$  running in time at most  $t$ , that makes  $q$  calls to procedure `funct-ror`, the advantage of  $\mathcal{A}$  in game PRF is at most  $\varepsilon$ .

Hence a pseudo-random function is a function which cannot be distinguished from a random function by any efficient distinguisher. Sometimes, however, the full power of a pseudo-random function is not needed and it is sufficient that the function cannot be distinguished when queried on random values. Such objects are referred to as *weak pseudo-random functions*. The associated security game WPRF is the same as PRF, except that the inputs of the pseudo-random function  $F$  in the `funct-ror` procedure are not adversarially chosen but are picked at random by the challenger. The procedures are presented in Fig. 26.

<b>proc. initialize</b>	<b>proc. funct-ror</b>
$K \xleftarrow{\$} \{0, 1\}^s;$	$x \xleftarrow{\$} \{0, 1\}^s$
$\text{funtab} \leftarrow \emptyset$	$R_0 \leftarrow F(x, K)$
$b \xleftarrow{\$} \{0, 1\};$	IF $\text{funtab}[x] = \perp,$
	$\text{funtab}[x] \xleftarrow{\$} \{0, 1\}^L$
<b>proc. finalize(<math>b^*</math>)</b>	$y \leftarrow \text{funtab}[x]$
IF $b = b^*$ RETURN 1	$R_1 \leftarrow y$
ELSE RETURN 0	RETURN $(x, R_b)$

**Figure 26:** Procedures in Security Game WPRF

**Definition 25** (Weak Pseudo-Random Function). A keyed family of functions  $F : \{0, 1\}^s \times \{0, 1\}^\ell \rightarrow \{0, 1\}^L$  is a  $(t, q, \varepsilon)$ -weak pseudo-random function if for any adversary  $\mathcal{A}$  running in time at most  $t$ , that makes  $q$  calls to procedure `funct-ror`, the advantage of  $\mathcal{A}$  in game WPRF is at most  $\varepsilon$ .

<b>proc. initialize()</b>	<b>proc. funct-ror(<math>x</math>)</b>
$K \xleftarrow{\$} \{0, 1\}^n;$	$R_0 \leftarrow F(x_i, K)$
$\text{funtab} \leftarrow \emptyset;$	IF $\text{funtab}[x] = \perp,$
$T \leftarrow \emptyset;$	$\text{funtab}[x] \xleftarrow{\$} \{0, 1\}^n \setminus T$
$b \xleftarrow{\$} \{0, 1\};$	$T = T \cup \text{funtab}[x_i]$
	$y \leftarrow \text{funtab}[x]$
<b>proc. finalize(<math>b^*</math>)</b>	$R_1 \leftarrow y$
IF $b = b^*$ RETURN 1	RETURN $R_b$
ELSE RETURN 0	

**Figure 27:** Procedures in Security Game PRP

In a Keyed Family of Functions, each function is specified by a short, random key. One can similarly define a Keyed Family of Permutations, where each function is a permutation.

We can define an objective similar to that for pseudo-random functions, in the sense that a computationally bounded adversary that is given the key cannot distinguish the input-output behavior of the permutation from a random one. This property is formalized with the security game PRP described in Fig. 27.

**Definition 26** (Pseudo-Random Permutation). A keyed family of permutations  $F : \{0, 1\}^p \times \{0, 1\}^n \rightarrow \{0, 1\}^p$  is a  $(t, q, \varepsilon)$ -pseudo-random permutation if for any adversary  $\mathcal{A}$  running in time at most  $t$ , that makes  $q$  calls to procedure `funct-ror`, the advantage of  $\mathcal{A}$  in game PRP is at most  $\varepsilon$ .

The following lemma, referred to the ‘PRF/PRP Switching Lemma’ shows the relation an advantage in game PRF and an advantage in game PRP. See [GB01] for a complete proof of this Lemma.

**Lemma 2.** *Let  $n \geq 1$  be an integer. Let  $\mathcal{A}$  be an adversary that makes at most  $q$  queries. Then:*

$$|\text{Adv}_{\mathcal{A}}^{\text{PRF}} - \text{Adv}_{\mathcal{A}}^{\text{PRP}}| \leq \frac{q(q-1)}{2^{n+1}}.$$



## B Proof of the Leftover Hash Lemma

The proof we present is adapted from [DRV12] and [Che09]. Fix any  $I \neq I' \in \{0, 1\}^p$ , with  $\mathbf{H}_\infty(I) \geq k$  and  $\mathbf{H}_\infty(I') \geq k$ . Fix  $X \in \{0, 1\}^s$  independently of  $I$  and  $I'$  and  $R \in \{0, 1\}^n$ , with  $\mathbf{H}_\infty(R|X) \geq k'$ . First consider the statistical distance between  $(X, h_X(I))$  and  $(X, R)$ . We introduce a second notion of distance between two random variables  $X$  and  $Y$ :  $\Delta_2(X, Y) = \sqrt{\sum_x |\Pr[X = x] \Pr[Y = x]|}$ , and we define the collision probability of a random variable  $X$  as the probability that two independent samples of  $X$  are equal:  $\mathbf{CP}(X) = \sum_x \Pr[X = x]^2$ .

We can bound the statistical distance between  $(X, h_X(I))$  and  $(X, R)$  by their  $\Delta_2$  distance:  $\mathbf{SD}((X, h_X(I)), (X, \mathcal{U}_m)) \leq \frac{1}{2} \sqrt{2^s \cdot 2^{k'}} \cdot \Delta_2((X, h_X(I)), (X, R))$ , and we have  $\Delta_2((X, h_X(I))^2 = \Delta_2((X, h_X(I)), (X, R))^2 + 2^{-k'-s}$ .

Now as  $\Delta_2((X, h_X(I))^2 \leq \mathbf{CP}(X) \cdot (\Pr_I[I = I'] + \Pr_X[I \neq I' \mid h_X(I) = h_X(I')])$ , and as  $I$  and  $I'$  are sampled independently of  $X$ , as  $\mathbf{H}_\infty(I) \geq k$  and  $\mathbf{H}_\infty(I') \geq k$  and as  $\mathcal{H}$  is  $2^{-n} \cdot (1 + \alpha)$ -universal, we have that:

$$\Delta_2((X, h_X(I))^2 \leq 2^{-s} \cdot (2^{-k} + 2^{-n} \cdot (1 + \alpha)),$$

and finally the statistical distance between  $(X, h_X(I))$  and  $(X, R)$  is bounded by:

$$\frac{1}{2} \sqrt{2^s \cdot 2^{k'}} \sqrt{2^{-s-k} + 2^{-s-n}(1 + \alpha) - 2^{-s-k'}}.$$

Following, the hash functions family  $\mathcal{H}$  is a strong  $(k, k', \varepsilon)$ -condenser, where:

$$\varepsilon = \frac{1}{2} \sqrt{2^{k'-k} + 2^{k'-n}(1 + \alpha) - 1}.$$

Then setting  $k' = m$ , the hash functions family  $\mathcal{H} = \{h_X : \{0, 1\}^p \rightarrow \{0, 1\}^n\}_{X \in \{0, 1\}^s}$ , is a  $(k, \varepsilon)$ -strong extractor for  $\varepsilon = \frac{1}{2} \sqrt{2^{n-k} + \alpha}$  and setting  $k' = k$ , the hash functions family  $\mathcal{H} = \{h_X : \{0, 1\}^p \rightarrow \{0, 1\}^n\}_{X \in \{0, 1\}^s}$ , is a  $(k, \varepsilon)$ -strong accumulator for  $\varepsilon = \frac{1}{2} \sqrt{2^{k-n}(1 + \alpha)}$ .

## C Recovering and Preserving Security

The notion of *recovering security* considers an adversary that compromises the state to some arbitrary value  $S_0$ , either by asking for the state (`get-state`), setting it (`set-state`) or with the output (`next-ror`) when the internal state is unsafe. Afterwards, sufficient calls to  $\mathcal{D}$ -refresh are made to increase the entropy estimate  $c$  above the threshold  $\gamma^*$ . The *recovering* process should make the bit  $b$  involved in the `next-ror` procedure indistinguishable: when the internal state is considered to be secure, the output randomness  $R$  should look indistinguishable from random.

Formally, we consider the security game RECOV for a pseudo-random number generator with input (`setup`, `refresh`, `next`), whose procedures are described in Fig. 28.

<pre> <b>proc.</b> initialize(<math>\mathcal{D}</math>) seed <math>\stackrel{\\$}{\leftarrow}</math> setup; <math>\sigma_0 \leftarrow 0</math>; <math>b \stackrel{\\$}{\leftarrow} \{0, 1\}</math>; <b>FOR</b> <math>k = 1</math> <b>TO</b> <math>q_r</math> <b>DO</b>   <math>(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})</math> <b>END FOR</b> <math>k \leftarrow 0</math>; <b>OUTPUT</b> seed, <math>(\gamma_k, z_k)_{k=1, \dots, q_r}</math>  <b>proc.</b> finalize(<math>b^*</math>) <b>IF</b> <math>b = b^*</math> <b>RETURN</b> 1 <b>ELSE</b> <b>RETURN</b> 0 </pre>	<pre> <b>proc.</b> getinput <math>k \leftarrow k + 1</math> <b>OUTPUT</b> <math>I_k</math>  <b>proc.</b> set-state(<math>S^*</math>) <math>S \leftarrow S^*</math> <math>c \leftarrow 0</math>  <b>proc.</b> <math>\mathcal{D}</math>-refresh <math>k \leftarrow k + 1</math>; <math>S = \text{refresh}(S, I_k)</math>; <b>IF</b> <math>c &lt; \gamma^*</math>,   <math>c = \min(c + \gamma_k, n)</math> </pre>	<pre> <b>proc.</b> next-ror <math>(S^{(0)}, R^{(0)}) \leftarrow \text{next}(S)</math> <math>(S^{(1)}, R^{(1)}) \stackrel{\\$}{\leftarrow} \{0, 1\}^{n+\ell}</math> <b>RETURN</b> <math>(S^{(b)}, R^{(b)})</math>, <math>(I_{k+1}, \dots, I_{q_r})</math> </pre>
---	---	---

**Figure 28:** Procedures in Security Game RECOV( $q_r, \gamma^*$ )

The security game RECOV is described as follow, with an adversary  $\mathcal{A}$ , a sampler  $\mathcal{D}$ , and bounds  $q_r, \gamma^*$ :

1. The challenger generates a seed  $\text{seed} \stackrel{\$}{\leftarrow} \text{setup}$  and a bit  $b \stackrel{\$}{\leftarrow} \{0, 1\}$  uniformly at random. It sets  $\sigma_0 = 0$  and for  $k = 1, \dots, q_r$ , it computes  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$ , initializes  $k = 0$  and sets  $c = 0$ . It then gives back the `seed` and the values  $\gamma_1, \dots, \gamma_{q_r}$  and  $z_1, \dots, z_{q_r}$  to the adversary.
2. The adversary gets access to an oracle `getinput` which on each invocation increments  $k := k + 1$  and outputs  $I_k$ .
3. At some point the adversary  $\mathcal{A}$  calls the procedure `set-state`: it sets a chosen internal state  $S^* \in \{0, 1\}^n$ . It then chooses an integer  $d$  such that  $k + d \leq q_r$  and  $\gamma_{k+1} + \dots + \gamma_{k+d} \geq \gamma^*$ , then calls  `$\mathcal{D}$ -refresh`  $d$  times: this procedure updates the state  $S := \text{refresh}(S, I_{k+j})$  and updates  $c \leftarrow c + \gamma_k$  sequentially.
4. Eventually, the challenger sets  $(S^{(0)}, R^{(0)}) \leftarrow \text{next}(S)$  and generates  $(S^{(1)}, R^{(1)}) \stackrel{\$}{\leftarrow} \{0, 1\}^{n+\ell}$ . It then gives  $(S^{(b)}, R^{(b)})$  to the adversary, together with the next inputs  $I_{k+1}, \dots, I_{q_r}$  (if  $k$  was the number of `refresh`-queries asked up to this point);
5. The adversary  $\mathcal{A}$  outputs a bit  $b^*$ .

The output of the game is the output of the `finalize` oracle at the end, which is 1 if the adversary correctly guesses the challenge bit, and 0 otherwise. Note that the challenge concerns the total output of the `next` algorithm. We define the advantage of the adversary  $\mathcal{A}$  and sampler  $\mathcal{D}$  in the above game as  $|2 \Pr[b^* = b] - 1|$ .

**Definition 27** (Recovering Security). A pseudo-random number generator with input (`setup`, `refresh`, `next`) is said  $(t, q_r, \gamma^*, \varepsilon)$ -recovering if for any adversary  $\mathcal{A}$  and sampler  $\mathcal{D}$ , both running in time  $t$ , the advantage of  $\mathcal{A}$  in Game RECOV( $q_r, \gamma^*$ ) is at most  $\varepsilon$ .

The preserving security notion considers a secure internal state. After several calls to  $\mathcal{D}$ -refresh with known (and even chosen) inputs, the internal state should remain secure. An initial state  $S$  is generated with entropy  $n$ . Then it is refreshed with arbitrary many calls to  $\mathcal{D}$ -refresh. This is the *preserving* process, which should make the bit  $b$  involved in the next-ror procedure indistinguishable: since the internal state is considered as secure, the output randomness  $R$  should look indistinguishable from random.

Formally, we consider the security game PRES for a pseudo-random number generator with input (setup, refresh, next), whose procedures are described in Fig. 29.

<b>proc.</b> initialize( $\mathcal{D}$ ) seed $\stackrel{\$}{\leftarrow}$ setup; $S \stackrel{\$}{\leftarrow} \{0, 1\}^n$ ; $b \stackrel{\$}{\leftarrow} \{0, 1\}$ ; OUTPUT seed	<b>proc.</b> $\mathcal{D}$ -refresh( $I$ ) $S = \text{refresh}(S, I)$	<b>proc.</b> next-ror $(S^{(0)}, R^{(0)}) \leftarrow \text{next}(S)$ $(S^{(1)}, R^{(1)}) \stackrel{\$}{\leftarrow} \{0, 1\}^{n+\ell}$ RETURN $(S^{(b)}, R^{(b)})$
<b>proc.</b> finalize( $b^*$ ) IF $b = b^*$ RETURN 1 ELSE RETURN 0		

**Figure 29:** Procedures in Security Game PRES

The security game PRES is described as follow, with an adversary  $\mathcal{A}$  and a sampler  $\mathcal{D}$ :

1. The challenger generates an initial state  $S \stackrel{\$}{\leftarrow} \{0, 1\}^n$ , a seed  $\text{seed} \leftarrow \text{setup}$ , and a bit  $b \stackrel{\$}{\leftarrow} \{0, 1\}$  uniformly at random. It gives back the `seed` to the adversary;
2. The adversary  $\mathcal{A}$  gets `seed` and can ask as many queries as it wants to the oracles  $\mathcal{D}$ -refresh but with chosen inputs  $I$  to the  $\mathcal{D}$ -refresh-queries. The  $\mathcal{D}$ -refresh procedure simply applies the refresh algorithm to the current state and the input.
3. Eventually, the challenger sets  $(S^{(0)}, R^{(0)}) \leftarrow \text{next}(S)$  and generates  $(S^{(1)}, R^{(1)}) \stackrel{\$}{\leftarrow} \{0, 1\}^{n+\ell}$ . It then gives  $(S^{(b)}, R^{(b)})$  to the adversary.
4. The adversary  $\mathcal{A}$  outputs a bit  $b^*$ .

The output of the game is the output of the `finalize` oracle at the end, which is 1 if the adversary correctly guesses the challenge bit, and 0 otherwise. Note that the challenge concerns the total output of the `next` algorithm. We define the advantage of the adversary  $\mathcal{A}$  in the above game as  $|2 \Pr[b^* = b] - 1|$ .

**Definition 28** (Preserving Security). A pseudo-random number generator with input (setup, refresh, next) is said  $(t, \varepsilon)$ -preserving if for any adversary  $\mathcal{A}$  and sampler  $\mathcal{D}$ , both running in time  $t$ , the advantage of  $\mathcal{A}$  in the game PRES is at most  $\varepsilon$ .

Dodis *et al.* [DPR<sup>+</sup>13] proved that the recovering and preserving security notions together imply the full notion of robustness. This result is of paramount importance as it allows to prove the full robustness of a given construction with the notions of recovering and preserving, that are in practice easier to assess. This result has been used intensively to construct a robust generator, and also in the different model extensions.

**Theorem 1.** *If a pseudo-random number generator with input (setup, refresh, next) has both  $(t, q_r, \gamma^*, \varepsilon_r)$ -recovering security and  $(t, \varepsilon_p)$ -preserving security, then it is  $((t', q_r, q_n, q_s), \gamma^*, q_n(\varepsilon_r + \varepsilon_p))$ -robust where  $t' \approx t$ .*

## D Potential Vulnerability of NIST CTR\_DRBG

We identify a potential vulnerability of the generator described in [BK15], named CTR\_DRBG, and published as a standard by the NIST. As before, if we allow the distribution sampler to depend on `seed`, the adversary can mount an attack against the robustness of the generator. Here the critical point is that the parameter `seed` is not defined in the specification [BK15], hence an assumption shall be made on its definition. A careful analysis of the specification shows that it defines a public parameter  $K = 0x00010203040506070809101112131415$ , which is used exactly for randomness extraction (through a ‘derivation function’ that we describe below). If we allow the distribution sampler  $\mathcal{D}$  to sample an input that depends on  $K$ , the adversary  $\mathcal{A}$  can mount an attack against the robustness of the generator.

Let us first describe the operations of CTR\_DRBG. The complete description of CTR\_DRBG is given in [BK15]; here we give a shorter description that focuses on important facts. Also note that the generator uses a block cipher (`bc`) during its operations. In our description, we assume that the block cipher is AES\_128. We verified that our attack works independently of this choice. We intentionally simplified the description of CTR\_DRBG:

- The specification separates the input used to refresh the generator into two components: the ‘source entropy input’ and the ‘additional input’, the former being used to refresh the internal state during output generation. Note that this is close to the security model of Desai *et al.* [DHY02], described in Sect. 4.1. As noted in the following sections, we prefer to consider the complete list of inputs as a single entity, therefore we will drop the ‘additional input’ parameter in our descriptions and only consider that there is one class of input, the ‘source entropy input’. This is equivalent to set the ‘additional input’ to  $\emptyset$  in the descriptions.
- The specification considers two cases, depending on the use of a ‘derivation function’ named `Block_Cipher_df`. The difference between these two cases is the following: for a given input, either the input is directly used ‘as is’ or the input is first transformed with an internal function (the so-called ‘derivation function’) and then afterwards used by the generator. Whenever an algorithm uses the function `Block_Cipher_df`, the algorithm is named ‘with derivation’. In our descriptions, we only keep the algorithms ‘with derivation’ as our attacks are related to the use of this function.
- A ‘Setup’ function and an ‘Instantiate’ function are defined, that are used to initialize the internal state of the generator. In our description, we do not take into account these algorithms, as we focus on the algorithm used to refresh the internal state of the generator (named the ‘Reseed function’ in the specification) and the algorithm used to generate output (named the ‘Generate function’ in the specification). We omit these functions because our attack relies on a state compromise and for any initialization value, the adversary has access to it.

The internal state of CTR\_DRBG is composed of three parts,  $S = (V, K, \text{ctr})$  where:  $|V| = 128$ ,  $|K| = 128$  and `ctr` is a counter that indicates the number of requests for pseudo-random bits since instantiation or reseeding. The values of  $V$  and  $K$  are the critical values of the internal state (i.e.,  $V$  and  $K$  are the ‘secret values’ of the internal state).

### CTR\_DRBG Reseed Algorithm

The Reseed algorithm is described in Fig. 30. It takes as input the current values for  $V$ ,  $K$ , and `ctr`, and the input  $I$ . The output from the Reseed function is the new working state, the new values for  $V$ ,  $K$ , and `ctr`. Two Reseed algorithms are defined, one using a derivation function `Block_Cipher_df`, one not using this function. As noted before, we focus on the one using the derivation function.

CTR_DRBG Reseed
<b>Require:</b> $S = (V, K, \text{ctr}), I$ <b>Ensure:</b> $S' = (V', K', \text{ctr}')$ 1: $(K', V') = \text{CTR\_DRBG\_update}(\text{Block\_Cipher\_df}(I, 256), K, V)$ 2: $\text{ctr}' = 1$ 3: <b>return</b> $(V', K', \text{ctr}')$

Figure 30: NIST CTR\_DRBG Reseed

### CTR\_DRBG Generate Algorithm

The Generate algorithm is described in Fig. 31. It takes as input the current values for  $V$ ,  $K$ ,  $\text{ctr}$  and  $n$ , the number of pseudo-random bits to be returned. It outputs  $R$ , the pseudo-random bits returned, and the new values for  $V$ ,  $C$ , and  $\text{ctr}$ . Two Generate algorithms are defined, one using a derivation function `Block_Cipher_df`, one not using this function. As noted before, we focus on the one using the derivation function.

CTR_DRBG Generate
<b>Require:</b> $S = (V, K, \text{ctr}), n$ <b>Ensure:</b> $S' = (V', K', \text{ctr}'), R$ 1: $U = \emptyset$ 2: <b>while</b> $\text{len}(U) < n$ 3: $V' = V' + 1 \bmod 2^{128}$ 4: $U = [U    \text{AES\_ECB\_Encrypt}(K', V')]$ 5: <b>endwhile</b> 6: $R = [U]^n$ 7: $(K', V') = \text{CTR\_DRBG\_Update}(I_a, K', V')$ 8: $\text{ctr}' = \text{ctr} + 1$ 9: <b>return</b> $(V', K', \text{ctr}'), R$

Figure 31: NIST CTR\_DRBG Generate

### CTR\_DRBG\_Update Algorithm

The two previous algorithms both rely on an internal algorithm, named `CTR_DRBG_Update`, described in Fig. 32. It takes as input  $I$ , the data to be used, the current value of  $K$  and  $V$ , and outputs the new value for  $K$  and  $V$ .

CTR_DRBG Update
<b>Require:</b> $V, K, I$ <b>Ensure:</b> $V', K'$ 1: $U = \emptyset$ 2: <b>while</b> $\text{len}(U) < (k + 128)$ 3: $V' = V + 1 \bmod 2^{128}$ 4: $U = [[U    \text{AES\_ECB\_Encrypt}(K, V')]^{k+128} \oplus I]$ 5: $K' = [U]^k$ 6: $V' = [U]_{128}$ 7: <b>endwhile</b> 8: <b>return</b> $(V', K')$

Figure 32: NIST CTR\_DRBG\_Update

### Block\_Cipher\_df Function

The derivation function `Block_Cipher_df` is used in the previous algorithms. It is described in Fig. 33. This function uses the public parameter  $K = 0x00010203040506070809101112131415$  as a key to encrypt the input of the generator.

CTR_DRBG Block_Cipher_df
<b>Require:</b> $I, n$ <b>Ensure:</b> $R$ 1: $L = \text{len}(I)/8, N = \text{len}(n)/8, S = [L  N  I  0x80]$ 2: <b>while</b> $\text{len}(S) \bmod 128 \neq 0$ 3: $S = S  0x00$ 4: <b>endwhile</b> 5: $U = \emptyset, i = 0, K = 0x00010203040506070809101112131415$ 6: <b>while</b> $\text{len}(U) < 256$ 7: $IV = i  0, U = [U  \text{BCC}(K, IV  S)], i = i + 1$ 8: <b>endwhile</b> 9: $K = [U]^{128}, X = [U]_{128}, V = \emptyset$ 10: <b>while</b> $\text{len}(V) < (k + 128)$ 11: $X = \text{AES\_ECB\_Encrypt}(K, X)$ 12: $V = [V  X]$ 13: <b>endwhile</b> 14: <b>return</b> $R = [V]^{128}$

Figure 33: NIST Block\_Cipher\_df

## BCC Function

The BCC function is used in the previous algorithms. It is described in Fig. 34. This function operates a block cipher `AES_ECB_Encrypt`, which corresponds to the block cipher AES in ECB mode, and chains the successive outputs.

CTR_DRBG BCC
<b>Require:</b> $K, I,  I  \bmod 128 = 0$ <b>Ensure:</b> $R,  R  = 128$ 1: $U = 0$ 2: $n =  I /128$ 3: <b>parse</b> $I$ as $[B_n, \dots, B_1]$ 4: <b>for</b> $i = 1$ to $n$ 5: $I = B_i \oplus U$ 6: $U = \text{AES\_ECB\_Encrypt}(K, I)$ 7: <b>endfor</b> 14: <b>return</b> $R = U$

Figure 34: NIST CTR\_DRBG BCC

## Attack against the robustness of CTR\_DRBG

Let us now describe the attack against the security of CTR\_DRBG in the security game ROB (Sect. 4.4). Define the 32-byte distribution  $\mathcal{D}$ . On input a state  $i$ ,  $\mathcal{D}$  updates its state to  $i + 1$  and outputs a 32-byte input  $I^i$ :  $(i + 1; [I_0^i, \dots, I_{31}^i]) \leftarrow \mathcal{D}(i)$ ; where  $I_0, \dots, I_{15}$  are random and  $I_{16}, \dots, I_{31} = \text{AES\_ECB\_Decrypt}(K, I_0, \dots, I_{15})$ , where  $K = 0x00 \dots 15$  (*i.e.*  $\mathcal{D}$  is legitimate with  $\gamma_i = 128$ , in accordance with Def. 15).

Let us consider an adversary  $\mathcal{A}$  against the security of the generator that chooses the distribution  $\mathcal{D}$ , and that makes the following oracle queries in the security game ROB: one get-state, one  $\mathcal{D}$ -refresh with  $I^0$ , one next-ror. Then (following algorithm notations):

- After get-state,  $S, K$  and ctr are known.
- After  $\mathcal{D}$ -refresh, the Reseed algorithm is first applied: the new state is the output of `CTR_DRBG_update(Block_Cipher_df( $I^0, 256$ ),  $K, V$ )` and `ctr = 1`. Let us describe the algorithm `Block_Cipher_df( $I^0, 256$ )`: on input  $I^0$  and 256, `Block_Cipher_df` calculates  $L = 32, N = 32, S = [32||32||I^0||0x80]$  and then  $S = [32||32||I^0||0x80||0x00||\dots||0x00]$ . Next, it calculates `BCC( $K, IV||S$ )`, for  $IV = 0||0$  and  $IV = 1||0$ , with  $K = 0x00 \dots 15$ , then sets  $U = \text{BCC}(K, 1||0||S) || \text{BCC}(K, 0||0||S)$ ,  $K = [U]^{128}, X = [U]_{128}$ .

Let us describe the algorithm  $\text{BCC}(K, IV || S)$ : on input  $S = [32 || 32 || I^0 || 0x80 || 0x00 || \dots || 0x00]$ ,  $IV = 0 || 0$  and  $K = 0x00 \dots 15$ , it parses  $S$  as  $B_4, B_3, B_2, B_1$  and calculates  $I = B_1$ ,  $U = \text{AES\_ECB\_Encrypt}(K, I)$ ,  $I = B_2 \oplus U$ ,  $U = \text{AES\_ECB\_Encrypt}(K, I)$ ,  $I = B_3 \oplus U$ ,  $U = \text{AES\_ECB\_Encrypt}(K, I)$ ,  $I = B_4 \oplus U$ ,  $U = \text{AES\_ECB\_Encrypt}(K, I)$ . However, the input distribution is such that  $B_3 = \text{AES\_ECB\_Decrypt}(K, B_2)$  and therefore the output of algorithm  $\text{BCC}$  is known to  $\mathcal{A}$ . Hence the output of algorithm  $\text{Block\_Cipher\_df}$  is also known to  $\mathcal{A}$  and also the output of the  $\text{Reseed}$  algorithm, although the initial input was of high entropy.

- After  $\text{next-ror}$ , the output of the generator is computed from a known state and is therefore predictable.

In this last  $\text{next-ror}$ -oracle query,  $\mathcal{A}$  obtains a 16-byte string that is predictable, whereas this event should occur with probability  $2^{-128}$ . Therefore  $\mathcal{A}$  can distinguish an output of  $\text{CTR\_DRBG}$  from random in the game  $\text{ROB}(\gamma^*)$ , for all  $\gamma^*$  and this pseudo-random number generator with input is not robust.