Security Analysis for Pseudo-Random Numbers Generators

Sylvain Ruhault

École Doctorale de Sciences Mathématiques de Paris Centre

Security Analysis for Pseudo-Random Number Generators

THÈSE

présentée pour l'obtention du

Doctorat de l'École normale supérieure

(Spécialité Informatique)

par

Sylvain Ruhault

Soutenue publiquement le 30 Juin 2015 devant le jury composé de

Pierre-Alain Fouque ..... *Examinateur*
Marc Girault ..... *Examinateur*
David Pointcheval ..... *Directeur de thèse*
Bart Preneel ..... *Rapporteur*
Emmanuel Prouff ..... *Rapporteur*
Phillip Rogaway ..... *Examinateur*
Nicolas Sendrier ..... *Examinateur*
Damien Vergnaud ..... *Directeur de thèse*

Travaux effectués au sein de l'Équipe de Cryptographie
du Département d'Informatique de l'École normale supérieure

Remerciements

Je voudrais en premier lieu remercier David Pointcheval et Damien Vergnaud pour m'avoir accueilli au laboratoire de cryptographie du Département d'Informatique de l'ENS et pour avoir si bien dirigé mes travaux de thèse. Leur disponibilité, leur rigueur scientifique et leur implication ont grandement contribué à l'aboutissement de ce projet. Je les remercie sincèrement pour tout le temps passé à répondre patiemment à toutes mes questions, scientifiques ou autres, pour avoir lu et relu les différentes versions de ce manuscrit et de mes articles et pour m'avoir aidé dans les répétitions de présentations.

Je voudrais également remercier Eric Dehais pour m'avoir toujours soutenu dans ce projet et pour voir autant engagé Oppida pour son aboutissement dans les meilleures conditions. Je voudrais aussi rendre ici hommage à sa volonté d'investir dans la connaissance et la recherche.

Je remercie vivement l'ensemble des membres du jury qui m'ont fait l'honneur d'évaluer mon travail. Je suis très reconnaissant à Emmanuel Prouff et à Bart Preneel pour l'intérêt qu'ils ont porté à ma thèse en acceptant d'en être rapporteurs. Je remercie également Pierre-Alain Fouque, Marc Girault, Phillip Rogaway et Nicolas Sendrier d'avoir accepté de faire partie du jury en tant qu'examinateurs.

Je remercie tous mes co-auteurs avec qui j'ai eu des collaborations enrichissantes grâce à leurs qualités scientifiques et humaines. En plus de Damien et David, j'ai eu l'opportunité de travailler avec Yevgeniy Dodis lors de son séjour au laboratoire en juin 2012 et de co-écrire avec lui et Daniel Wichs un premier article scientifique. Par la suite j'ai eu l'opportunité de co-écrire un second article avec Mario Cornejo et un troisième avec Sonia Belaïd et Michel Abdalla. J'espère sincèrement que nous aurons l'occasion de travailler ensemble sur de nouveaux projets.

Ces années de thèse ont été l'occasion de rencontres très enrichissantes, que ce soit au laboratoire, à Oppida ou ailleurs. Je voudrais remercier les chercheurs et les étudiants que j'ai pu rencontrer au laboratoire: Adrian, Alain, Angelo, Aurélie, Aurore, Cécile, Céline, Duong-Hieu, Dario, Elisabeth, Fabrice, Geoffroy, Guiseppe, Hoeteck, Houda, Louiza, Itai, Joana, Kenneth, Léo, Mario, Miriam, Pierre-Alain, Pierrick, Rafael, Roch, Sorina, Tancrède, Thomas, Thomas et Vadim; Louiza, la future étudiante d'Oppida; Marion et Patrick, qui m'ont invité à présenter mes travaux au LORIA au GREYC; et toute la communauté C2 rencontrée à Dinard et aux Sept Laux.

Je remercie également l'ENS pour fournir le cadre de travail aussi favorable à la recherche et de m'avoir permis de participer à de nombreuses conférences. J'en profite pour remercier l'ensemble du personnel du Département d'Informatique, en particulier Joëlle Isnard, Valérie Mongiat, Michele Angely et Jacques Beigbeder pour l'efficacité avec laquelle ils ont géré toutes les questions administratives et techniques au cours de ma thèse.

Je pense aussi à tous mes amis et à ma famille qui de près ou de loin m'ont suivi et m'ont encouragé dans ce projet. Merci beaucoup à Christophe, Jean-François, Jan et Aziz pour leurs encouragements précieux.

Je voudrais terminer ces remerciements par les personnes qui comptent le plus pour moi, ma femme Raja et mes deux filles Mona et Nada. Ce projet n'aurait pas été possible sans leur soutien constant et leur joie de vivre communicative.

LEJEUNE: Monsieur le Président, je voudrais poser la même question qu'hier: d'où vient le mot "random"? What is the etymology of that word? Who introduced it? And what does it mean really? Is it just a flap of swatches or what? Is randomness chosen at random? Je voudrais faire remarquer que nous avons deux grands spécialistes de "randomness", et que ni l'un ni l'autre ne sait ce que veut dire le concept qu'ils emploient. Je m'explique. Dans l'utilisation, ils savent parfaitement la fonction de ce mot, ils nous l'ont expliqué de façon parfaitement claire. Mais ce qui m'intéresse, c'est que les mots du vocabulaire sont reliés les uns aux autres, et en fait, l'étymologie nous permet de comprendre la consistance de la pensée humaine. Et quand on rajoute un concept qui ne vient de nulle part, au sens étymologique au moins, on aimerait connaître son histoire.

THOM: The etymology of "random" is very well known. It is the old French word "randon", it was a way of hunting, "chasse à la randon", and the word "randon" has later given the word "randonnée", which means "a long hike".

Figure 1 – Extract from the Proceedings of the plenary session of the Pontifical Academy of Sciences, Vatican City, Italy, October 27-31 1992 [Pul]

.

Contents

Chapter 1

Introduction

1.1 Pseudo-Random Number Generators

Standard Pseudo-Random Number Generators

In cryptography, randomness plays an important role in multiple applications. It is required in fundamental tasks such as key generation, masking and hiding values, nonces and initialization vectors generation. The security of these cryptographic algorithms and protocols relies on a source of unbiased and uniform distributed random bits and cryptography practitioners usually assume that parties have access to perfect randomness. If a user has access to a truly random bit-string, he can use a *deterministic* algorithm to expand into a longer sequence. 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 above algorithm can be defined precisely with a formal security game and is referred to in this thesis as a *standard pseudo-random number generator*. In this situation, the seed of the generator is the most critical part of it as an adversary that has access to it can predict the future output of the generator.

Stateful Pseudo-Random Number Generators

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 defined precisely with a formal security game and is referred to in this thesis as a *stateful pseudo-random number generator*. The generator is modelled here as a stateful algorithm and its security is formalized by the indistinguishability 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 have been implemented in a large amount of systems, including hardware security modules. As a drawback, 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 is then its most critical part, as an adversary that has access to it can predict the future output of the generator. In this thesis, we refer to the *internal state* for the memory of the stateful pseudo-random number generator.

Pseudo-Random Number Generators with Input

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 in this thesis 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 actually 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 adversarially influenced, 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 and is the main objective of this thesis. We present in Chapter 3 the successive models for pseudo-random number generators with inputs that have been proposed, and we present our new security model in Chapter 4. A major contribution of our new security model is the formalization of both these *recovering* and *preserving* properties.

1.2 Randomness Extractors

A randomness extractor takes as input a source of possibly correlated bits, and produces an output which is close to the uniform distribution.

We present in Chapter 2 a survey of the different notions of extractors that we will use for this thesis. In this survey, we show that there is an impossibility result as no deterministic extractor can extract randomness from all sources and therefore we need to consider a family of extractors named *seeded* extractors, that uses a second random parameter seed for extraction. We recall that the existence of seeded extractor is guaranteed (by the probabilistic method), and we present the famous Leftover Hash Lemma that constructively builds randomness extractors from hash function families. As we show, an application of the Leftover Hash Lemma is the contruction of *strong* extractors from universal hash functions families (and similarly the construction of *resilient* extractors from finite pairwise independent hash functions families).

As we explain in Chapter 2, if one wants to build a secure scheme upon seeded extractors, the parameter seed will preferably be made public and a tradeoff shall be made between the independence of seed and the randomness source, the size of the randomness source and the adversary's capabilities:

1. We assume that independence between the seed and the randomness source can not be ensured. One solution is to restrict the randomness sources to use a resilient extractor: this is the solution proposed in [BST03, BH05]. One second solution would be to restrict the adversary's computational capabilities.

2. We assume that independence between the seed and the randomness source can be ensured. As we also want that seed is public, one solution is to use *strong* extractors.

The security model that we propose for pseudo-random number generators with input relies on this second assumption. We therefore exhibit some impossibility results that show that a scheme, secure when independence between the seed and the randomness source is ensured, can

be broken if there is a correlation between them. In particular, we point an explicit impossibility result for the pseudo-random number generator named CTR_DRBG, proposed as a standard by the NIST.

1.3 Security Models

Security Against Source and State Compromise Attacks

Several desirable security properties for stateful pseudo-random number generators and pseudo-random number generators with inputs have been identified in various standards [ISO11, Kil11, ESC05, BK12]. These standards consider adversaries with various means: those who have access to the output of the generator; those who can control (partially or totally) the source of the generator and those who can control (partially or totally) the internal state of the generator (and combination of them). Several security notions have been defined: (a) *Resilience*: an adversary must not be able to predict future outputs even if he can influence the input used to initialize or used to refresh the internal state of the generator and *Forward security (*resp. backward security*)*: an adversary must not be able to predict past (*resp.* future) outputs even if he can compromise the internal state of the generator. Note that backward security implies that the generator is refreshed with new inputs after compromise.

In 1998, Gutmann [Gut98], and Kelsey, Schneier, Wagner and Hall [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 couple of algorithms, one to collect inputs and one second to generate outputs. In 2001, Bellare and Yee [BY03] proposed a dedicated security model to assess *Forward Security*, for which a stateful pseudo-random number generator shall be designed so that it is infeasible to recover any information on previous states or previous output blocks from the compromise of the current state. In 2002, 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. In 2003, 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 settlement 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.

Security Against Side-Channel Attacks

Under the robustness security notion, an adversary can observe the inputs and outputs of a generator, manipulate its entropy source, and compromise its internal state. While this notion seems reasonably strong for practical purposes, it still does not fully consider the reality of embedded devices, which may be subject to *side-channel attacks*. In these attacks, an adversary can exploit the physical leakage of a device by several means such as power consumption, execution time or electromagnetic radiation. While many countermeasures have been proposed to thwart specific attacks, it was only recently that significant efforts have been made to define generic

security models. Among these, the bounded retrieval model [DLW06, Dzi06], for instance, captures attacks where the adversary is limited to a bounded amount of leakage over the entire lifetime of a cryptosystem. The leakage-resilient model [DP08], on the other hand, encompasses many more attacks with only a limitation in the amount of leakage per execution. The global amount of leakage is not limited as in the bounded retrieval model. Since the leakage-resilient model captures most of the known side-channel attacks, it has led to the design of several secure primitives [Pie09, DP10, FPS12, YS13, ABF13]. Note that another model proposed by Prouff and Rivain [PR13] fits well with the reality of embedded security by assuming that every elementary computation in the implementation leaks a noisy function of its input. In that case, the security of the system directly depends on the level of noise.

In the specific context of pseudo-random number generators, several leakage-resilient models and constructions have been proposed so far (e,g., [YSPY10, SPY13, YS13]). The work of Yu *et al.* [YSPY10], for instance, proposes a very efficient construction of a leakage-resilient pseudo-random number generators. Likewise, the work of Standaert *et al.* [SPY13] shows how to obtain very efficient constructions of leakage-resilient pseudo-random number generators by relying on empirically verifiable assumptions. None of these works, however, consider potentially biased random sources.

Security Against Memory Attacks

Designers of pseudo-random number generators with input assume that the internal state S remains secret to the adversary. However, for software implementations this may be unrealistic as the internal state can be partially compromised through memory corruption attacks such as buffer overflows or side-channel attacks. Different memory corruption attacks were presented by Erlingsson *et al.* in [EYP10] and by van der Veen *et al.* in [vdVdSCB12] and faults attacks against cryptographic schemes were presented by Biham and Shamir in [BS97] and by Boneh *et al.* in [BDL01]. For example, recently, the Heartbleed Bug [Hea] affected the OpenSSL cryptographic library. This bug allows an adversary to get the content of the memory of the OpenSSL process run by a server (or a client). Although the adversary can control the size of the compromised memory, the location cannot be controlled. The adversary can get total or partial access to sensitive information as the internal state of the generator.

We present in Chapter 3 a complete description of the security models for source and state compromise attacks and we propose a comparison between these models. We also present in Chapter 3 three proposals of constructions of stateful pseudo-random number generators that are secure against side-channel attacks.

1.4 Potential Weaknesses

Currently there are numerous implementations of pseudo-random number generators with input from different providers, and most of them rely on internal directives and parameters that are poorly documented or even undocumented. In most implementations, a generator contains a dedicated internal state S which is *refreshed* periodically with inputs collected from its environment (such as network input/output, keyboard presses, processor clock cycles) and secondly used to compute pseudo-random strings. The randomness collection task is harder and takes much more time than the output generation task; this is the reason why implementations typically maintains a dedicated memory as the internal state, which, as we mentioned previously, is the most critical part of the generator and therefore needs to be kept secure during its update.

The lack of insurance about the generated random numbers can cause serious damages in cryptographic protocols, and vulnerabilities can be exploited by adversaries to mount concrete attacks.

In 1996, Goldberg and Wagner [Net96] completed an analysis of Netscape pseudo-random number generator used in Version 1.1 of the international version of Netscape's Solaris 2.4 browser. Their analysis showed that the creation of the internal state of the generator only depended on three values: the PID, the PPID and a call to time, mixed together using a linear function and MD5 hash function. Their analysis also showed that any generated cryptographic key only relied on these four values, which could easily be guessed by an adversary.

One other striking example is the failure in the Debian Linux distribution [CVE08], where a commented code in the OpenSSL pseudo-random number generator with input led to insufficient entropy gathering and allowed an adversary to conduct brute force guessing attacks against cryptographic keys.

Moreover, in addition to these concrete attacks, cryptographic algorithms are highly vulnerable to weaknesses in the underlying random number generation process. For instance, several works demonstrated that if nonces for the Digital Signature Algorithm are generated with a weak pseudo-random number generator then the secret key can be quickly recovered after seeing a few signatures (see [NS02] and references therein). This illustrates the need for precise evaluation of pseudo-random number generators with input based on clear security requirements.

Despite this, only few implementations of pseudo-random number generators have been analyzed since [Gut98, KSWH98].

Concerning system pseudo-random number generators with input, an analysis of Linux pseudo-random number generators with input `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. 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.

Lenstra, Hughes, Augier, Bos, Kleinjung and Wachter [LHA+12] showed that a non-negligible percentage of RSA keys share prime factors. Heninger, Durumeric, Wustrow and Halderman [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.*

Concerning application pseudo-random number generators, Argyros and Kiayias [AK12] showed practical attacks on web applications exploiting randomness vulnerabilities in PHP applications. Michaelis *et al.* [MMS13] described and analyzed several Java implementations; they have also identified some weaknesses. More recently, a flaw in the Android pseudo-random number generator with input, identified by Kim, Han and Lee in [KHL13], has been actively exploited against Android-based Bitcoin wallets [SEC13].

1.5 Contributions of this Thesis

New Security Models

Robustness. In 2013, in [DPR+13], in a common work with Dodis, Pointcheval, Vergnaud and Wichs, we proposed the first contribution of this thesis. We extended the previous work of [BH05] and we formalized the accumulation process of a pseudo-random number generator

with input.

We introduced the notion of adversarially 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*. We proposed two simpler notions of security, the *recovering security* that models how a generator should recover from a compromise of its internal state by entropy accumulating, and the *preserving security*, that models how a generator with a non compromised internal state should behave in presence of adversarial inputs. We complemented the *robustness* security model with these stronger adversaries and we proved that taken together, recovering and preserving security imply the full notion of robustness. We proposed a simple and very efficient construction that is provably provably secure (*i.e.* robust) in our new and stronger adversarial model, based on simple operations in a finite field and a standard secure pseudo-random number generator **G**. We also analyzed the pseudo-random number generator with input proposed by Barak and Halevi. This scheme was proven robust in [BH05] but we proved that it does not generically satisfy our new property of entropy accumulation. We presented benchmarks between this construction and the Linux generators that show that our construction is on average more efficient when recovering from a compromised internal state and when generating cryptographic keys.

This work is presented in Chapter 4.

Robustness Against Memory Attacks. In 2014, in [CR14], in a common work with Cornejo, we extended the previous works of [BH05] and [DPR+13] to model the expected security of pseudo-random number generators with input against *Memory Attacks*. These attacks captures real-life situations and refers to situations in which 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. Formalization of security against these attacks is fully described by Akavia, Goldwasser and Vaikuntanathan in [AGV09]. In our work we focused on a class of memory attacks where the adversary directly gets access to some fraction of the internal state of the generator or sets this fraction to a chosen value, we formally extended the security model of [DPR+13] with this new adversary profile and we proved that the original construction of [DPR+13] can be extended in this model.

This work is presented in Chapter 5.

Robustness Against Side-Channel Attacks. In 2015, in [ABP+15], in a common work with Abdalla, Belaid, Pointcheval and Vergnaud, we built a practical and robust pseudo-random number generator with input that can resist side-channel attacks. Since the construction of [DPR+13] seemed to be a good candidate, we used it as the basis of our work. In doing so, we extended its security model to integer the leakage-resilient security and we defined stronger properties for the underlying standard pseudo-random generator for them to resist side-channel attacks. We analyzed the robust construction based on polynomial hash functions given in [DPR+13] showing why its instantiation may be vulnerable to side-channel attacks. We also proposed three concrete instantiations with a small overhead. While two of them are adaptation of existing constructions, the third one is a new proposal which provides a better security at the expense of a larger internal state. We proved that the whole construction and its instantiations are leakage-resilient robust and we provided features on the performances for several security levels. Finally, we gave instantiations of this construction based on AES in counter mode that are only slightly less efficient than the original instantiation proposed in [DPR+13]. Our instantiations only require that the implementation of AES in counter mode is secure against Simple Power Analysis attacks since very few calls are made with the same secret key.

This work is presented in Chapter 6.

Security Analysis of Concrete Pseudo-Random Number Generators

We propose a new analysis of concrete pseudo-random number generators with input that are used in practice in real-life security products.

Security Analysis of the Linux generators `/dev/random` and `/dev/urandom`. In [DPR+13], we gave a precise assessment of the security of the two Linux pseudo-random number generators with input, `/dev/random` and `/dev/urandom`. In particular, we showed several attacks proving that these generators are not robust according to our definition, and do not accumulate entropy properly. These attacks are due to the vulnerabilities of the entropy estimator and the internal mixing function of the generators.

Security Analysis of OpenSSL and Java Generators In [CR14], we gave an analysis of real-life generators using the security model of robustness against memory attacks and we demonstrated how it can help to identify new vulnerabilities. In particular, we showed that a *full* internal state corruption is not necessary to compromise a lot of concrete implementation of real-life generators, instead only a *partial* one may be sufficient. We characterized how a generator can be attacked in order to produce a predictible output and we identified how many bits of the internal state are required to mount an attack against the generator. In this aim, we characterized and gave a new security analysis of pseudo-random number generators with input implementations from widely used providers in real-life applications: OpenSSL, OpenJDK, Android, Bouncycastle and IBM. To our knowledge, while intensively used in practice, these generators had not been evaluated with recent security models. Our analysis revealed new vulnerabilities of these generators due to the implementation of their internal state in several fields that are not updated securely during generators operations.

This work is fully described in Chapter 7.

Chapter 2

Preliminaries

Throughout this thesis we refer to discrete probability distributions. For notations, definitions and theorems presented in this Chapter, we refer to [GB01, Sho06, BR06, Vad12].

2.1 Probabilities

Random Variable. Let X be a random variable over a sample set S. Then X defines a probability distribution $P_X : S \to [0,1]$, where $P_X(x) := \Pr[X = x]$ called the distribution of the random variable X. In this thesis, we will denote by X both the random variable X and the distribution of the random variable X and we denote $x \xleftarrow{\$} X$ when x is sampled according to X. The support of a random variable X is the set $supp(X) = \{x : \Pr[X = x] > 0\}$. If $\mathcal{A}$ is an algorithm, then $\mathcal{A}(X)$ denotes the random variable that samples $x \xleftarrow{\$} X$ and returns $\mathcal{A}(x)$.

Uniform Distribution. Let X be a random variable over a non empty finite sample set S. If $\forall x \in S, P_X(x) = \frac{1}{|S|}$, the random variable X is said uniformly distributed over S, that we denote $X \xleftarrow{\$} S$. Let $n > 0$ be an integer, the uniform distribution over the sample set $\{0,1\}^n$ is denoted $\mathcal{U}_n$.

Independence. 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 $\{X_i\}_{i \in I}$ be a finite family of random variables. The family is *pairwise-independent* if for all $i, j \in I$ such that $i \neq j$, the random variables X_i and X_j are independent.

2.2 Indistinguishability

Statistical Indistinguishability. 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: $\mathbf{SD}(X,Y) = \frac{1}{2}\sum_x |\Pr[X = x] - \Pr[Y = x]|$.

Theorem 1 shows that the statistical distance is a distance. In particular, it satisfies the triangle inequality, that will be useful to build reductions between security notions.

Theorem 1 (Statistical Distance Properties [Sho06])**.** *Let $n > 0$ be an integer and let X, Y and Z be random variables over the sample set $\{0,1\}^n$. Then: $0 \leq \mathbf{SD}(X,Y) \leq 1$, $\mathbf{SD}(X,X) = 0$, $\mathbf{SD}(X,Y) = \mathbf{SD}(Y,X)$ and $\mathbf{SD}(X,Z) \leq \mathbf{SD}(X,Y) + \mathbf{SD}(Y,Z)$.*

Theorem 2 will also be useful when we build reductions between security notions. In particular, it implies that if the statistical distance between two random variables X and Y is small, no efficient algorithm can distinguish between them.

Theorem 2 (Statistical Distance Properties [Sho06])**.** *Let $n > 0$ be an integer and let X and Y be random variables over the sample set $\{0,1\}^n$. Then for every subset $T \subseteq \{0,1\}^n$, we have* $\mathbf{SD}(X,Y) \geq |\Pr[X \in T] - \Pr[Y \in T]|$.

Finally, the random variables X and Y are said ε-close if $\mathbf{SD}(X,Y) \leq \varepsilon$.

Computational Indistinguishability. Let X and Y be two random variables over $\{0,1\}^n$, let t be an integer and let $\mathcal{A}$ be a probabilistic algorithm running in time t, that takes as input a bitstring in $\{0,1\}^n$. Note that the running time t includes both the computation time and the pre-computation time (e.g. memory setting). The t-computational distance between the two random variables X and Y is equal to $\mathbf{CD}_t(X,Y) = \max_{\mathcal{A} \leq t} |\Pr[\mathcal{A}(X) = 1] - \Pr[\mathcal{A}(Y) = 1]|$ where the notation $\max_{\mathcal{A} \leq t}$ denotes that the maximum is over all $\mathcal{A}$ running in time at most t. Theorems 1 and 2 can be stated for the computational distance.

The random variables X and Y are said (t,ε)-close if for any probabilistic algorithm $\mathcal{A}$ running within time t, $\mathbf{CD}_t(X,Y) \leq \varepsilon$. When $t = \infty$, meaning $\mathcal{A}$ is unbounded, then X and Y are ε-close.

2.3 Hash Functions

Let p and m be integers, such that $m < p$. A hash function is a function $h : \{0,1\}^p \to \{0,1\}^m$.

Pairwise Independence. A family of hash functions $\mathcal{H} = \{h : \{0,1\}^p \to \{0,1\}^m\}$ is *pairwise-independent*:

1. $\forall x \in \{0,1\}^p$, $h(x)$ is uniformly distributed in $\{0,1\}^m$, when $h \xleftarrow{\$} \mathcal{H}$.
2. $\forall x_1 \neq x_2 \in \{0,1\}^p$, the random variables $h(x_1)$ and $h(x_2)$ are independent, when $h \xleftarrow{\$} \mathcal{H}$.

The two above conditions can be combined as follow: $\forall x_1, x_2 \in \{0,1\}^p, \forall y_1, y_2 \in \{0,1\}^m$,

$$\begin{aligned} \Pr_{h \xleftarrow{\$} \mathcal{H}}[h(x_1) = y_1 \text{ and } h(x_2) = y_2] &= \Pr_{h \xleftarrow{\$} \mathcal{H}}[h(x_1) = y_1] \cdot \Pr_{h \xleftarrow{\$} \mathcal{H}}[h(x_2) = y_2] \\ &= \frac{1}{2^{2m}} \end{aligned}$$

Universality. A hash functions family $\mathcal{H} = \{h : \{0,1\}^p \to \{0,1\}^m\}$ is ε-universal if for any inputs $x_1 \neq x_2 \in \{0,1\}^p$ we have:

$$\Pr_{h \xleftarrow{\$} \mathcal{H}}[h(x_1) = h(x_2)] \leq \varepsilon.$$

2.4 Game Playing Framework

In this work, we focus on giving precise security properties for systems. In cryptography, a scheme has reductionist security (or provable security), as opposed to heuristic security, if its security requirements can be stated formally in an adversarial model where the capabilities of the adversary are formally described with clear assumptions. This formal description includes the potential accesses of the adversary to the system and its computational resources. In this approach, the security of a cryptographic scheme is based on algorithmic problems that are

supposed to be hard to solve. The scheme is secure as long as the underlying algorithmic problems are difficult and the security of the scheme is proven by reduction to the security of the underlying algorithmic problems.

For our security definitions and proofs we use the code-based game playing framework of [BR06]. A security game involves a challenger and an adversary, denoted $\mathcal{A}$. The adversary will always be modelled with a probabilistic algorithm running in time t. 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. To describe procedures, we use the expression '**proc**. '. When some parameters are adversarially chosen, they are used as input to the procedures. When procedures generate some outputs (as a result of a computation, for example):

- The output is given with a directive named `OUTPUT` when the output is given to the adversary and the security games continues.
- The output is given with a directive named `RETURN` when the output is the result of the security game (which is therefore terminated).

A security game GAME has an $\mathsf{initialize}$ procedure, procedures to respond to adversary oracle queries, and a $\mathsf{finalize}$ procedure. A security game GAME is executed with an adversary $\mathcal{A}$ as follows. First, challenger executes procedure $\mathsf{initialize}$, 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. In this description, $\mathcal{A}$ can be restricted to a limited number or order of oracle queries. When $\mathcal{A}$ terminates, its output becomes the input to the $\mathsf{finalize}$ procedure.

The output of the $\mathsf{finalize}$ procedure is called the output of the security game GAME, and we denote the output of the adversary as $\mathsf{GAME}^{\mathcal{A}}$. Finally we denote the event that this output takes value y as $\mathsf{GAME}^{\mathcal{A}} \Rightarrow y$ and we define the *advantage* of $\mathcal{A}$ in GAME as

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{GAME}} = 2 \times \Pr[\mathsf{GAME}^{\mathcal{A}} \Rightarrow 1] - 1.$$

Our convention is that Boolean flags are assumed initialized to false and that the running time of the adversary $\mathcal{A}$ is defined as the total running time of the game with the adversary in expectation, including the procedures of the game.

To prove a reduction from the security of a scheme to the intractability of an algorithmic problem, we define sequences of security games as follows: the first game is the game that defines the security of the scheme, the last game is the game that defines the intractability of the algorithmic problem, and the games in between describe successive transitions from the two games. We then estimate the distance between the successive security games and the estimation of the reduction uses Theorems 1 and 2 as the properties of the computational distance will ensure that we can bound the (global) distance between the two games by the sum of all distances.

2.5 Shannon Entropy, Min-Entropy

We now model the concept of 'how random' is the distribution of a random variable. We will consider that a phenomenon is described by a random variable and we want to model 'how random' its distribution is. We will name *sources of randomness* or *sources* the random variables that will be used because they 'look random' or they 'contain a certain amount of randomness'. Hence a *source* on $\{0,1\}^p$ is a random variable on $\{0,1\}^p$.

We need a tool to estimate the 'amount of randomness' that is contained in a given source. In

doing so, we will be able to formalize that we can 'extract k bits of randomness' from a source that contains 'n bits of randomness', for $k \leq n$. This idea is captured with the notion of entropy, that is given in Definition 1.

The first notion of entropy (the *Shannon entropy*) is described in the seminal paper of Shannon [Sha48]. Consider a sequence of random variables $X_1, \cdots, X_n$, of distribution probabilities $p_1, \cdots, p_n$. Shannon shows that entropy is the only function that satisfies the three properties: (a) it shall be continuous (b) if $p_i = \frac{1}{n}$, then it shall be maximal (when every outcome is equally like the uncertainty is greatest and hence so is the entropy) and (c) it should be additive. This leads to the notion of 'Shannon entropy', denoted $\mathbf{H}_1$ below.

The second notion of entropy (the *min-entropy*) was first used as a measure of randomness in the seminal work of Chor and Goldreich [CG85], as explained in the survey of Shaltiel [Sha02]. This notion is very close to the notion of randomness extractor, a notion that we will describe in Section 2.6, in the sense that a necessary condition to extract randomness from distributions is that they shall have high min-entropy.

Definition 1 (Entropy)**.** *Let X be a random variable on a sample set S.*

- *The Shannon entropy of X is $\mathbf{H}_1(X) = \mathbb{E}_{x \in S}[-\log \Pr[X = x]]$.*
- *The min-entropy of X is $\mathbf{H}_\infty(X) = \min_{x \in S}\{-\log \Pr[X = x]\}$.*

First note that if X is uniform on (e.g.) $\{0,1\}^{128}$, then $\mathbf{H}_\infty(X) = \mathbf{H}_1(X) = 128$. However when X is not uniform, the two notions give different values. Let us illustrate this with one example. Consider the discrete random variable X defined on $\{0,1\}^{128}$, where $\Pr[X = 0] = 2^{-7}$ and $\Pr[X = y, y \neq 0] = \frac{1-2^{-7}}{2^{128}-1}$. Then $\mathbf{H}_1(X) = 127,006$ and $\mathbf{H}_\infty(X) = 7$.

Hence the two notions of entropy describe a different deviation to the uniform distribution: the estimated Shannon entropy $\mathbf{H}_1(X)$ is close to 128, whereas its min-entropy $\mathbf{H}_\infty(X)$ is on the opposite very low and is such that $\Pr[X = 0] = 2^{-\mathbf{H}_\infty(X)}$, setting a direct relation between $\mathbf{H}_\infty(X)$ and the set of non-uniformity.

Let us now illustrate why the Shannon Entropy can not be used for cryptographic purposes. Suppose now that we use directly the source X to generate a 128-bits encryption key for a symmetric algorithm (AES for example). Recall that an encryption scheme is a triple $(\mathsf{key}, \mathsf{enc}, \mathsf{dec})$, where key is a probabilistic algorithm for key generation, enc is the encryption algorithm and dec is the decryption algorithm. For each key K sampled by key and for all x, we have that $\mathsf{enc}(\mathsf{K}, \mathsf{dec}(\mathsf{K}, x)) = x$. Consider the (simple) security game ENC described in Figure 2.1, where the key sampling algorithm key is the algorithm that samples a key of distribution X. Consider

proc. initialize()	**proc**. enc-ror(m_0, m_1)
$\mathsf{K} \xleftarrow{\$} X$;	$c_0 \leftarrow \mathsf{enc}(\mathsf{K}, m_0)$
$b \xleftarrow{\$} \{0,1\}$	$c_1 \leftarrow \mathsf{enc}(\mathsf{K}, m_1)$
	OUTPUT c_b
proc. finalize(b^*)	
IF $b = b^*$ RETURN 1	
ELSE RETURN 0	

Figure 2.1 – Procedures in Security Game ENC

now an adversary $\mathcal{A}$ in game ENC. As the key K is sampled from the random variable X, with probability 2^{-7}, we have that $\mathsf{K} = 0$. Hence with probability 2^{-7}, $\mathcal{A}$ can distinguish between

$\mathsf{enc}(\mathsf{K}, m_0)$ and $\mathsf{enc}(\mathsf{K}, m_1)$, for all m_0 and m_1 and for all encryption scheme $(\mathsf{key}, \mathsf{enc}, \mathsf{dec})$. However, it is expected that the outputs c_0 and c_1 are distinguishable with probability 2^{-128}. If one considers Shannon entropy in place of min-entropy, one could have argued that the probability of distinguishing between c_0 and c_1 is close to 128, which is wrong. Hence there is a direct relation between the min-entropy of a random variable and the advantage for an adversary in distinguishing between two computations in a security game where the random variable is used as a source of randomness. In other words, $\mathbf{H}_1$ measures the amount of randomness that a source contains *on average*, (as justified with the use of the expectation $\mathbb{E}$) while $\mathbf{H}_\infty$ measures the amount of randomness on the *worst-case*, which are typically the cases that an adversary will use to break a security scheme. This justifies the notion of k-sources (or distributions with min-entropy at least k) as the formalization of the notion of sources 'containing k bits of randomness'.

Definition 2 (k-source)**.** *A source X is a k-source if $\mathbf{H}_\infty(X) \geq k$.*

2.6 Randomness Extractors

Randomness is concretely generated from sources which are potentially biased, where the only known information is that they potentially contain some amount of randomness, or, as formalized in Definition 2, they are k-sources. We therefore need a map that extracts the randomness that is actually contained in these sources, and produces an output which is close to uniform. These maps are named *extractors*.

Let first illustrate this idea with the two following examples:

- **Extractor for Independent Sources**. Consider a sequence of *independent* sources of bits $X_i \in \{0,1\}$ where for all $i, \Pr[X_i = 1] = \delta$ (i.e., all sources are *biased* with the same bias). Consider the following map $\mathsf{Extract} : \{0,1\} \times \{0,1\} \to \{0,1\} \cup \{\varnothing\}$, where: $\mathsf{Extract}(0,0) = \mathsf{Extract}(1,1) = \varnothing, \mathsf{Extract}(1,0) = 1$ and $\mathsf{Extract}(0,1) = 0$. Then the output of $\mathsf{Extract}$ is uniformly distributed. This map is known as the 'Von Neumann extractor', as described in [VN51].

- **Extractor for Independent-Bit Sources**. Consider a sequence of *independent* sources of bits $X_i \in \{0,1\}$, where for all $i, \Pr[X_i = 1] = \delta_i, \Pr[X_i = 0] = 1 - \delta_i, 0 < \delta \leq \delta_i \leq 1 - \delta$ (i.e., all sources are *biased* with different bounded biases). Consider the following function $\mathsf{Extract} : \{0,1\}^p \to \{0,1\}$, where $\mathsf{Extract}(x_1, \cdots, x_p) = x_1 \oplus \cdots \oplus x_p$. Then if p sources are used, each bit output by $\mathsf{Extract}$ has bias in the interval $[\frac{1}{2} - (1-2\delta)^p, \frac{1}{2} + (1-2\delta)^p]$, hence:
$$|\Pr[y_i = 0 | y_1, \cdots, y_{i-1}] - \Pr[y_i = 1 | y_1, \cdots, y_{i-1}]| < (1-2\delta)^p.$$
Therefore the outputs of $\mathsf{Extract}$ are indistinguishable for large p. This function is described by Santha and Vazirani in [SV84] and is also referred to as the 'parity extractor'.

The previous 'parity extractor' is an example of *deterministic extractors*, as defined by Nisan and Zuckerman in [NZ93]. A deterministic extractor is formalized in Definition 3. Note that formally, the 'Von Neumann extractor' is not an extractor, as the definition supposes that an output is generated for any input (the output $\varnothing$ is not possible).

Definition 3 (Deterministic Extractors)**.** *Let p and m be integers, such that $p \geq m$. Let $\mathcal{C}$ be a class of sources on $\{0,1\}^p$. An ε-deterministic extractor for $\mathcal{C}$ is a function $\mathsf{Extract} : \{0,1\}^p \to \{0,1\}^m$, such that for every $X \in \mathcal{C}$, $\mathsf{Extract}(X)$ and $\mathcal{U}_m$ are ε-close.*

Note that Definition 3 requires that the function $\mathsf{Extract}$ works for all the sources X that belong to the class $\mathcal{C}$. Therefore, if one wants to extract randomness from sources, these distributions

do not need to be known: for example, in case of the 'parity extractor' presented before, the bias δ do not need to be known. Moreover the link between min-entropy and extraction comes directly from Definition 3, as a necessary condition to extract m bits of randomness from a distribution X is that $\mathbf{H}_\infty(X) \geq m$.

Let us now describe another class of sources, named *Santha-Vazirani* sources, or *δ-Unpredictable-bit* sources. These sources are also described in [SV84]. Consider the sequence of sources of bits $X_i \in \{0,1\}$:

$$\frac{1-\delta}{2} \leq \Pr[X_i = 0 | X_1 = x_1, X_2 = x_2, \cdots, X_{i-1} = x_{i-1}] \leq \frac{1+\delta}{2}, \text{ where } x_i \in \{0,1\}.$$

Note that this class of sources is similar to the class of Independent-Bit Sources, as for all sources X_i, the bias is also bounded between δ and $(1-\delta)$, however, in this class of sources, the independence between the sources is not required. Lemma 1 shows that the deterministic extraction of more than a single bit is impossible for this class of sources.

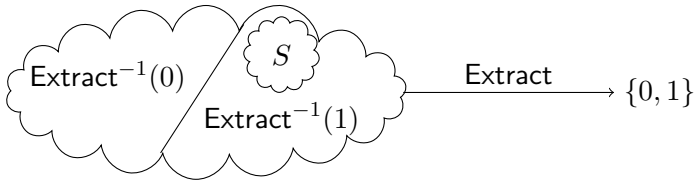

Figure 2.2 – Impossibility of Deterministic Extraction for δ-Unpredictable-bit sources

Lemma 1. *For every $p \in \mathbb{N}, \delta > 0$ and every map $\mathsf{Extract} : \{0,1\}^p \to \{0,1\}$, there exists a δ-Unpredictable-bit source X such that $\Pr[\mathsf{Extract}(X) = \mathsf{b}] \geq \frac{1}{2}(1+\delta)$, for at least one $\mathsf{b} \in \{0,1\}$.*

Proof. Partition the set $\{0,1\}^p$ between $\mathsf{Extract}^{-1}(0)$ and $\mathsf{Extract}^{-1}(1)$. Then at least for one $\mathsf{b} \in \{0,1\}$ we have that $|\mathsf{Extract}^{-1}(\mathsf{b})| \geq 2^{p-1}$. As illustrated in Figure 2.2, consider a subset $S \subseteq \mathsf{Extract}^{-1}(\mathsf{b})$, of size 2^{p-1} and the source X such that $\forall x \in \{0,1\}^p, \Pr[X = x] = \frac{1+\delta}{2^p}$ if $x \in S$ and $\Pr[X = x] = \frac{1-\delta}{2^p}$ if $x \notin S$. Then X is a $\delta-$Unpredictable-bit Source and we have that:

$$\Pr[\mathsf{Extract}(X) = \mathsf{b}] \geq \Pr[X \in S] = \frac{1}{2}(1+\delta).$$

□

As a consequence of the previous examples, one objective of research on deterministic extractors is to identify the richest classes of randomness sources for which deterministic extraction is possible, and construct explicit extractors for those sources.

In this thesis, we do not rely on the potential results from this line of research, as we want to build schemes that do not depend on the structure of the randomness source. As described before, the notion of k-source comes naturally as it is the most general way to formalize that a source contains k bits of randomness. Therefore, we will assume in the following that all sources of randomness are k-sources.

Unfortunately, Lemma 2 below shows that the deterministic extraction of even a single bit is impossible for this class of sources and motivates the use of a more elaborate notion of extractor that uses a second source of randomness called seed.

Lemma 2. *For every map $\mathsf{Extract} : \{0,1\}^p \to \{0,1\}$, there exists a $(p-1)$-source X such that $\mathsf{Extract}(X)$ is constant.*

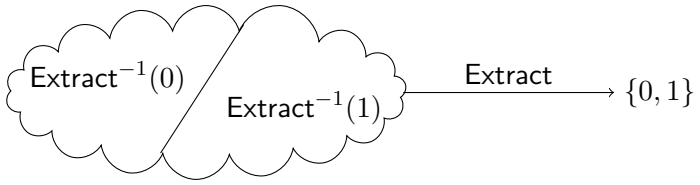

Figure 2.3 – Impossibility of Deterministic Extraction for k-sources

Proof. Partition the set $\{0,1\}^p$ between $\mathsf{Extract}^{-1}(0)$ and $\mathsf{Extract}^{-1}(1)$, as illustrated in Figure 2.3. Then at least for one $\mathsf{b} \in \{0,1\}$ we have that $|\mathsf{Extract}^{-1}(\mathsf{b})| \geq 2^{p-1}$. Define the source X as the uniform distribution on $\mathsf{Extract}^{-1}(\mathsf{b})$, where $|\mathsf{Extract}^{-1}(\mathsf{b})| \geq 2^{p-1}$. Then X is a $(p-1)$-source and $\mathsf{Extract}(X) = \mathsf{b}$ is constant. □

Consider now that the function $\mathsf{Extract}$ is chosen randomly from the set of all functions from $\{0,1\}^p$ to $\{0,1\}$ and suppose that one wants to exhibit a similar source than the one from the previous Lemma. Recall that the set of all functions from $\{0,1\}^p$ to $\{0,1\}$ is of size 2^{2^p}, therefore we can consider that a random choice for the function $\mathsf{Extract}$ is done in the set $\{\mathsf{Extract}_i\}_{i=1\cdots 2^{2^p}}$. Suppose that one defines again the source X as the uniform distribution on $\mathsf{Extract}_i^{-1}(\mathsf{b})$, where $|\mathsf{Extract}_i^{-1}(\mathsf{b})| \geq 2^{p-1}$, for a randomly chosen $i \in \{1 \cdots 2^{2^p}\}$, then as before, X is a $(p-1)$-source. However, for $j \neq i$, $\mathsf{Extract}_j(X)$ is balanced with high probability, as illustrated in Figure 2.4. Therefore one cannot construct a $(p-1)$-source as in Lemma 2. The formal statement is given in the proof of Theorem 3 below based on the probabilistic method.

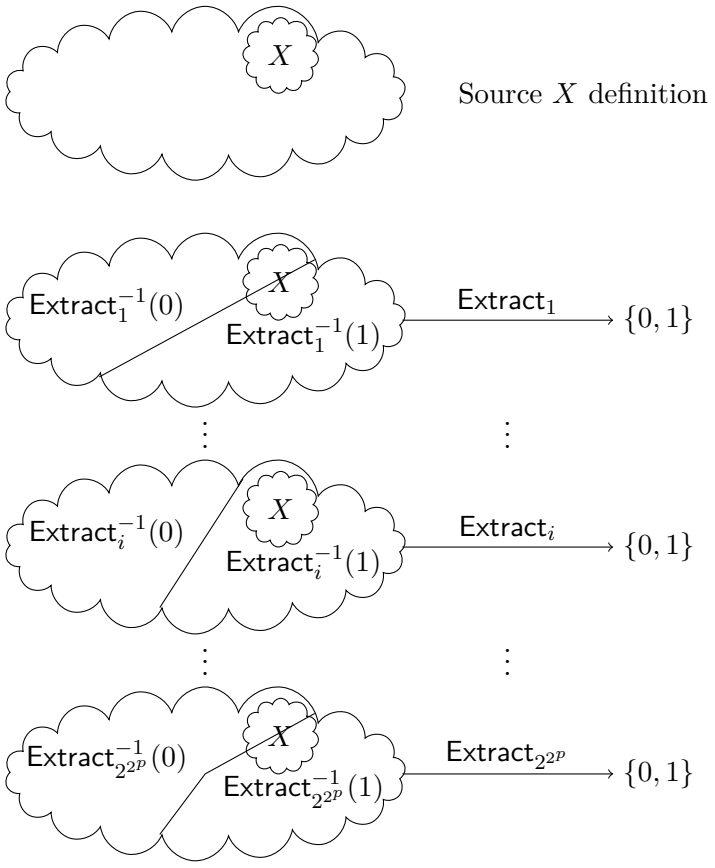

Figure 2.4 – Randomly Chosen Function $\mathsf{Extract}$

This illustrates that the impossibility result of Lemma 2 can be overcome with the probabilistic method: if we allow to choose the extraction function *at random*, then, with a high probability, it will become possible to extract the randomness from *each* k-source. To choose the extractor at random, we will assume that it belongs to a *family* of functions (which can be the family of all functions from $\{0,1\}^p$ to $\{0,1\}^m$) and we *uniformly* select a random element from this family. The selection process implies choosing a random parameter called $\mathsf{seed} \in \{0,1\}^s$ and setting the extraction function as $\mathsf{Extract}_{\mathsf{seed}} = \mathsf{Extract}(.,\mathsf{seed})$.

This discussion leads to the notion of *seeded extractor*, as in Definition 4.

Definition 4 (Seeded Extractors)**.** *A function* $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^m$ *is a* (k,ε)-seeded extractor *if for all* k*-sources* X*, the distributions* $\mathsf{Extract}(X,\mathsf{seed})$ *and* $\mathcal{U}_m$ *are* ε*-close, where* $\mathsf{seed} \xleftarrow{\$} \{0,1\}^s$ *is chosen independently of* X*.*

Hence the difference between *deterministic* and *seeded* extractors relies on the use of the supplementary random parameter $\mathsf{seed} \xleftarrow{\$} \{0,1\}^s$. Moreover, the above definition means that the extraction works for all k-sources. In addition, we can now prove that (k,ε)-extractors exist, with Theorem 3 below. The proof of Theorem 3 uses the Chernoff bound:

Proposition 1 (Chernoff Bound [Sho06])**.** *Let* $Z_1, Z_2, \cdots, Z_n$ *be independent random variables such that* $0 \leq Z_i \leq 1, \forall i$*. Let* $Z = \sum_i Z_i$ *and* $\mu = \mathbb{E}[Z] = \sum_i \mathbb{E}[Z_i]$*. Then for all* $\varepsilon > 0$*,*

$$\Pr[|Z - \mu| \geq \varepsilon\mu] \leq 2\exp(-\frac{\varepsilon^2}{3}\mu).$$

Theorem 3. *For every* $p \in \mathbb{N}$*,* $k \in [0, \cdots, p]$*, there exists a* (k,ε)*-extractor* $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^m$*, with* $m = k + \log(p-k)$ *and* $s = \log(p-k) + 2\log(1/\varepsilon) + O(1)$*.*

Proof. The proof uses the probabilistic method. We give the proof for flat k-sources (that is, with uniform distribution over a subset of $\{0,1\}^p$ of size 2^k), the proof extends to general k-sources (see [Vad12]). Consider (a) a randomly chosen function $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^m$ (b) a *flat* k-source X and (c) T a subset of $\{0,1\}^m$.

Then as X is a flat k-source, the random variable (X,seed) is a flat $(k+s)$-source.

Consider the (indicator) random variables $Z_{x,y} = 1_{\mathsf{Extract}(x,y)\in T}$ and the random variable:

$$Z = \sum_{(x,y)\in supp(X,\mathsf{seed})} Z_{x,y} = \sum_{(x,y)\in supp(X,\mathsf{seed})} 1_{\mathsf{Extract}(x,y)\in T}$$

Then as $\mathsf{Extract}$ is chosen randomly and seed is sampled independently of X, the random variables $Z_{x,y}$ are independent and $\mathbb{E}(Z_{x,y}) = \frac{|T|}{2^m}$ and as (X,seed) is uniform on its support, $\Pr[X = x, \mathsf{seed} = s] = \frac{1}{2^{k+s}}$ in the support only and therefore $\mathbb{E}(Z) = \frac{2^{k+s}|T|}{2^m}$. We can apply Proposition 1 to the random variables $Z_{x,y}$:

$$\Pr(|Z - \frac{2^{k+s}|T|}{2^m}| \geq \varepsilon \cdot \frac{2^{k+s}|T|}{2^m}) \leq 2\exp(-\frac{\varepsilon^2}{3}\frac{2^{k+s}|T|}{2^m}),$$

which implies, with $\varepsilon = \varepsilon' \frac{2^m}{|T|}$:

$$\Pr(|\frac{Z}{2^{k+s}} - \frac{|T|}{2^m}| \geq \varepsilon') \leq 2\exp(-\frac{\varepsilon'^2}{3}\frac{2^{k+s}2^m}{|T|}).$$

Then as $\Pr[\mathcal{U}_m \in T] = \frac{|T|}{2^m}$, the last inequality shows that:

$$\Pr(|\Pr[\mathsf{Extract}(X,\mathsf{seed}) \in T] - \Pr[\mathcal{U}_m \in T]| \geq \varepsilon') \leq 2\exp(-\frac{\varepsilon'^2}{3}\frac{2^{k+s}2^m}{|T|}),$$

Then as $\frac{2^m}{|T|} \geq 1$, we have that:

$$\Pr(|\Pr[\mathsf{Extract}(X,\mathsf{seed}) \in T] - \Pr[\mathcal{U}_m \in T]| \geq \varepsilon') \leq 2\exp(-\frac{\varepsilon'^2 2^{k+s}}{3}).$$

There are 2^{2^m} possible sets $T \in \{0,1\}^m$ and[1] $\binom{2^p}{2^k} \leq \left(\frac{2^p e}{2^k}\right)^{2^k}$ flat k-sources in $\{0,1\}^p$. By the union bound over all possible sets and all possible flat k-sources, the probability that $\mathsf{Extract}$ is a (k,ε)-extractor for all flat k-sources satisfies:

$$\Pr(\max_{T\subseteq\{0,1\}^m} |\Pr[\mathsf{Extract}(X,\mathsf{seed}) \in T] - \Pr[\mathcal{U}_m \in T]| \leq \varepsilon') \leq 2^{2^m}\left(\frac{2^p e}{2^k}\right)^{2^k} 2\exp(-\frac{\varepsilon'^2 2^k}{3}),$$

Then $2^{2^m}\left(\frac{2^p e}{2^k}\right)^{2^k} 2\exp(-\frac{\varepsilon'^2 2^{k+s}}{3}) < 1$ as soon as $2^m + 1 + 2^k(p-k+\log(e)) < \frac{\varepsilon'^2 2^{k+s}}{3}$, which is satisfied if (a) $6\cdot(2^m+1) < \varepsilon'^2 2^{k+s}$ and (b) if $6\cdot 2^k(p-k+\log(e)) < \varepsilon'^2 2^{k+s}$, that are satisfied if (a) $m = k+s-2\log(\frac{1}{\varepsilon'}) - \log(12) - 1$ and (b) $s = \log(p-k) + 2\log(\frac{1}{\varepsilon'}) + \log(12) + 1$. □

Doing this, we can consider that the extraction is defined over the product set $\{0,1\}^p \times \{0,1\}^s$, where $\{0,1\}^p$ will be the set from which randomness will be extracted and $\{0,1\}^s$ will be the set from which the parameter seed will be chosen. Hence the new objective is to analyze the statistical distance of the distribution $\mathsf{Extract}(X,\mathsf{seed})$ and the uniform distribution, where X is a k-source ($\mathbf{H}_\infty(X) \geq k$) and $\mathsf{seed} \xleftarrow{\$} \{0,1\}^s$.

The use of a second random parameter is not sufficient to guarantee that the extraction is possible for any source. It is indeed straightforward to see that there is a new impossibility result (Lemma 3 below) when the source of parameter seed and the randomness source are not independent. The proof of Lemma 3 is the same as the proof of Lemma 2.

Lemma 3. *For every map* $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^m$ *and every* $\mathsf{seed} \in \{0,1\}^s$*, there exists a* $(p-1)$*-source* X *(depending on* seed*) such that* $\mathsf{Extract}(X,\mathsf{seed})$ *is constant.*

Lemma 3 shows that we face two issues: (a) the generation of the uniformly random parameter seed and (b) the potential correlation between seed and the source from which we will try to extract randomness. Hence in an adversarial viewpoint, we need to consider situations where the seed or the environment may be controlled by an adversary, and situations where a potential correlation between the randomness source and seed may be exploited to mount an attack against the scheme. This may occur for example in a hardware device, that extracts from physical sources of randomness of a computer (e.g. timing of various events). These sources may be modified by the device and hence this behavior implies correlations between seed and the randomness sources. Therefore, we need to add optional requirements, either on the independence between the source and seed, or on the capabilities of the adversary.

Suppose now that the independence between the source and seed cannot be ensured and we want to model situations where we need to perform randomness extraction. As noted before, to overcome the impossibility result, we mainly have two options: (a) restrict the randomness source to a given family of k-sources (which is a similar strategy as for deterministic extractors) or (b) restrict the adversary $\mathcal{A}$.

We first propose to limit the extraction to a finite family of k-source for which we are sure that extraction is possible. This leads to the notion *resilient* extractor, as in Definition 5.

[1] For $n,m \in \mathbb{N}$, such that $2 \leq m \leq n$, $\binom{n}{m} \leq \left(\frac{ne}{m}\right)^m$

Definition 5 (Resilient Extractor)**.** *A function* $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^m$ *is a* (k, ε, δ)-resilient extractor *if for all finite families of* k*-sources* $\mathcal{F}$*, with probability at least* $(1-\delta)$ *over the choice of* $\mathsf{seed} \xleftarrow{\$} \{0,1\}^s$*, the distributions* $(\mathsf{seed}, \mathsf{Extract}(X, \mathsf{seed}))$ *and* $(\mathsf{seed}, \mathcal{U}_m)$ *are* ε*-close, for all* $X \in \mathcal{F}$*.*

Note that to simplify the number of parameters, one can set $\varepsilon = \delta$ in the above definition, in this case, we refer to (k, ε)-resilient extractors.

Resilient extractors stand for (a) bounded family of randomness source and (b) correlated seed and source. Definition 5 can be expressed in terms of hash functions: Let $\mathcal{H} = \{h : \{0,1\}^p \to \{0,1\}^m\}$ be a family of hash functions. Then $\mathcal{H}$ is a (k, ε)-*resilient extractor* if for any random variable I over $\{0,1\}^p$ with $\mathbf{H}_\infty(I) \geq k$, the distributions $h(I)$ and $\mathcal{U}_m$ are ε-close with probability $(1-\delta)$ over the choice of h, where $\mathcal{U}_m$ is uniformly random over $\{0,1\}^m$. An important result is the Leftover Hash Lemma, presented in Section 2.7, that constructively leads resilient extractors from *pairwise* independent families of hash functions.

This notion of extractor is used in the model of Barak, Shaltiel and Tromer [BST03], described in Section 3.5 and in the model of Barak and Halevi [BH05], described in Section 3.6. In these models, a finite family of k-sources is first chosen, then the random parameter seed is chosen and finally a source is adversarially chosen (and therefore *without* independence with seed).

Suppose now that we do not want to limit the extraction to a finite family of sources. Definition 6, which generalizes Definition 5, describes the objectives for a randomness extractor.

Definition 6 (Seed Dependent Extractor)**.** *A function* $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^m$ *is a seed-dependent* (k, ε)*-extractor if for all probabilistic adversaries* $\mathcal{A}$ *who take as input a random seed* $\mathsf{seed} \xleftarrow{\$} \{0,1\}^s$ *and output* $X \leftarrow \mathcal{A}(\mathsf{seed})$ *of entropy* $\mathbf{H}_\infty(X|\mathsf{seed}) \geq k$*, the distributions* $(\mathsf{seed}, \mathsf{Extract}(X, \mathsf{seed}))$ *and* $(\mathsf{seed}, \mathcal{U}_m)$ *are* ε*-close.*

One way to restrict adversary $\mathcal{A}$ is to force its running time to be less than the running time of the extractor $\mathsf{Extract}$. This idea was formalized by Trevisan and Vadhan in [TV00]. In this work, they show how seed-dependent randomness extraction is possible from a *samplable* distribution, provided that the complexity of the extractor is larger than the complexity of the adversary $\mathcal{A}$ that generates the source X. In particular, they show that if the adversary's running time is larger than the extractor's one by a factor of t, it can fix roughly $\log(t)$ bits of the output. Note that this result motivates the introduction of *randomness condensers*, as described in [DRV12]. In this work, we do not consider randomness condensers but focus on randomness extractors, as we will want that the output of the extraction phase is ε-close to uniform, to apply a standard pseudo-random number generator $\mathbf{G}$ after extraction. Hence to consider adversarial situations where seed and the source may be correlated, without restriction on the randomness source, it will be necessary to restrict the adversary $\mathcal{A}$ where its running time is less than the running time of the extractor. To conclude, seed-dependent extractors stand for (a) unbounded family of randomness source, (b) correlated seed and source and (c) limited adversary.

Suppose now that, as opposed to *seeded* extractors, we do not want to restrict the running time of the adversary $\mathcal{A}$. As pointed, we need to ensure that independence between the source and the seed can be ensured. In addition, we want to use the 'extra' randomness seed in Definition 4 as less as possible. This leads to the notion of *strong extractors*, given in Definition 7, where the randomness seed is *maintained* by the extractor, and therefore (a) it can be *reused* through successive calls to $\mathsf{Extract}$ and (b) it can be made public.

Definition 7 (Strong Extractors.)**.** *A function* $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^m$ *is a* strong (k, ε)-extractor *if for all probabilistic adversaries* $\mathcal{A}$ *who sample a distribution* X *of entropy*

$\mathbf{H}_\infty(X) \geq k$, *the distributions* $(\mathsf{seed}, \mathsf{Extract}(X, \mathsf{seed}))$ *and* $(\mathsf{seed}, \mathcal{U}_m)$ *are* ε*-close, where* $\mathsf{seed} \leftarrow \{0,1\}^s$ *and* X *is independent of* seed.

This definition ensures that once random parameter seed is chosen, extraction is processed and the same parameter can be reused for the next extraction.

Definition 7 can be expressed in terms of universal hash functions: The hash functions family $\mathcal{H}$ is a (k, ε)-*extractor* if for any random variable I over $\{0,1\}^p$ with $\mathbf{H}_\infty(I) \geq k$, the distributions $(\mathsf{seed}, h_{\mathsf{seed}}(I))$ and $(\mathsf{seed}, \mathcal{U}_m)$ are ε-close where seed is uniformly random over $\{0,1\}^s$. The Leftover Hash Lemma (Lemma 4) constructively builds a strong extractor from a universal hash functions family.

We summarize the different notions of extractors seen in this section with Table 2.1. The five extractor types are given (deterministic, resilient, seed-dependent, seeded, strong) and for each type, we precise:

1. If the parameter seed shall be independent from the randomness source (which we denote with symbol ✖) or can be correlated to it (which we denote with symbol ✓).
2. If the parameter seed shall remain secret (which we denote with symbol ✖) or can be made public (which we denote with symbol ✓).
3. If existence of an extractor of a given type implies either restriction on the number of randomness source or a restriction on the capacities of the adversary $\mathcal{A}$ (which we denote with symbol ✖) or no restriction is needed (which we denote with symbol ✓).

As shown in Table 2.1, a consequence of Lemma 3 is the impossibility to build an randomness extractor that posseses 'all' the properties: for which (a) independence between seed and the randomness source is not required, (b) secrecy of seed is not required, (c) a restriction on the number of randomness source or a restriction on the capacities of the adversary $\mathcal{A}$ shall be enforced. This table shows that the use of a randomness extractor in the design of security

Table 2.1 – Tradeoff for Randomness Extractors

Extractor Type	seed		**Number of Sources**	**Attacker Capacities**
	Correlation	**Secrecy**		
Deterministic	∅		✖	✓
Resilient	✓	✓	✖	✓
Seed-dependent	✓	✓	✓	✖
Seeded	✖	✖	✓	✓
Strong	✖	✓	✓	✓
Impossible (Lemma 3)	✓	✓	✓	✓

∅: no parameter seed is used, ✓: possible correlation, public seed / no restriction on the source or on $\mathcal{A}$, ✖: independence required, private seed, restrictions on the source or on $\mathcal{A}$.

scheme shall be done with care as any choice seems to have a drawback.

2.7 Leftover Hash Lemma

We present two versions of the Leftover Hash Lemma. The first one constructively builds strong extractors from universal hash functions families and the second one builds resilient extractors from pairwise independant hash functions families. This important Lemma was first formally stated in [HILL99].

Lemma 4 (Leftover-Hash Lemma for Universal Hash Functions Family Family)**.** *Assume that the hash functions family $\mathcal{H} = \{h : \{0,1\}^p \to \{0,1\}^m\}$ is ρ-universal where $\rho = (1+\alpha)2^{-m}$ for some $\alpha > 0$. Then, for any $k > 0$, it is also a strong (k,ε)-extractor for $\varepsilon = \frac{1}{2}\sqrt{2^{m-k} + \alpha}$.*

Proof. We recall the proof described in [Vad12]. 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 $U \xleftarrow{\$} \{0,1\}^m$. First consider the statistical distance between $(X, h_X(I))$ and $(X, \mathcal{U}_m)$. As in [Vad12], 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: $\mathbf{CP}(X) = \sum_x \Pr[X=x]^2$.

Then we can bound the statistical distance between $(X, h_X(I))$ and $(X, \mathcal{U}_m)$ by their Δ_2 distance:

$$\mathbf{SD}((X,h_X(I)),(X,\mathcal{U}_m)) \;\leq\; \frac{1}{2}\sqrt{2^s \cdot 2^m} \cdot \Delta_2((X,h_X(I)),(X,\mathcal{U}_m)),$$

and we have $\Delta_2((X,h_X(I))^2 = \Delta_2((X,h_X(I)),(X,\mathcal{U}_m))^2 + 2^{-m-s}$.

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 $2^{-m} \cdot (1+\alpha)$-universal:

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

Finally, with $\alpha = 4 \cdot \varepsilon^2 - 2^{m-k}$:

$$\begin{aligned}\mathbf{SD}((X,h_X(I)),(X,U)) &\leq \frac{1}{2} \cdot \sqrt{2^s \cdot 2^m} \cdot \sqrt{\frac{4\cdot\varepsilon^2}{2^s \cdot 2^m}} \\ &\leq \varepsilon\end{aligned}$$

Following, the hash functions family $\mathcal{H} = \{h_X : \{0,1\}^p \to \{0,1\}^m\}_{X \in \{0,1\}^s}$, is a (k,ε)-strong extractor for $\varepsilon = \frac{1}{2}\sqrt{2^{m-k} + \alpha}$. □

As Lemma 4 shows, it is possible to construct *strong* extractors from universal hash functions family. This results motivates the use of such functions to build security schemes that rely on randomness. Note that in the proof of Lemma 4, the independence between the samples I, I' and seed is used to estimate the collision probability of the joint distribution $(X, h_X(I))$: $\mathbf{CP}(X, h_X(I)) = \Delta_2((X,h_X(I))^2$. In situations where the independence between the samples I, I' and seed can not be ensured, we can prove an alternative version of Lemma 4 with a stronger requirement: we require that the hash functions family is pairwise independent, to obtain a resilient extractor. The version of the Leftover-Hash Lemma for pairwise independent hash functions family can be stated similarly as Lemma 4.

2.8 Pseudo-Random Number Generators

2.8.1 Standard Pseudo-Random Number Generator

A secure pseudo-random number generator is an extending function, that on input a random bit string S (named a *seed*), outputs a longer bit string which is indistinguishable from random. Note that the notion of *seed* shall not be confused with the notion of seed explained in the previous section for randomness extractors. Here the parameter *seed* models a secret, random input of the pseudo-random number generator.

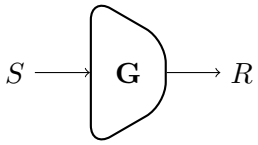

Figure 2.5 – Standard Pseudo-Random Number Generator

Definition 8 (Standard Pseudo-Random Number Generator)**.** *Let p and ℓ be integers such that $p < \ell$. A standard pseudo-random number generator is a function $\mathbf{G} : \{0,1\}^p \to \{0,1\}^\ell$, that takes as input a bit string S (called a seed), of length p and outputs bit string R, of length ℓ bits.*

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

Figure 2.6 – Procedures in Security Game PR

Consider the security game PR described in Figure 2.6. In this security game, the challenger generates a random input S and challenges the adversary $\mathcal{A}$ on its capacity to distinguish the real output of the pseudo-random number generator from random. Definition 9 formalizes security for a standard pseudo-random number generator.

Definition 9 (Security of a Standard Pseudo-Random Number Generator)**.** *A standard pseudo-random number generator is (t, ε)-secure if for any adversary $\mathcal{A}$ running in time at most t, the advantage of $\mathcal{A}$ in game* PR *is at most ε.*

2.8.2 Stateful Pseudo-Random Number Generator

A stateful pseudo-random number generator is an iterative and stateful algorithm, that at each invocation produces some output bits as a function of the current seed and updates the seed. The associated security property generalizes the security of a standard pseudo-random number generator, as the adversary is challenged *after several iterations of the generator* on its capability to distinguish the output of the generator from random, whereas in Definition 8, only one iteration is considered.

Definition 10 (Stateful Pseudo-Random Number Generator)**.** *A stateful pseudo-random number generator is a couple of algorithms* (key, next)*, where* key *is a probabilistic algorithm that takes no input and outputs an initial state $S \in \{0,1\}^p$,* next *is a deterministic algorithm that, given the current state S, outputs a pair $(S', R) \leftarrow \mathsf{next}(S)$ where S' is the new state and $R \in \{0,1\}^\ell$ is the output.*

The security game SPR uses procedures described in Figure 2.7. The procedure initialize sets the first internal state S with a call to algorithm key and sets the random parameter b. Procedure next-ror challenges $\mathcal{A}$ on its capability to distinguish the output of the stateful pseudo-random number generator from random, where the real output (R_0) of the stateful pseudo-random number generator is obtained with a call to algorithm next and the random string (R_1) is picked uniformly at random by the challenger. Attacker $\mathcal{A}$ responds to the challenge with a bit b^*. 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.

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

Figure 2.7 – Procedures in Security Game SPR

Definition 11 (Security of a Stateful Pseudo-Random Number Generator)**.** *A stateful pseudo-random number generator* $\mathcal{G} = (\mathsf{key}, \mathsf{next})$ *is* (t, ε)*-secure, if for any attacker* $\mathcal{A}$ *running in time at most* t*, the advantage of* $\mathcal{A}$ *in game* SPR *is at most* ε*.*

Bellare and Yee [BY03] proposed a new definition of a stateful pseudo-random number generator, where the number of outputs the generator is allowed to produce is a parameter of the generator. This definition is described inChapter 3.

2.8.3 Pseudo-Random Number Generator with Input

Consider now an iteration of the pseudo-random number generator where, at each iteration, we let the pseudo-random number generator process a different auxiliary input, in addition to the key. This leads to the notion of pseudo-random number generator with input. Informally, a pseudo-random number generator with input mixes two different processes: the collection of new inputs and the generation of the output. This idea is illustrated in Figure 2.8.

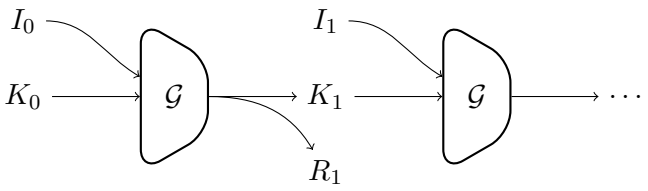

Figure 2.8 – Pseudo-Random Number Generator with Input

Hence, pseudo-random number generators with input model situations where new inputs are continuously used by the generator. We give in the chapter 3 different definitions for pseudo-random number generators with input, that we present briefly here:

- In 1998, in two different works, Gutmann (in [Gut98]), and Kelsey, Schneier, Wagner and Hall, (in [KSWH98]) gave useful guidelines for the design of a secure pseudo-random number generator with input.
- In 2002, in [DHY02], Desai, Hevia and Yin modelled a pseudo-random number generator with input as an iterative algorithm, which in each iteration take three inputs: a key K, an internal state S, and an auxiliary input I. The algorithm generates two outputs: pseudo-random numbers R and a new state S'.
- In 2003, in [BST03], Barak, Shaltiel and Tromer proposed a security model for a pseudo-random number generator with input where an attacker can have some control on the

inputs. As we explain, their definition of pseudo-random number generators with input is based on the use of resilient extractors.

- In 2005, in [BH05] Barak and Halevi proposed a security model in which a pseudo-random number generator with input is clearly defined as a couple of deterministic polynomial-time algorithms $\mathcal{G} = (\mathsf{refresh}, \mathsf{next})$, where the first algorithm $\mathsf{refresh}$ models the update of the internal state S with an input I containing randomness ($S \leftarrow \mathsf{refresh}(S, I)$) and the second algorithm next models the output R generation and the update of the internal state S during this generation ($(S, R) \leftarrow \mathsf{next}(S)$). As before, their definition of pseudo-random number generators with input is based on the use of resilient extractors.

In chapters 4, 5 and 6, we present the main contributions of this thesis, which are (a) a new definition of pseudo-random number generator with input based on strong extractors and (b) the formal statement of security properties for pseudo-random number generators with input.

2.9 Pseudo-Random Functions

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. Note that the term *random function* means *function chosen at random.*

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 $2^{k.2^k}$, hence to describe a (random) function from this set, we would need $k.2^k$ bits, which is impractical. Suppose now that we select a set of cardinal 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 12.

Definition 12 (Keyed Family of Functions)**.** *A keyed family of functions is a map* $\mathbf{F} : \{0,1\}^s \times \{0,1\}^\ell \rightarrow \{0,1\}^L$*, where (a)* $\{0,1\}^s$ *is the key space of* $\mathbf{F}$ *and* s *is the key length (b)* $\{0,1\}^\ell$ *is the domain of* $\mathbf{F}$ *and* ℓ *is the input length and (c)* $\{0,1\}^L$ *is the range of* $\mathbf{F}$ *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 an adversary that is given the key, and is computationally bounded, cannot distinguish the input-output behavior of the function from a random function. This property is formalized with the security game PRF described in Figure 2.9.

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 $\mathsf{funct\text{-}ror}$ with chosen inputs. For each input, the challenger generates a real output with function $\mathbf{F}$ or a random output and challenges $\mathcal{A}$ on its capability to distinguish the output of $\mathbf{F}$ from random. Note that the challenger constructs

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

Figure 2.9 – Procedures in Security Game PRF

a lookup table `funtab` for the random outputs to ensure that the evaluation of equal inputs gives equal outputs: `funtab` is first initialized with $\varnothing$, then at each oracle call, if the value does not exists in the lookup table `funtab`, it is randomly created, otherwise it is directly given as a random output.

Definition 13 (Pseudo-Random Function)**.** *A keyed family of functions* $\mathbf{F} : \{0,1\}^s \times \{0,1\}^\ell \to \{0,1\}^L$ *is a* (t, q, ε)*-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* ε*.*

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 when 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 $\mathbf{F}$ in the funct-ror procedure are not adversarially chosen but are picked at random by the challenger. The procedures are presented in Figure 2.10.

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

Figure 2.10 – Procedures in Security Game WPRF

Definition 14 (Weak Pseudo-Random Function)**.** *A keyed family of functions* $\mathbf{F} : \{0,1\}^s \times \{0,1\}^\ell \to \{0,1\}^L$ *is a* (t, q, ε)*-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* ε*.*

2.10 Pseudo-random Permutations

As explained in Section 2.9, 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 a similar objective than for pseudo-random functions, in the sense that an adversary that is given the key, and is computationally bounded, cannot distinguish the input-output

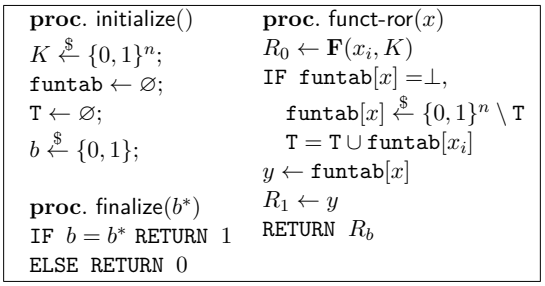

Figure 2.11 – Procedures in Security Game PRP

behavior of the permutation from a random one. This property is formalized with the security game PRP described in Figure 2.11.

Definition 15 (Pseudo-Random Permutation)**.** *A keyed family of permutations* $\mathbf{F} : \{0,1\}^p \times \{0,1\}^n \to \{0,1\}^p$ *is a* (t, q, ε)*-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* ε*.*

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

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

The bound of the previous intuitively comes from the birthday bound, because one way to distinguish between a pseudo-random function and a pseudo-random permutation below is to search for collisions.

Chapter 3

Security Models for Pseudo-random Number Generators

3.1 Introduction

This chapter presents the state of the art security models assessing he security of pseudo-random number generators *before* the introduction of the models from this thesis (see Chapters 4, 5 and 6). We give a syntaxtic formalization for security models that have been proposed. These models consider pseudo-random number generator as a cryptographic primitive that needs to be studied on its own, hence considering dedicated threats and security requirements. For each model, we recall the syntactic definition of pseudo-random number generators that is used and the goal of the adversaries that are considered and their means. We then give a description of the security model and the associated constructions.

Security Guidelines. These guidelines concern pseudo-random number generators with input. In 1998, in two different works, Gutman [Gut98], and Kelsey, Schneier, Wagner and Hall [KSWH98] gave useful guidelines for the design of secure pseudo-random number generators. In these guidelines, they all consider a pseudo-random number generator with input as a couple of algorithms, one to collect inputs and a second one to generate outputs. They considered adversaries that have access to the output of the generator with input and adversaries that can control inputs used to refresh the generator. They proposed guidelines to build pseudo-random number generators. Note however that these properties are not formalized using a game playing framework, but as guidelines that should help security application designers. Therefore they did not insist in giving a formal statement but more in explaining concepts. To allow comparison between these guidelines and the following security models, we propose a formalization of these guidelines in the game playing framework presented in Section 2.4.

Security Against Chosen Input Attack, Chosen State Attack and Known Key Attack. This security model concerns pseudo-random number generator with input. In 2002, Desai, Hevia and Yin [DHY02] modelled a pseudo-random number generator with input as an iterative algorithm, which in each iteration takes three inputs: a key, an internal state, and an auxiliary input. The algorithm generates two outputs: random numbers and a new state. In this model, the adversary $\mathcal{A}$ has different capacities (inputs are allowed to be hidden, known or chosen and the outputs can be hidden or known). This leads to several different attacks, ranging from the attacks in which $\mathcal{A}$ has the highest capacities (where it is allowed to set all the inputs and to compromise all the outputs) to the attacks in which $\mathcal{A}$ does not compromise any input. They proposed constructions secure in their model, that are instantiations of ANSI X9.17 [ANS85] and of FIPS 186 [DSS00], both based on a pseudo-random function $\mathbf{F}$ and they proved their security by reduction to the security of the pseudo-random function .

Forward Security. This security model concerns the standard notion pseudo-random number generators (Definition 10). In 2003, Bellare and Yee [BY03] proposed a security model to assess *Forward Security*, for which a stateful pseudo-random number generator shall be designed so that it is infeasible to recover any information on previous states or previous output blocks from the compromise of the current state. They proposed a construction that is forward secure, based on a secure standard pseudo-random number generator **G**, and they proved its security by reduction to the security of the standard pseudo-random number generator.

τ-Resilience. This security model concerns pseudo-random number generator with input. In 2003, 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 pseudo-random number generator with input and proposes an analysis of the settlement of the public parameter seed which is inherent to this component. They defined the resilience of a pseudo-random generator with input and they proposed a construction secure in their model from universal hash functions, as described in Section 2.3, based on linear maps.

Robustness. This security model concerns pseudo-random number generators with input. In 2005, Barak and Halevi [BH05] proposed a security model in which a pseudo-random number generator with input is clearly defined as a couple of deterministic polynomial-time algorithms $\mathcal{G} = (\mathsf{refresh}, \mathsf{next})$, where the first algorithm $\mathsf{refresh}$ models the update of the internal state S with an input I containing randomness ($S \leftarrow \mathsf{refresh}(S, I)$) and the second algorithm next models the output R generation and the update of the internal state S during this generation ($(S, R) \leftarrow \mathsf{next}(S)$). In their model, they formalized the *robustness* property that is the expected behavior of the pseudo-random number generator with input after an internal state compromise when $\mathcal{A}$ has also control of the input used to refresh the internal state. They proposed a robust construction based on a randomness extractor and a secure standard pseudo-random number generator and they prove its security by reduction to the security of the extractor and the security of the standard pseudo-random number generator.

3.2 Guidelines from [Gut98, KSWH98]

3.2.1 Description

In [Gut98,KSWH98], Gutman and Kelsey, Schneier, Wagner and Hall gave useful guidelines for the design of secure pseudo-random number generators with input. They considered a pseudo-random number generator with input as a couple of algorithms, one to collect randomness from sources and one to generate outputs. The randomness is collected in the *internal state* of the generator, named S hereafter and outputs are generated from S. Note that they do not formalize the properties using a game playing framework, but as guidelines that should help security application designers, therefore they do not insist in giving a formal statement but more in explaining concepts.

They consider the following attacks:

- *Direct Cryptanalytic Attack* (DCA), when adversary $\mathcal{A}$ is directly able to distinguish between generator outputs and random values. In this scenario, adversary $\mathcal{A}$ has only access to the output of the generator.

- *Input-Based Attacks* (IBA), when adversary $\mathcal{A}$ is able to use its knowledge or some control of the inputs I to distinguish between output and random values. They refined this attack in the following three categories: *chosen input*, *replayed input* and *known input*, respectively,

where adversary $\mathcal{A}$ can choose the source of randomness, force the generator to reuse a source of randomness or get access to the source of randomness, respectively.

- *State Compromise Attacks* (SCA), when adversary $\mathcal{A}$ gets access to the internal state S of the generator.

Note that in [Gut98], Gutman considered that a state compromise should be prevented by the environment and therefore it is not considered in the design of the generator. He therefore proposed several security measures that shall be implemented at system level to prevent state compromise.

To respond to DCA, IBA and SCA, they proposed the following guidelines for the design of a secure pseudo-random number generator with input:

- To prevent DCA, a pseudo-random number generator with input should rely on standard primitives to produce outputs.
- To prevent SCA, a pseudo-random number generator with input should (a) ensure the entire state S changes over time, (b) enforce complete renewal of the internal state S and (c) resist backtracking attacks (a state compromise does not give information about past outputs). As a consequence of (b), they considered that the part of the internal state that is used to generate outputs should be *separated* from the entropy pool, the generation state should be changed only when enough entropy has been collected, according to a conservative estimate.
- To prevent IBA, a pseudo-random number generator with input should (a) combine the collected randomness in such a way that an adversary who gets access to the state S but not to the collected randomness, and an adversary who gets the collected randomness but not the state S, are both unable to get information about the next state and (b) take advantage of every bit of entropy in the inputs it receives.

Note that these guidelines have had a strong impact on concrete pseudo-random number generators with input, as for example the Linux generators `dev/random` and `dev/urandom`, were designed with different pools, to collect randomness and to produce outputs, and a dedicated entropy estimator, which controls the transfers between the pools. We give a precise assessment of these two generators in Section 7.2.

3.2.2 Proposed Formalization

We now translate these guidelines in our game playing framework. Note that this formalization is not part of [KSWH98, Gut98] .However, to compare these guidelines with the security models described in the next section, we find it relevant to propose the corresponding security model. We first translate their definition of a pseudo-random number generator with input and secondly we translate their adversary descriptions in the real or random model.

As described in [KSWH98, Gut98], a pseudo-random number generator with input is a couple of deterministic algorithms, a first one to collect inputs and a second one to generate outputs. To be consistent with the descriptions of other models, we name (refresh, next) this couple of algorithms, where algorithm $S' \leftarrow \mathsf{refresh}(S, I)$ takes as input an input $I \in \{0,1\}^p$ and the current internal state $S \in \{0,1\}^n$ and produces a new internal state $S' \in \{0,1\}^n$, and algorithm $(S', R) \leftarrow \mathsf{next}(S)$ generates an output $R \in \{0,1\}^\ell$ and produces a new internal state $S' \in \{0,1\}^n$. In our formalization, we also denote q_r the number of inputs that the pseudo-random number generator with input is allowed to use with algorithm refresh.

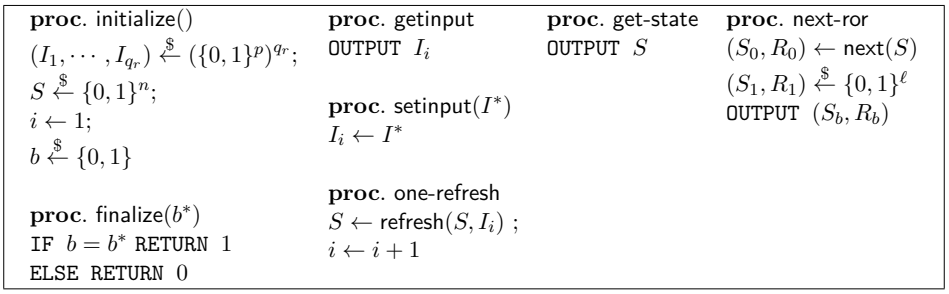

Figure 3.1 – Procedures for Security Games DCA, IBA, SCA

To formalize the security game, we use the procedures described in Figure 3.1. The procedure initialize allows challenger to set the internal state S of the generator, to generate a sequence of random inputs $(I_1, \cdots, I_{q_r})$ and to generate the Boolean parameter b used to challenge adversary $\mathcal{A}$. After all oracle queries, adversary $\mathcal{A}$ outputs a bit b^*, given as input to the procedure finalize, which is used by the challenger to compare the response of $\mathcal{A}$ to the challenge bit b.

We formalize DCA, IBA and SCA as follow:

- To formalize DCA, we use the procedure named next-ror. This procedure challenges $\mathcal{A}$ on its capability to distinguish the output of the generator from random, where the real output (R_0) of the generator is obtained with a call to algorithm next and the random string (R_1) is generated by the challenger.
- To formalize IBA, we use the procedures named getinput, setinput and one-refresh: procedure getinput allows $\mathcal{A}$ to get access to the current input I, procedure setinput allows $\mathcal{A}$ to set the current input to a chosen value I^*. Finally, procedure one-refresh allows challenger to update the current internal state S with algorithm refresh applied with the current input I.
- To formalize SCA, we use the procedure named get-state. This procedure gives $\mathcal{A}$ access to the current value of the internal state S.

The security of a pseudo-random number generator with input is given in Definition 16.

Definition 16 (Security of a pseudo-random number generator with input [KSWH98, Gut98])**.** *A pseudo-random number generator with input* (refresh, next) *is called* (t, ε)-secure against Direct Cryptanalytic Attack (resp. Input-Based Attack or State Compromise Attack)*, if for any adversary* $\mathcal{A}$ *running in time at most* t*, the advantage of* $\mathcal{A}$ *in game* DCA*, (resp.* IBA*,* SCA*) is at most* ε*, where:*

- DCA *is the restricted game where* $\mathcal{A}$ *is only allowed to make calls to* next-ror*.*
- IBA *is the restricted game where* $\mathcal{A}$ *is not allowed to make any calls to* get-state*, and is allowed to make calls to* getinput*,* setinput *and* next-ror*.*
- SCA *is the restricted game where* $\mathcal{A}$ *is not allowed to make calls to* getinput *or* setinput *and is allowed to make calls to* get-state *and* next-ror*.*

Comparison Between Notions. Security game DCA is similar to the security game PR. However, these two security notions can not be compared, as they are not based on the same definition of pseudo-random number generator. Note that if one drops procedure get-state in the security game SCA, we obtain security game DCA and if one drops procedures getinput and setinput in the security game IBA, we also obtain security game DCA.

3.3 Security Model From [BY03]

3.3.1 Description

In 2003, Bellare and Yee [BY03] generalized the notion of standard pseudo-random number generators (Definition 8) 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 formalized in Definition 17 and illustrated in Figure 3.2.

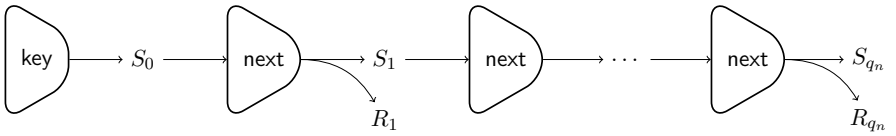

Figure 3.2 – Stateful Pseudo-Random Number Generator [BY03]

Definition 17 (Stateful Pseudo-Random Number Generator [BY03])**.** *A stateful pseudo-random number generator is a couple of algorithm* $(\mathsf{key}, \mathsf{next})$ *and an integer* q_n*, where* key *is a probabilistic algorithm that takes no input and outputs an initial state* $S \in \{0,1\}^n$*,* next *is a deterministic algorithm that, given the current state* S*, outputs a pair* $(S', R) \leftarrow \mathsf{next}(S)$ *where* S' *is the new state and* $R \in \{0,1\}^\ell$ *is the output and* q_n *is the maximal number of outputs the pseudo-random number generator is allowed to produce.*

Bellare and Yee proposed a new security property where a stateful pseudo-random number generator shall be designed so that it is infeasible to recover any information on previous states or previous output blocks 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.

The security game BY-FWD uses procedures described in Figure 3.3. The procedure initialize sets the first internal state S with a call to algorithm key and sets the random parameter b. 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. The other procedures are defined below:

- Procedure next-ror: This procedure challenges $\mathcal{A}$ on its capability to distinguish the output of the stateful pseudo-random number generator from random, where the real output (R_0) of the stateful pseudo-random number generator is obtained with a call to algorithm next and the random string (R_1) is picked uniformly at random by the challenger. Attacker $\mathcal{A}$ responds to the challenge with a bit b^*.
- Procedure get-state: This procedure gives $\mathcal{A}$ access to the current value of the internal state S.

Definition 18 (Forward Security of a Stateful Pseudo-Random Number Generator [BY03])**.** *A stateful pseudo-random number generator* $\mathcal{G} = (\mathsf{key}, \mathsf{next}, q_n)$ *is called* $(T = (t, q_n), \varepsilon)$-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 oracle call* $\mathcal{A}$ *is allowed to make, the advantage of* $\mathcal{A}$ *in game* BY-FWD *is at most* ε*.*

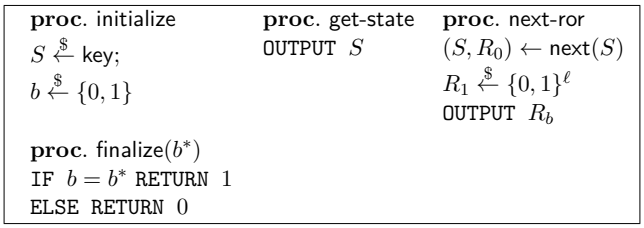

Figure 3.3 – Procedures in Security Game BY-FWD

Comparison with previous models. If one drops procedure get-state in the security game BY-FWD, we come back to the usual security of a stateful standard pseudo-random number generator SPR. As for the state compromise, BY-FWD has the same objective than the security game SCA, from [KSWH98, Gut98]. However, the two security models can not be compared, as one concerns pseudo-random number generator with input and the other stateful pseudo-random number generator and they do not rely on the same definition. A stateful pseudo-random number generator does not contain a refresh algorithm that would be used to periodically refresh its internal state of twith new inputs. Similarly, there is no relation between BY-FWD and IBA.

3.3.2 A Secure Construction

Let $\mathbf{G} : \{0,1\}^n \to \{0,1\}^{n+\ell}$ a $(t, \varepsilon_{\mathbf{G}})$-secure standard pseudo-random number generator, as formalized in Definition 8. Consider the stateful pseudo-random number generator GEN, defined with the following algorithms:

- GEN.key : returns $S \xleftarrow{\$} \{0,1\}^n$.
- GEN.next, on input S, returns $(S', R) = \mathbf{G}(S)$.

We prove the forward security of GEN by reduction to the standard security of $\mathbf{G}$.

Theorem 4 (Security of GEN [BY03]). *Let $\mathbf{G} : \{0,1\}^n \to \{0,1\}^{n+\ell}$ be a $(t, \varepsilon_{\mathbf{G}})$-secure standard pseudo-random number generator, $n \geq 1$ be an integer and let* GEN *be the stateful generator associated to $\mathbf{G}$, as described above. Then* GEN *is $((t', q_n), 2q_n\varepsilon_{\mathbf{G}})$-backward secure, with $t' \approx t$.*

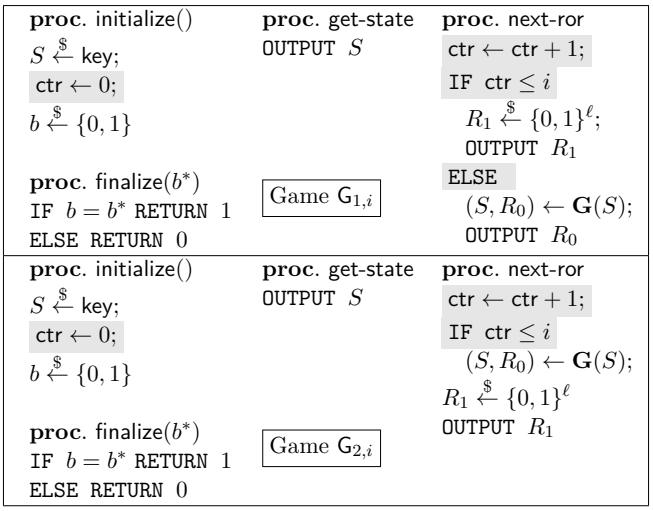

Figure 3.4 – Reduction to the Standard Security for BY-FWD

Proof. We adapt the proof from [BY03] in the game playing framework presented in Section 2.4. Consider two sequences of hybrid security games where G_0 is the initial forward security game BY-FWD, games $\mathsf{G}_{1,i}$ are modifications of game G_0 and games $\mathsf{G}_{2,i}$ are modifications of game G_{1,q_n}, for $i = 1, \cdots n$, all described in Figure 3.4. The differences between G_0 and $\mathsf{G}_{1,i}$ and between G_{1,q_n} and $\mathsf{G}_{2,i}$ are explained below:

Differences Between G_0 and $\mathsf{G}_{1,i}$: in game $\mathsf{G}_{1,i}$, procedure initialize is different from game G_0: the challenger sets a new parameter ctr to 0. Procedure next-ror is different from G_0: ctr is incremented and if $\mathsf{ctr} \leq i$, the challenger generates a random output R_1 and returns it to $\mathcal{A}$. If $\mathsf{ctr} > i$, the challenger behaves as in G_0.

Differences Between G_{1,q_n} and $\mathsf{G}_{2,i}$: in game $\mathsf{G}_{2,i}$, procedure initialize also sets a new parameter ctr to 0, as in $\mathsf{G}_{1,i}$. Procedure next-ror behaves as follow: ctr is incremented and if $\mathsf{ctr} \leq i$, the challenger generates the real output couple $(S, R_0) \leftarrow \mathsf{next}(S)$ (this call is used to update the internal state), then for any value of ctr, a random output R_1 is generated and sent to $\mathcal{A}$.

Note that $\Pr[\mathsf{G}_0 = 0] = \Pr[\mathsf{G}_{1,0} = 0]$, $\Pr[\mathsf{G}_{1,q_n} = 1] = \Pr[\mathsf{G}_{2,0} = 1]$, $\Pr[\mathsf{G}_0 = 1] = \Pr[\mathsf{G}_{2,q_n} = 1]$.

We construct an adversary $\mathcal{A}'$ with advantage $\varepsilon_{\mathbf{G}}$ in game PR, whose objective is to distinguish between $\mathsf{G}_{1,i}$ and $\mathsf{G}_{1,i+1}$ and between $\mathsf{G}_{2,i}$ and $\mathsf{G}_{2,i+1}$, for $i = 0, \cdots, q_n - 1$.

First consider the distance between $\mathsf{G}_{1,i}$ and $\mathsf{G}_{1,i+1}$. Consider an adversary $\mathcal{A}$ in both games, that will be used by $\mathcal{A}'$ as a subroutine. The challenger of $\mathcal{A}'$ generates a random state S' and a random bit b', then it generates a couple $(S'_0, R'_0) = \mathsf{GEN}(S')$, a random couple (S'_1, R'_1) and sends (S'_b, R'_b) to $\mathcal{A}'$. Then $\mathcal{A}'$ challenges $\mathcal{A}$ in game $\mathsf{G}_{1,i}$: $\mathcal{A}'$ generates a random bit b and initializes a counter ctr to 0. Following, $\mathcal{A}'$ responds to oracle queries of $\mathcal{A}$ in game $\mathsf{G}_{1,i}$. It increments ctr and:

- If $\mathsf{ctr} < i$, then $\mathcal{A}'$ generates a random sample R_1 and sends it to $\mathcal{A}$.
- If $\mathsf{ctr} \geq i$, then $\mathcal{A}'$ sets $S_0 = S'_b$, computes the successive couples $(S_0, R_0) \leftarrow \mathsf{next}(S_0)$ and sends the output R_0 to $\mathcal{A}$.
- Finally, it responds to the get-state query with the last calculated state S_0.

Then $\mathcal{A}$ answers the bit b^* to $\mathcal{A}'$ and $\mathcal{A}'$ responds to its challenger the bit $b'^* = 1$ if $b^* = b$ and the bit $b'^* = 0$ elsewhere. Then if $b' = 0$, then $\mathcal{A}'$ exactly simulates game $\mathsf{G}_{1,i}$, while if $b' = 1$, then $\mathcal{A}'$ simulates game $\mathsf{G}_{1,i+1}$. Therefore the distance between games $\mathsf{G}_{1,i}$ and $\mathsf{G}_{1,i+1}$ is bounded by $\varepsilon_{\mathbf{G}}$.

Similarly, consider the distance between $\mathsf{G}_{2,i}$ and $\mathsf{G}_{2,i+1}$. The challenger of $\mathcal{A}'$ generates a random state S' and a random bit b', then it generates a couple $(S'_0, R'_0) = \mathsf{GEN}(S')$, a random couple (S'_1, R'_1) and sends (S'_b, R'_b) to $\mathcal{A}'$. Then $\mathcal{A}'$ challenges $\mathcal{A}$ in game $\mathsf{G}_{2,i}$: $\mathcal{A}'$ generates a random bit b and initializes a counter ctr to 0. Following, $\mathcal{A}'$ responds to oracle queries of $\mathcal{A}$ in game $\mathsf{G}_{2,i}$. It increments ctr and:

- If $\mathsf{ctr} < i$, then $\mathcal{A}'$ sets $S_0 = S'_b$, computes the successive couples $(S_0, R_0) \leftarrow \mathsf{next}(S_0)$, generates a random sample R_1 and sends it to $\mathcal{A}$.
- If $\mathsf{ctr} \geq i$, then $\mathcal{A}'$ generates a random sample R_1 and sends it to $\mathcal{A}$.
- Finally, it responds to the get-state query with the last calculated state S_0.

Then $\mathcal{A}$ answers the bit b^* to $\mathcal{A}'$ and $\mathcal{A}'$ responds to its challenger the bit $b'^* = 1$ if $b^* = b$ and the bit $b'^* = 0$ elsewhere. Then if $b' = 0$, then $\mathcal{A}'$ exactly simulates game $\mathsf{G}_{2,i}$, while if $b' = 1$, then $\mathcal{A}'$ simulates game $\mathsf{G}_{2,i+1}$. Therefore the distance between games $\mathsf{G}_{2,i}$ and $\mathsf{G}_{2,i+1}$ is bounded by $\varepsilon_{\mathbf{G}}$.

Finally, the above reductions show that $|\Pr[\mathsf{G}_0 = 0] - \Pr][\mathsf{G}_0 = 1] \leq 2q_n\varepsilon_{\mathbf{G}}$. □

3.4 Security Model from [DHY02]

3.4.1 Description

Desai, Hevia and Yin [DHY02] proposed a security model for pseudo-random number generator with input where the internal state is split into two parts: a first part named K (that they name the *key*) and second part named S (that they name the *state*). In their model, a pseudo-random number generator with input is a stateful and iterative algorithm, that at each invocation produces some output bits as a function of the current value of K and S, in addition to another *auxiliary* input I, then updates the state S, and then deletes the old one. They proposed different security properties, that capture the potential compromise of the state S, the key K or the auxiliary input I. The generator operations are illustrated in Figure 3.5, in accordance with Definition 19.

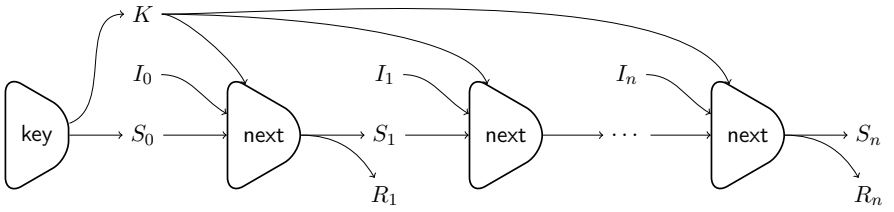

Figure 3.5 – Pseudo-Random Number Generator with Input [DHY02]

Definition 19 (Pseudo-Random Number Generator with Input [DHY02])**.** *A pseudo-random number generator with input is a couple of algorithms* $(\mathsf{key}, \mathsf{next})$*, where* key *is a probabilistic algorithm that takes no input and outputs a key* $K \in \{0,1\}^n$ *and an initial state* $S \in \{0,1\}^n$*,* 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* $(S', R) \leftarrow \mathsf{next}(S, K, I)$ *where* $S' \in \{0,1\}^n$ *is the new state and* $R \in \{0,1\}^\ell$ *is the output.*

Note that the model assumes the existence of an entropy pool in which entropy is accumulated and that is used as input for algorithm key to generate the key K and a first state S. However, the model does not propose any secure way to accumulate entropy nor capture the potentially adversarial inputs that may be accumulated. Also note that in Definition 19, the next algorithm updates the state S while the key K is not updated.

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 in that 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.

We now describe their security model. The security games CIA, CSA, KKA use procedures described in Figure 3.6. In our description, we denote with i the counter on the auxiliary inputs and with q_r the maximal number of auxiliary inputs the generator is allowed to use. The procedure initialize sets a random key K and a random state S with a call to algorithm key, generates q_r random inputs $(I_1, \cdots, I_{q_r}) \xleftarrow{\$} (\{0,1\}^p)^{q_r}$, sets a counter i to 0 and sets a random Boolean parameter b. 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. The other procedures are defined below:

- The procedures getinput / setinput are used by $\mathcal{A}$ to get or set the value of the auxiliary inputs.
- The procedures get-state / set-state are used by $\mathcal{A}$ to get or set the current value of the internal state S.
- The procedure get-key is used by $\mathcal{A}$ to get the value of the key K.
- The procedure next-ror challenges $\mathcal{A}$ on its capability to distinguish the output of algorithm next from random, where the real output (R_0) is obtained with a call to algorithm next and the random string (R_1) is generated by the challenger. The counter i is incremented during this procedure. Attacker $\mathcal{A}$ responds to the challenge with a bit b^*.

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

Figure 3.6 – Procedures in Security Games CIA, CSA, KKA

Definition 20 (Security of a Pseudo-Random Number Generator with Input [DHY02])**.** *A pseudo-random number generator with input* (key, next) *is called* ($(T = (t, q_r), \varepsilon)$- 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*, that uses at most* q_r *inputs, the advantage of* $\mathcal{A}$ *in game* CIA*, (resp.* CSA*,* KKA*) is at most* ε*, where:*

- CIA *is the restricted game where* $\mathcal{A}$ *is not allowed to make calls to* get-key *and to* set-state *and is allowed to make calls to* get-state*,* setinput*,* getinput *and* next-ror*.*
- CSA *is the restricted game where* $\mathcal{A}$ *is not allowed to make calls to* get-key *and to* setinput *and is allowed to make calls to* get-state*,* set-state*,* getinput *and* next-ror*.*
- KKA *is the restricted game where* $\mathcal{A}$ *is not allowed to make calls to* get-state*,* set-state *and* setinput *and is allowed to make calls to* get-key*,* getinput *and* next-ror*.*

Hence all security notions allow adversary $\mathcal{A}$ to get the content of the auxiliary input through procedure getinput. In addition, when $\mathcal{A}$ does not have access to the key K (through procedure get-key), $\mathcal{A}$ can mount attacks CIA and CSA. Furthermore, when $\mathcal{A}$ has access to K, the state S shall remain secret.

Comparison with previous models. If one drops procedures getinput, setinput, get-state, set-state and get-key in the security game, the adversary $\mathcal{A}$ has only access to the next-ror procedure. With this single procedure, the objective of the adversary is similar to the objective in the security games PR and DCA, although the definition of the generator is different.

Concerning input compromise, the model focuses on the difference between an adversary that has access to the input (which is possible for all security notions) and an adversary that is able to choose an input. When the key K remains secret, if $\mathcal{A}$ can choose the auxiliary input I, $\mathcal{A}$ only has access to the state S and conversely, if $\mathcal{A}$ can choose the state S, $\mathcal{A}$ only has access to

the auxiliary input I. However CIA is closely similar to IBA described in Section 3.2.2, as they only differ with the addition of procedure get-state for CIA; however, in CIA, the signification of get-state differs from the same procedure in IBA, because in CIA, this procedure implies only a partial compromise of the internal state (considered as the union of S and K), while in IBA this procedure implies a complete compromise.

In addition, the adversary $\mathcal{A}$ has always access to procedure getinput, therefore CIA, CSA and KKA can not be compared with FWD (which does not concern pseudo-random number generator with input). If one wanted to add a definition of *forward security* that would be close to FWD, forward security would be the restricted game where $\mathcal{A}$ is not allowed to make calls to getinput and to setinput and is allowed to make one call to get-state followed with one call to get-key, which are the two last calls $\mathcal{A}$ is allowed to make. However, this notion of forward security is not described in [DHY02].

Finally, the get-key procedure allows $\mathcal{A}$ to get the value of the parameter key and the associated security property KKA ensures that the generator remains safe even if this parameter key gets compromised. This can be used to model pseudo-random number generator with input where a *public* parameter is used, for instance to select from a family of functions to instantiate a randomness extractor, as described in Section 2.6. The use of a public parameter that is the seed of a randomness extractor is the basis of the security models of [BST03] and [BH05].

3.4.2 Secure Constructions

Desai *et al.* proposed constructions secure against CSA, CIA and KKA, that are based on existing standard specifications (ANSI X9.17 [ANS85] and FIPS [DSS00]).

Construction Secure Against CSA and CIA. Let $\mathbf{F}$ a pseudo-random function. Let us use algorithm key to generate a key K for $\mathbf{F}$. This allows to define a function $\mathbf{F}_K : \{0,1\}^n \to \{0,1\}^n$ and an associated pseudo-random number generator with input ANSI, in accordance with [ANS85].

- ANSI.key : $K \xleftarrow{\$} \{0,1\}^n; S_0 \xleftarrow{\$} \{0,1\}^n$, returns (K, S_0),
- ANSI.next(S_{i-1}, K, I_i) : $y_i \leftarrow \mathbf{F}_K(S_{i-1} \oplus \mathbf{F}_K(I_i)), S_i \leftarrow \mathbf{F}_K(y_i \oplus \mathbf{F}_K(I_i))$, returns (y_i, S_i).

Theorem 5 shows that if we model $\mathbf{F}$ as a pseudo-random function, as in Definition 13, then the pseudo-random number generator with input ANSI is secure against CIA and CSA.

Theorem 5 (Security of ANSI [DHY02])**.** *Let* $\mathbf{F} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ *a* $(t, 3q_r, \varepsilon_{\mathbf{F}})$*-pseudo-random function. Let* ANSI *be the pseudo-random number generator with input associated to* $\mathbf{F}$*, as described above. Then* ANSI *is*

- $((t, q_r), 2\varepsilon_{\mathbf{F}} + \frac{q_r(2q_r-1)}{2^n})$ *secure against* CSA*,*
- $((t, q_r), 2\varepsilon_{\mathbf{F}} + \frac{(2q_r-1)^2}{2^n})$ *secure against* CIA*.*

As precised in [DHY02], if one models the generator with a pseudo-random permutation, as in Definition 15, an additional term $\frac{3q_r(3q_r-1)}{2^{n+1}}$ shall be added in the previous bounds. This is a direct application of the 'PRF/PRP Switching Lemma' (Lemma 5).

Construction Secure against KKA. Let $\mathbf{F}$ a pseudo-random function. Let us use algorithm key to generate a key K for $\mathbf{F}$. This allows to define a function $\mathbf{F}_K : \{0,1\}^n \to \{0,1\}^n$ and an associated pseudo-random number generator with input FIPS in accordance with [DSS00]:

- FIPS.key : $K \xleftarrow{\$} \{0,1\}^n; S_0 \xleftarrow{\$} \{0,1\}^n$, returns (K, S_0),

- $\mathsf{FIPS.next}(S_{i-1}, K, I_i)$: $y_i \leftarrow \mathbf{F}_K(S_{i-1} + I_i \bmod 2^n), S_i \leftarrow S_{i-1} + y_i + 1 \bmod 2^n$, returns (y_i, S_i).

Theorem 6 shows that if we model $\mathbf{F}$ as a pseudo-random function, as in Definition 13, then the pseudo-random number generator with input FIPS is secure against KKA. The proof of Theorem 6 is similar to the proof of Theorem 5.

Theorem 6 (Security of FIPS [DHY02])**.** *Let* $\mathbf{F} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ *a* $(t, q_r, \varepsilon_{\mathbf{F}})$*-pseudo-random function. Let* FIPS *be the pseudo-random number generator with input associated to* $\mathbf{F}$*, as described above. Then* FIPS *is* $((t, q_r), \varepsilon_{\mathbf{F}} + \frac{q_r(q_r-1)^2}{2^{n-1}})$*-secure against* KKA*.*

Remark 1. *The argument in [DHY02] to prove these theorems relies 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, therefore we are not confident with the bounds presented in these theorems.*

Consider now the *forward security* of the generators ANSI and FIPS. As noticed, this security notion is not part of the security model of [DHY02], however, one could formalize it at the restricted game where $\mathcal{A}$ is not allowed to make calls to getinput and to setinput and is allowed to make one call to get-state followed with one call to get-key, which are the two last calls $\mathcal{A}$ is allowed to make. If one used this definition, then the generator ANSI is *not* forward secure, as the complete compromise of state S and K allows adversary $\mathcal{A}$ to completely reverse the underlying block cipher and therefore to compute past outputs. A similar attack can be mounted on the generator FIPS.

3.5 Security Model From [BST03]

3.5.1 Description

Barak, Shaltiel and Tromer [BST03] defined a resilient pseudo-random number generator with input. They named their class of pseudo-random number generators with input 'True Random Number Generators'.

Their objective is to model situations where the entropy source can be influenced by an adversary who has partial control on it and to model a generator that will be secure against such an adversary. This model is the first that considers potentially adversarial inputs and the distribution of the inputs instead of their real values.

They intended to model 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 control at most τ Boolean properties of the environment, and can thus create at most 2^τ different environments. Their definition of pseudo-random number generator with input is not explicitly given in [BST03], but it simply corresponds to a stateless algorithm that takes two inputs, corresponding to the inputs of a (k, ε)-resilient extractor, as given in Definition 5. This notion of extractor assumes the use of a public parameter seed that shall be selected once for all and therefore can be pre-processed and even hard-coded. The price for this is that the family $\mathcal{F}$ of k-sources is bounded. They explicitly set the size of the family to 2^τ.

Definition 21 (Pseudo-Random Number Generator with Input [BST03])**.** *A pseudo-random number generator with input is an algorithm $\mathcal{G}$ that given a first input $I \in \{0,1\}^p$ and a second input* $\mathsf{seed} \in \{0,1\}^s$*, outputs* $\mathcal{G}(I, \mathsf{seed}) = R \in \{0,1\}^\ell$*.*

The associated security game allows an adversary $\mathcal{A}$ to choose a family of distributions $\mathcal{F}$, of size 2^τ: $\mathcal{F} = \{\mathcal{D}_1, \cdots, \mathcal{D}_{2^\tau}\}$, such that for all i and all input I sampled with $\mathcal{D}_i$, I are k-sources ($\mathbf{H}_\infty(I) \geq k$) for all $i \in \{1, \ldots, 2^\tau\}$. The security game uses procedures described in Figure 3.7 and is explained below:

1. The procedure initialize allows $\mathcal{A}$ to set the family $\mathcal{F}$, the challenger parses $\mathcal{F}$ as 2^τ distributions $\mathcal{D}_1, \cdots, \mathcal{D}_{2^\tau}$, sets the public parameter seed and sets the Boolean parameter b. The parameter seed is given to the adversary $\mathcal{A}$.

2. The procedure next-ror challenges $\mathcal{A}$ on its capability to distinguish the output of $\mathcal{G}$ from random: $\mathcal{A}$ chooses a distribution $\mathcal{D}_i \in \mathcal{F}$. Then the challenger samples an input I of distribution $\mathcal{D}_i$ and finally the challenger generates the real output ($R_0 = \mathcal{G}(\mathsf{seed}, I)$) and picks a random string (R_1). The challenge R_b is returned to the adversary $\mathcal{A}$.

3. Attacker $\mathcal{A}$ responds to the challenge with a bit b^*, which is compared with the previously generated bit b by the challenger in procedure finalize.

proc. initialize($\mathcal{F}$)	**proc**. next-ror(i)
seed $\xleftarrow{\$} \{0,1\}^s$;	$I \xleftarrow{\$} \mathcal{D}_i$
parse $\mathcal{F}$ as $\{\mathcal{D}_1, \cdots, \mathcal{D}_{2^\tau}\}$	$R_0 \leftarrow \mathcal{G}(I, \mathsf{seed})$
$b \xleftarrow{\$} \{0,1\}$;	$R_1 \xleftarrow{\$} \{0,1\}^\ell$
OUTPUT seed	RETURN R_b
proc. finalize(b^*)	
IF $b = b^*$ RETURN 1	
ELSE RETURN 0	

Figure 3.7 – Procedures in Security Game BST-RES(τ)

Definition 22 (Resilience of Pseudo-Random Number Generator with Input [BST03])**.** *A pseudo-random number generator with input* $\mathcal{G} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^\ell$ *is* (t, τ, ε)*-Resilient if for any adversary running in time* t*, with probability* $(1 - \varepsilon)$ *over the choice of* seed*, the advantage of* $\mathcal{A}$ *in game* BST-RES(τ) *it at most* ε*.*

Note that if one parses the family $\mathcal{F}$ as $\{\mathcal{D}_1, \cdots, \mathcal{D}_{2^\tau}\}$, then for all i and all input I sampled with $\mathcal{D}_i$, we have $\mathbf{H}_\infty(I) \geq k$ for all $i \in \{1, \ldots, 2^\tau\}$. Recall (Definition 5) that a function $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^\ell$ is a (k, ε)-resilient extractor if for all finite families of k-sources $\mathcal{F}$, with probability at least $(1 - \varepsilon)$ over the choice of $\mathsf{seed} \xleftarrow{\$} \{0,1\}^s$, the distributions $(\mathsf{seed}, \mathsf{Extract}(X, \mathsf{seed}))$ and $(\mathsf{seed}, \mathcal{U}_\ell)$ are ε-close, for all $X \in \mathcal{F}$. Hence if $\mathcal{G}$ is a resilient extractor, it is a resilient pseudo-random number generator with input.

Comparison with Previous Models. In this security model, the adversary can choose a high entropy distribution from a finite family of distributions. Once the distribution is chosen, the extraction is processed and the adversary is challenged on its capacity to distinguish the output of the extraction from random. This security property is closely related to the previous security properties of IBA (from [KSWH98, Gut98]) and CIA (from [DHY02]), however they are not equivalent as they are not based on the same definition of a generator. In particular, here there is no equivalent to the getinput or setinput procedures, that allows $\mathcal{A}$ to get or set the input given to the extractor. In fact, the security model shall be seen as an extension of the initial

PR model, where the challenger has complete control on the generation of the first internal state, here, we give the adversary some control on this generation as we allow her to choose the distribution from which the first internal state will be generated. The composition of a randomness extractor and a secure or forward-secure stateful pseudo-random number generator will be the core of the work of Barak and Halevi in [BH05].

3.5.2 A Secure Construction

Barak, Shaltiel and Tromer proposed several constructions, based on pairwise independent hash functions families. They proposed two results, that link the length of the parameter seed with the number of distributions (2^τ) that are chosen at the beginning of the security game BST-RES(τ), with the min-entropy of the distribution k, with the size of the input p and with the size of the output m.

Theorem 7 gives the two results, first for a size of parameter seed that is the double of the size of the inputs ($s = 2p$), and second that is a multiple of the size of the inputs ($s = \kappa p$).

Theorem 7 (Existence of Resilient Pseudo-Random Number with Input [BST03])**.** *For every p, k, ℓ, t and ε, there is a (t, τ, ε)-resilient pseudo-random number with input, with a public parameter* seed *of size $s = 2p$ such that $\tau = \frac{k-\ell}{2} - 2\log(1/\varepsilon) - 1$ and there is a (t, τ, ε)-resilient pseudo-random number with input, with a public parameter* seed *of size $s = \kappa p$ such that: $\tau = \frac{\kappa}{2}(k - \ell - 2\log(1/\varepsilon) - \log\kappa + 2) - \ell - 2 - \log(1/\varepsilon)$.*

Note that the proof given in [BST03] for the bound on the resilient generator with a public seed of size $s = \kappa p$ uses the probabilistic method, a similar argument than the proof of the existence of seeded extractor (Theorem 3) based on the Chernoff bound (Proposition 1). The proof for the bound on the resilient generator with a public seed of size $s = 2p$ is constructive as it is a direct application of the Leftover Hash Lemma for a finite pairwise independent hash function family (Section 2.7).

Concrete Construction. They proposed one concrete construction of pairwise independent families of hash functions, based on simple operations in a finite field. Let $\mathbb{F}_{2^p}$ be the field with 2^p elements, and consider the set $S = \{(a, b) | a, b \in \mathbb{F}_{2^p}\}$. For $s = (a, b) \in S$, $I \in \{0,1\}^p$ and $\ell < p$, let $h_s(I) = [a \cdot I + b]_\ell$ (the ℓ first bits of $[a \cdot I + b]$, where all operations are in $\mathbb{F}_{2^p}$). Then the family $\mathcal{H} = \{h_s, s \in S\}$ is a pairwise independent family of hash functions. Barak *et al.* noticed that $h_s(I)$ is close to uniform if and only if $[a \cdot I]_\ell$ is close to uniform. They proposed the following process to build a resilient generator:

1. In a first stage (the *preprocessing stage*), choose an irreducible polynomial of degree p and generate a random parameter $a \xleftarrow{\$} \{0,1\}^p$. The union of parameters of the polynomial and of the parameter a is the public parameter seed of the extractor, that can be hard-coded. Hence we have for this construction $|\mathsf{seed}| = d = 2p$.

2. In a second stage (the *runtime*), set $\mathcal{G}(\mathsf{seed}, I) = [a \cdot I]_\ell$.

Note that this construction involves a multiplication in the binary field $\mathbb{F}_{2^p}$ followed by a truncation. This composition (multiplication followed by truncation) will be used in the design of a *robust* pseudo-random number generator with input, as described in Chapter 4.

3.6 Security Model From [BH05]

3.6.1 Description

Barak and Halevi [BH05] proposed a new security model for pseudo-random number generator 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-theoric and generation is cryptographic. Furthermore, these two operations should be separated and analysed independantly. The generator operations are illustrated in Figure 3.8, in accordance with Definition 23.

Definition 23 (Pseudo-Random Number Generator with Input [BH05])**.** *A pseudo-random number generator with input is a couple of algorithms* $(\mathsf{refresh}, \mathsf{next})$ *where* $\mathsf{refresh}$ *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* $S' \leftarrow \mathsf{refresh}(S, I)$ *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* $(S', R) \leftarrow \mathsf{next}(S)$ *where* $S' \in \{0,1\}^n$ *is the new state and* $R \in \{0,1\}^\ell$ *is the output of the generator.*

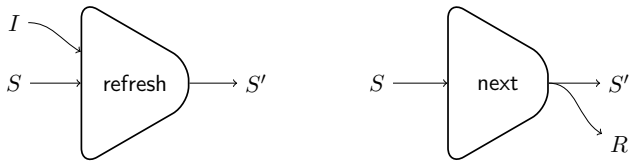

Figure 3.8 – Pseudo-Random Number Generator with Input [BH05]

In their security model, they aimed to capture 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 security properties that are defined in order respond to these attacks are the following ones:

- **Resilience**. The generator's output looks random to an observer with no knowledge of the internal state. This holds even if that observer has complete control over data that is used to refresh the internal state.
- **Forward security**. Past output of the generator looks random to an observer, even if the observer learns the internal state at a later time.
- **Backward security**. Future output of the generator looks random, even to an observer with knowledge of the current state, provided that the generator is refreshed with data of sufficient entropy.

It is important to note that Barak and Halevi used a notion of randomness extractor $\mathsf{Extract}$ that is parametrised by a family of distributions $\mathcal{H}$ (which in their work stands for 'high entropy distribution'). Formally, they proposed to use Definition 24 to describe $\mathcal{H}$*-extractors*. In this definition, to be consistent with the already used notations, we denote the size of the inputs with p and we denote the size of the output of the randomness extractor with m.

Definition 24 ($\mathcal{H}$-Extractor [BH05])**.** *Let* p, m *be integers such that* $p \geq m$ *and let* $\mathcal{H}$ *be a family of distributions over* $\{0,1\}^p$*. A function* $\mathsf{Extract} : \{0,1\}^p \to \{0,1\}^m$ *is an* $\mathcal{H}$*-extractor if for every* $\mathcal{D} \in \mathcal{H}$ *and every* $I \xleftarrow{\$} \mathcal{D}$*,* $\mathsf{Extract}(I)$ *is* 2^{-m}*-close to* $\mathcal{U}_m$*.*

In this definition, the extraction is done over the set $\{0,1\}^p$, and not over a couple of sets $\{0,1\}^p \times \{0,1\}^s$. Hence one possibility for the function Extract is to consider that it is a *deterministic* extractor, as in Definition 3. However, as Lemma 2 shows, such extractor cannot extract randomness from any k-source and therefore a clear limitation on the sources of randomness for such an extractor should be given. Note that this impossibility is also mentioned by Barak and Halevi, who gave mentionned [BST03] as an example of possible construction. However, the security model of [BST03] implicitly uses the notion of *resilient* extractor, which is a particular case for *seeded* extractor, as in Definition 4. Therefore we prefer to consider that the extractor is *seeded*, ensuring that its existence is guaranted with Theorem 3. As a consequence, we claim that the notion of extractor in Definition 24 is a special case of seeded extractor, *i.e.* a function $\mathsf{Extract} : \{0,1\}^p \times \{0,1\}^s \to \{0,1\}^m$, where the parameter seed is sampled from $\{0,1\}^s$. In the corresponding security game, the parameter seed should be generated once for all and made available to the adversary.

One more point that still remains is the correlation between the randomness source and the parameter seed, that is: do we consider that Extract is a *strong* extractor, as in Definition 7, or a *resilient* extractor, as in Definition 5 ? In our opinion, as [BH05] explicitly mentions [BST03] as a conform extractor, we state that [BH05] refers to resilient extractors.

Recall that the definition of a *resilient* randomness extractor stands for a finite family of distributions. However Barak and Halevi did not explicitly set the size of the family $\mathcal{H}$, therefore a bound is implicitly set from our assumption. Also from our assumption, we assume that in addition to the description done by Barak and Halevi, a *public* parameter seed is first randomly generated, which is the seed of the randomness extractor. They formalized the corresponding property in a security game in the real or random model, as described in Section 2.4. In our description, we denote by i the identifier for an element in the family of distribution $\mathcal{H} = \mathcal{D}_i$, where I is of finite length $q_r = |I|$ and q_n is the upper bound on the number of outputs.

proc. initialize	**proc**. good-refresh(i)	**proc**. set-state(S^*)	**proc**. next-ror
$S_0 \leftarrow 0^n$;	$x \xleftarrow{\$} \mathcal{D}_i$;	IF corrupt = true	$(S_0, R_0) \leftarrow \mathsf{next}(S_0)$
corrupt $\leftarrow$ true;	$S_0 \leftarrow \mathsf{refresh}(S_0, x)$;	OUTPUT S_0	IF corrupt = true,
$b \xleftarrow{\$} \{0,1\}$;	corrupt $\leftarrow$ false;	ELSE	OUTPUT R_0
parse $\mathcal{H}$ as $\{\mathcal{D}_i\}_{i \in I}$		$S_1 \xleftarrow{\$} \{0,1\}^n$	ELSE
seed $\xleftarrow{\$} \{0,1\}^s$;	**proc**. bad-refresh(x)	OUTPUT S_b	$R_1 \xleftarrow{\$} \{0,1\}^\ell$
OUTPUT seed	IF corrupt = true	corrupt $\leftarrow$ true	OUTPUT R_b
	$S_0 \leftarrow \mathsf{refresh}(S_0, x)$;	$S \leftarrow S*$	
proc. finalize(b^*)	ELSE $\perp$		
IF $b = b^*$ RETURN 1			
ELSE RETURN 0			

Figure 3.9 – Procedures in Security Game BH-ROB($\mathcal{H}$)

The security game uses procedures described in Figure 3.9. Before the description of the procedures, let us first clarify some important points on the security game:

- In order to clarify the original security model from [BH05], we let the challenger parse the family of distribution $\mathcal{H}$ as $\{D_i\}_{i \in I}$, where I is of finite length.

- In the security game, the adversary $\mathcal{A}$ has always two choices to refresh the generator, either with an input with high entropy, either with an input that $\mathcal{A}$ totally controls. In the first case, $\mathcal{A}$ uses procedure good-refresh: with this procedure, the challenger lets $\mathcal{A}$ choose the distribution from the family $\mathcal{H}$ that $\mathcal{A}$ wants to use, then $\mathcal{A}$ generates an input of the chosen distribution and finally applies algorithm refresh with the previously

generated input. In the second case, $\mathcal{A}$ uses procedure bad-refresh: with this procedure, the challenger lets $\mathcal{A}$ choose 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 maintained by the challenger, is part of the security game and is not a component of the generator. More precisely, whenever the adversary gets or sets the internal state, the flag is set to true and as soon as the generator is refreshed with a high entropy input, the flag is set to false.

- The security objective as conceived by Barak and Halevi concerns the pseudo-randomness of the output of the generator R and also the pseudo-randomness of the state S. Therefore in the associated security game, the adversary $\mathcal{A}$ is challenged on distinguishing the output of the generator from random (through procedure next-ror) and on distinguishing the state of the generator from random (through procedure set-state). Furthermore, the security model allows to define *resilience*, *backward security* and *forward security* using clearly defined security games. In addition, they proposed a new security property, named *robustness* that implies each of the previous security properties.

Let us now describe the procedures. The procedure initialize sets the first internal state S_0 with a call to algorithm key and sets parameter b. 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. The other procedures are defined below:

- Procedure good-refresh: on input $i \in I$, it first samples an input x of distribution $\mathcal{D}_i$ then refreshes the internal state of the generator with x.

- Procedure set-state: it generates a challenge for $\mathcal{A}$ on its capability to distinguish the state of the generator from random, where the real state (S_0) is the current state and the random string (S_1) is generated by the challenger. Furthermore, it allows $\mathcal{A}$ to set the state to a new value S^*.

- Procedure next-ror: it challenges $\mathcal{A}$ on its capability to distinguish the output of $\mathcal{G}$ from random, where the real output (R_0) of the generator is obtained with a call to algorithm next and the random string (R_1) is generated by the challenger. Attacker $\mathcal{A}$ responds to the challenge (and the previous on the state) with a bit b^*.

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. Similarly, the output of procedure set-state also depends on the flag corrupt: the real state is given to $\mathcal{A}$ if corrupt = true, otherwise, a random state is generated and given to $\mathcal{A}$.

The security of a pseudo-random number generator with input is given in Definition 25. In the original definition of [BH05], only the notion of *robustness* is given. In their original work, Barak and Halevi stated that robustness implies *resilience*, *backward security* and *forward security*, respectively, although they do not prove these implications. With Definition 25, the implications are direct.

Definition 25 (Security of a Pseudo-Random Number Generator with Input [BH05])**.** *A pseudo-random number generator with input* $\mathcal{G} = (\mathsf{refresh}, \mathsf{next})$ *is called* (t, ε)-robust (resp. resilient, forward secure, backward secure) *for the family* $\mathcal{H}$*, if for any adversary* $\mathcal{A}$ *running in time at most*

t, the advantage of $\mathcal{A}$ in game BH-ROB($\mathcal{H}$) *(resp.* BH-RES($\mathcal{H}$), BH-FWD($\mathcal{H}$), BH-BWD($\mathcal{H}$)*) is at most ε, where:*

- BH-ROB($\mathcal{H}$) *is the unrestricted game where $\mathcal{A}$ is allowed to make all the above calls.*
- BH-RES($\mathcal{H}$) *is the restricted game where $\mathcal{A}$ is allowed to make calls to* good-refresh *to* bad-refresh *and to* next-ror *and is not allowed to make any calls to* set-state*.*
- BH-FWD($\mathcal{H}$) *is the restricted game where $\mathcal{A}$ is allowed to make calls to* good-refresh*, to* bad-refresh*, to* next-ror *and to* set-state *which is the last oracle call $\mathcal{A}$ is allowed to make.*
- BH-BWD($\mathcal{H}$) *is the restricted game where $\mathcal{A}$ is allowed to make calls to* good-refresh *to* bad-refresh*, to* next-ror *and to* set-state *which is the first oracle call $\mathcal{A}$ is allowed to make.*

Comparison with Previous Models. The model of [BST03] also considers the potential compromise of the source where the adversary $\mathcal{A}$ can choose a finite family of distributions. While in the security models of [KSWH98, Gut98, DHY02], adversary $\mathcal{A}$ has access to a setinput procedure that allows her to directly set the value of the input that is collected by the random number generator, here the adversary has access to a procedure named bad-refresh, that is similar to the procedure setinput. However the associated security property CIA, from the model of [DHY02], is stronger as it allows $\mathcal{A}$ to get the content of the internal state through a call to get-state in addition to the chosen input.

Forward security is also captured in the security model of [BY03], with Definition 18: adversary $\mathcal{A}$ has access to two procedures get-state and set-state that allow her to get or to set the content of the internal state. As in Definition 18, the call to either get-state and set-state shall be the *last* call that $\mathcal{A}$ is allowed to make. However, in the model of [BH05], we have an additional security property once the state gets compromised. This recovering property is defined through the notion of *backward security* and more generally of *robustness*. In [BH05], a generator is backward secure if it starts with a compromised state and then recovers from its compromise. More generally, in [BH05], a generator is robust if it can recover from a state compromise that occurs at any time (not only at a last stage or in a first stage). Note that the definition of forward security is slightly different from Definition 3.3.1, from the security model of [BY03]. Here the security game starts with a known state, which becomes safe once a call to procedure good-refresh is done (and for which the flag corrupt = false), whereas in the security game of [BY03], the challenger generates first a random state. In both models, the state compromise through a call to get-state is done afterwards. In this sense, the notion of forward security is stronger here than in [BY03] because the adversary can choose the high entropy distribution from the family $\mathcal{H}$, whereas in [BY03] $\mathcal{A}$ has no control on it (not even on its distribution).

3.6.2 A Secure Construction

In [BH05] Barak and Halevi proposed a simple and elegant construction for a pseudo-random number generator with input. This construction (which we call BH) has a state $S \in \{0,1\}^n$, takes inputs $I \in \{0,1\}^p$ and outputs $R \in \{0,1\}^\ell$. The generator BH involves an $\mathcal{H}$-extractor (Definition 24) $\mathsf{Extract} : \{0,1\}^p \longrightarrow \{0,1\}^n$ and a $(t, \varepsilon_{\mathbf{G}})$-secure standard pseudo-random number generator $\mathbf{G} : \{0,1\}^n \longrightarrow \{0,1\}^{n+\ell}$ (Definitions 8 and 9). Note that the output length of the extractor Extract is equal to the size of the internal state of the generator BH (in accordance with the notations and Definition 24, we have $m = n$), and the input length of the generator $\mathbf{G}$ is equal to the size of the internal state of BH and its output length is the size of the internal state of BH added to the size of the output of BH. The refresh and next algorithms are given below, where $\mathbf{G}'$ denotes the truncation to the first n bits of $\mathbf{G}$.

- $\mathsf{refresh}(S, I) = \mathbf{G}'(S \oplus \mathsf{Extract}(I))$.
- $\mathsf{next}(S) = \mathbf{G}(S)$.

The security of the pseudo-random number generator with input BH is stated in the following theorem, where q_r denotes the upper bound on the number of calls to the refresh algorithm and q_n denotes the upper bound on the number of calls to the next algorithm.

Theorem 8 (Security of BH [BH05])**.** *Let* $\mathsf{Extract} : \{0,1\}^p \longrightarrow \{0,1\}^n$ *be an* $\mathcal{H}$*-extractor. The pseudo-random number generator with input* BH *is* $(t, q_r/2^n + q_n\varepsilon_{\mathbf{G}})$*-robust for the family* $\mathcal{H}$*.*

3.7 Leakage Resilient Stateful Pseudo-Random Number Generators

3.7.1 Security Models

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 secret or the next iteration of the generator). Yet a fundamental issue in the context of leakage-resilient cryptography is to define reasonable restrictions on the leakage functions. We propose to base our work on the following restrictions:

- **Only Computation Leaks.** From the axiom 'Only Computation Leaks' of Micali and Reyzin [MR04], one first assumption is to assume that only the data being manipulated in a computation can leak during this computation. That is, the adversary cannot learn information on a stored but not manipulated data as in [AGV09,DKL09]. This formalization is very classical [FPS12, YS13, ABF13] and close to the practical observations. From this consideration, it is possible (a) to split the cryptographic primitives into smaller blocks that leak independently on functions of their specific manipulated inputs and (b) to allow the adversary to choose a different leakage function for each small block. This model is referred to as *granular*. However note that in this model some leakage attacks are not captured, such as the cold-boot attack, where all memory contents leak information, even if they were never accessed.

- **Bounded Leakage per Iteration.** As most previous works [DP08, Pie09, YSPY10, FPS12,YS13,ABF13], the adversary can choose the polynomial time leakage functions with a restriction on the size of the output. Without this restriction, the adversary could choose the identity function and recover the entire secret state in one observation. Therefore, one second assumption is to bound the output length of the leakage functions with a parameter λ depending on the security parameter of the cryptographic primitives. Note that another choice has been made by Rivain and Prouff in [PR13]. They consider noisy leakage functions with a bound not on the output length but on the statistical distance between the distribution of the secret and the distribution of the secret given the knowledge of the leakage.

- **Non-Adaptive Leakage.** The third assumption is based on the practical observation whereby leakage functions completely depend on the inherent device. Another point of view is followed by some authors [DP08,Pie09] who give a stronger power to the adversary by authorizing adaptive leakage functions. The adversary is then allowed to adaptively choose the leakage function according to its current knowledge acquired from the previous invocations. Even if this model aims to be more general, this choice leads to unrealistic

scenarios since the adversary is then able to predict further steps of the algorithm. And as said above, in practice, the leakage function is related to the device and not on the previous computations. For these reasons, this work, as many others before [YSPY10, FPS12, YS13, ABF13], consider only non-adaptive leakage functions.

Let us now describe the leakage security of a stateful pseudo-random number generator $(\mathsf{key}, \mathsf{next})$, as in Definition 17. To model the potential leakage of sensitive information, we use leakage functions that we globally name f. Note that, as we mention earlier, the leakage is non-adaptive, therefore the leakage functions f are a parameter of the game: they are determined before the security game starts and not chosen by the adversary during the game.

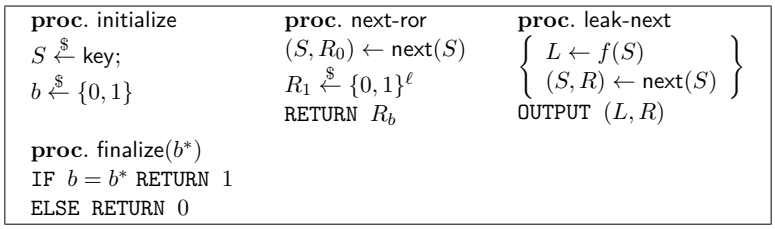

Figure 3.10 – Procedures in Security Game $\mathsf{LPR}(f)$

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 is described in Figure 3.10 and is similar as the one that defines the standard security of a stateful pseudo-random number generator, with the additional procedure leak-next. We denote the length of the output of the leakage functions with λ.

Definition 26. *A stateful pseudo-random number generator* $\mathbf{G} = (\mathsf{key}, \mathsf{next})$ *is* (t, ε, f)*-leakage resilient for the leakage function* f *if for any attacker* $\mathcal{A}$ *running in time at most* t*, the advantage of* $\mathcal{A}$ *in game* $\mathsf{LPR}(f)$ *is at most* ε*.*

It is important to note that the security targeted in this security model is not reachable for all leakage functions. Consider for example the leakage function $f : f(S) = \mathsf{next}(\mathsf{next}(S))$. Then with this leakage function, no construction can be proven secure. Therefore, a clear definition of the leakage function is a pre-requisite for any security statement.

3.7.2 Constructions

Several onstructions of leakage resilient stateful pseudo-random number generators have been proposed. We choose to present three of them, among all the existing ones, because we will use them as a basis to build a leakage resilient robust pseudo-random number generator with input, in Chapter 6. The first one is the construction from Yu, Standaert, Perreira and Yung [YSPY10]. The second one is a binary tree pseudo-random function introduced by Faust, Pietzrak and Schipper [FPS12] and the third one is a sequential stateful pseudo-random number generator with minimum public randomness proposed by Yu and Standaert [YS13]. We also focus on the presentation of their design and contrary to the previous presentations, we do not give precise statements and security bounds because we will use these constructions in a black-box way.

Construction from [YSPY10]. The first construction was proposed in [YSPY10]. It is illustrated in Figure 3.11. The stateful pseudo-random number generator $(\mathsf{key}, \mathsf{next})$ comes with an internal state made of three randomly chosen values: a secret key $S_0 \in \{0,1\}^n$ and two public parameters $(p_0, p_1) \in \{0,1\}^{2n}$. It uses a weak pseudo-random function $\mathbf{F}$, that uses the values

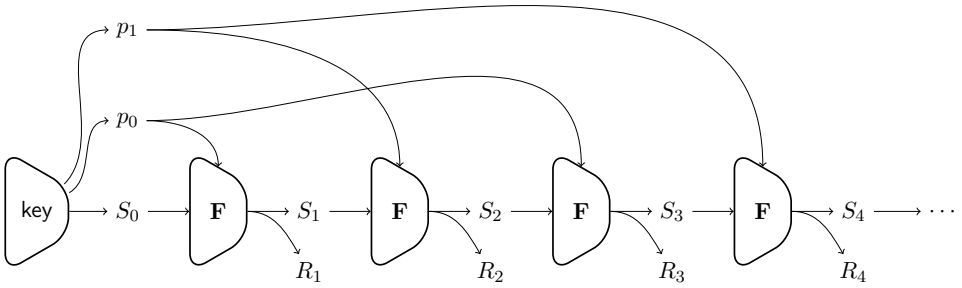

Figure 3.11 – Construction from [YSPY10]

p_0 and p_1 in an alternative way: at round i, the generators computes $\mathsf{next}(S_{i-1}) = (S_i, R_i) = \mathbf{F}(S_{i-1}, p_{i-1 \bmod 2})$.

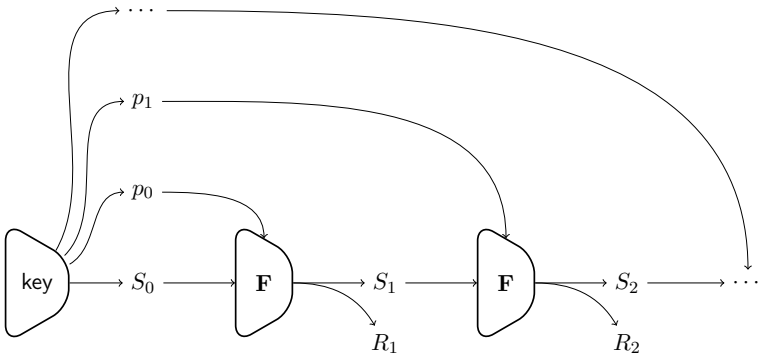

Figure 3.12 – Construction from [FPS12]

Construction from [FPS12]. The second solution was proposed in [FPS12]. It is illustrated in Figure 3.12. It is similar to the previous construction, with the difference that the key algorithm generates a secret state K_0 and a sequence of public values $p_0, p_1, \cdots$ that are used as input to the weak pseudo-random function $\mathbf{F}$. This construction was proposed as an extension of the previous one because they identified a subtle flaw in the proof of [YSPY10] and they therefore proposed a proven construction with independence between the inputs of the weak pseudo-random function $\mathbf{F}$. However, in the proposed construction, the needed large amount of public randomness prevents its practical use.

Construction from [YS13]. Yu and Standaert propose an extension of the previous constructions where the issue raised by the needed large amount of public randomness prevent from their practical use. The stateful pseudo-random number generator comes with an internal state made of two randomly chosen values: a secret key $S_0 \in \{0,1\}^\mu$ and a public parameter $\mathsf{seed} \in \{0,1\}^\mu$. The construction is made of two stages. In the upper stage, a (non leakage-resilient) pseudo-random function $\mathbf{F}' : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ is processed in counter mode to expand seed into uniformly random values $p_0, p_1, \ldots$. In the lower stage, a (non leakage-resilient) pseudo-random function $\mathbf{F}\{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^{2\mu}$ generates outputs R_i and updates the secret S_i so it is never used more than twice with the public values p_{i-1} . This two-stage construction is illustrated in Figure 3.13.

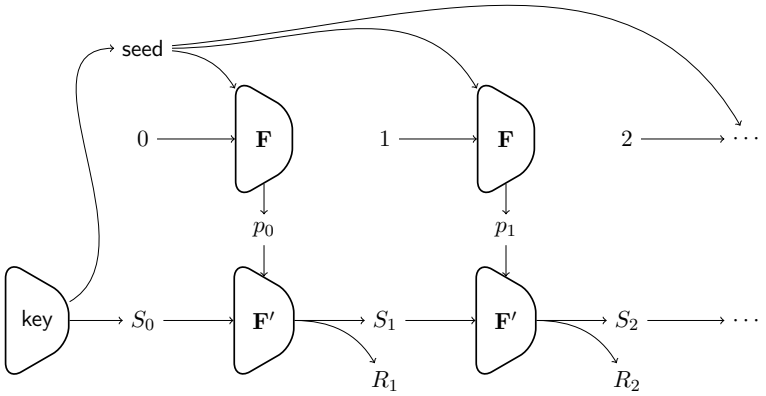

Figure 3.13 – Construction from [YS13]

3.8 Analysis

In this section, we summarize the features and differences of the security models presented in this chapter.

We present the different security properties seen in this chapter with Table 3.1. The security properties are given (CIA, CSA, KKA, BY-FWD, BST-RES, BH-ROB, LPR) and for each property, we precise (a) the pseudo-random number generator definition (b) the attacker capabilities (c) if there exists a secure construction and its security parameters. Note that [KSWH98, Gut98] do not give a secure construction.

Table 3.1 – Security Properties of Pseudo-Random Number Generators

Pseudo-Random Number Generator Definition	**Security Property**	**Attacker Capabilities**	**Construction**	
			G / F	Extract
$S \leftarrow \mathsf{key}$ $(S', R) \leftarrow \mathsf{next}(S)$	BY-FWD	next-ror, get-state	✖	
$(K, S) \leftarrow \mathsf{key}()$ $(S', R) \leftarrow \mathsf{next}(S, K, I)$	CIA	getinput, get-state, setinput	✖	
	CSA	getinput, get-state, set-state	✖	
	KKA	getinput, get-key	✖	
$\mathsf{Extract}(\mathsf{seed}, I) \leftarrow (\mathsf{seed}, I)$	BST-RES(τ)	$\mathcal{F}$		✖
$S' \leftarrow \mathsf{refresh}(S, I)$ $(S', R) \leftarrow \mathsf{next}(S)$	BH-ROB($\mathcal{H}$)	good-refresh, bad-refresh	✖	✖
		get-state, next-ror	✖	✖
$S \leftarrow \mathsf{key}$ $(S', R) \leftarrow \mathsf{next}(S)$	LPR(f)	next-ror, leak-next	✖	

✖: The construction involves a standard pseudo-random number generator **G** and / or a pseudo-random function **F** and / or a randomness extractor Extract.

We give now a comparison between these security notions.

State compromise. The first important feature of [KSWH98, Gut98] is the modelling of a pseudo-random number generator with input as a stateful algorithm. This modelling is also used in [BY03, DHY02, BH05]. The choice of stateful algorithms actually allows to define an adversary that interacts with the generator and can compromise its internal state: in all these

models, the adversary $\mathcal{A}$ has access to the current value of the internal state S with a procedure named get-state.

The response to state compromise, however, is specific to each model. In [Gut98], it is not considered that this attack should be taken into account in the design of the pseudo-random number generator, but countermeasures should be implemented by its environment. However, in [KSWH98], it is taken into account, and therefore a pseudo-random number generator should be protected against it by design. An adversary that gets access to the state S but not to the collected randomness, and an adversary that gets the collected randomness but not to the state S, should both be unable to get information about the next state. The same protection is also proposed in [DHY02], although the pseudo-random number generator definition is a bit different, as it involves a third component named key. The main idea that is implicit here is that a generator is not protected against a *joint* compromise of the internal state and of the randomness source, but only against a compromise of one *or* another.

In [KSWH98], an adversary that gets access to the state S should not be able to recover past outputs. This protection, named *Forward Security*, is formalized in [BY03], although the compromise of the randomness source is not considered here. Forward security is also captured in [BH05], in a similar way than in [BY03]: attacker $\mathcal{A}$ has access to two procedures get-state and set-state that allow her to get or to set the content of the internal state. As in [BY03], the call to either get-state or set-state shall be the *last* call that $\mathcal{A}$ is allowed to make. However, in [BY03], no additional security property is required once the state gets compromised and the model does not define how the generator shall recover from a compromise. This recovering property is defined in [BH05] through the notion of *backward security* and more generally through the notion of *robustness*. In [BH05], a pseudo-random number generator with input is backward secure if it starts with a compromised state and then recovers from its compromise. More generally, in [BH05], a pseudo-random number generator is robust if it can recover from a state compromise that occurs at any time (not only at a last stage or in a first stage). Similarly, in [DHY02], a compromise of the state S can be 'repaired' with the use of a non compromised auxiliary input while a compromise of the key can not be repaired. This recovering behavior, which is the starting point of *backward security*, is therefore not captured by [DHY02].

The model of [BST03] does not concern stateful pseudo-random number generators. Hence, the state compromise is not considered in the security model and it is assumed that the entropy source produces random bits at a high rate, as the only operation that is done is the extraction. Once extracted, the output can be used directly by a consuming application or by a standard pseudo-random number generator, however this operation is not described. The full description of a pseudo-random number generator with input with both an extraction phase and a generation phase and with a partial control on the distribution of the input distribution is give [BH05]. Also note that the construction proposed in [BH05] composes the randomness extractor with a stateful pseudo-random number generator; this construction benefits from the *forward security* of the stateful pseudo-random number generator, and the resilience of the pseudo-random number generator with input.

In [KSWH98], a pseudo-random number generator with input should enforce complete renewal of the internal state. This feature is also considered in [BH05], where there is a procedure named good-refresh by which the internal state of the generator is refreshed with a 'high entropy' source. Once refreshed with such a source, the generator is considered 'uncompromised' and output can be generated. Moreover, the model of [BH05] also requires that once refreshed with such a source, the internal state is pseudo-random, ensuring that is it completely refreshed with new entropy.

Source compromise. The security guidelines of [KSWH98, Gut98] model the potential com-

promise of the source of randomness: adversary $\mathcal{A}$ has access to two procedures getinput and setinput. The model in [DHY02] also considers a source compromise, but in a different way. In [DHY02], the security model assumes the existence of an entropy pool in which entropy is accumulated and that is used as input for state generation. However, the model does not capture the potentially adversarial inputs that may be used for this generation. The only adversarial procedures, also named getinput and setinput in [DHY02], are related to the compromise of an *auxiliary input* that is also used to generate the internal state. Hence, in [DHY02], the *complete* compromise of the randomness source is not taken into account.

Source compromise is not taken into account in [BY03]. In this model, the potentially adversarial inputs that could be used to compromise the generator are not described. It is assumed in [BY03] that the state is first properly generated will algorithm key. A recovery mechanism would be to enforce regular applications of algorithm key, however, it implies that a trusted source of randomness is available, from which extraction can be proceeded. This operation is not captured in [BY03], as it is by the guidelines of [KSWH98, Gut98] and more formally in [BST03, BH05]. This is obvious from the context of [BY03], as the model does not consider pseudo-random number generator with inputs, however it is implicit that the first generation of the state with algorithm key can not be compromised.

Source compromise is considered in [BST03], where the adversary $\mathcal{A}$ can choose a finite family of distributions $\mathcal{F}$. While in the security models of [KSWH98, Gut98], adversary $\mathcal{A}$ has access to a setinput procedure that allows her to directly set the value of the input that is collected by the random number generator, here the adversary has implicitly access to a so-called 'set-distribution' or 'setfamily' procedure, that allows $\mathcal{A}$ to set the high entropy distribution family $\mathcal{F}$. This procedure is not explicitly defined in the model of [BST03], however, as the family of distributions $\mathcal{F} = \{\mathcal{D}_1, \cdots, \mathcal{D}_{2^t}\}$, such that $\mathbf{H}_\infty(\mathcal{D}_i) \geq k$ is given as input for all procedures, we can consider that such procedure exists. In fact, in the model of [BH05], a procedure named good-refresh is defined, that allows adversary $\mathcal{A}$ to set a 'high entropy' distribution, which plays the same role as the underlying 'setdistribution' procedure of [BST03]. However, the model of [BST03] only considers the case of 'high-entropy' distribution, while the model of [BH05] also consider adversarially controlled inputs, with a procedure named bad-refresh, that is similar to the procedure setinput. Finally in [BST03, BH05] the need for a randomness extractor is clearly stated, while in [KSWH98, Gut98], this need is not identified.

Pseudo-Random Number Generators with Input Definition. The definition of a pseudo-random number generator with input in [DHY02] does not contain any refresh algorithm that could model input collection to continuously update the state S and the key S. The update is implicitly contained in the next algorithm. A clear separation between refresh algorithm and next algorithm is the basis of the robustness security game of [BH05].

Entropy Accumulation. The model of [BH05] (as for the model of [DHY02] with the setinput procedure) only considers that the adversary can either call the good-refresh procedure, which must produce an input I of high entropy, or call the bad-refresh procedure with an arbitrary, maliciously specified input I^*. Informally, the call to bad-refresh should not compromise the generator whenever the compromised flag corrupt = false, while the call to good-refresh should result in an immediate "recovery", and always resets corrupt = true. In real implementations, however, entropy can be accumulated slowly (and maliciously!), as opposed to in "one shot" (or "delayed" by calls to bad-refresh). This implies that once the generator is compromised, the recovering process should ensure that enough entropy is accumulated *before* outputs are generated. Note that this behavior is indeed implemented in practical generators, such as the Linux generators `dev/random` and `dev/urandom`, which place a lot of (heuristic) effort in trying to achieve this property.

Importance of setup. As we mentioned, the model of [BH05] did not have an explicit setup algorithm to initialize public parameters seed. Instead, they assumed that the required randomness extractor Extract in their construction is good enough to extract nearly ideal randomness from any high-entropy distribution I output by the good-refresh procedure. Ideally, we would like to make no other assumptions about I except its min-entropy. Unfortunately, no deterministic extractor is capable to simultaneously extract good randomness from *all* efficiently samplable high-entropy distributions (See Lemma 2). As noticed, the choice by [BH05] is to restrict the family of permitted high-entropy distributions I. Hence a key strengthening of the model will be the use of strong randomness extractors, as in Definition 7, that allow a *public* and reusable parameter seed and extract from arbitrary randomness sources. One condition to use this class of extractors is that the parameter seed is independent from the source.

Leakage Resilient Pseudo-Random Number Generators with Input. The security model of leakage resilience concerns stateful pseudo-random number generators. However, as pointed in [BH05], a large class of generators are implemented as pseudo-random number generators with inputs, that are continuously refreshed with new inputs collected from their environment. The potential leakage of the internal state for such generators therefore needs to be formalized.

Chapter 4

Robustness of Pseudo-random Number Generators with Inputs

4.1 Model Description

In this section we give a syntactic formalization and security definitions for pseudo-random number generator with input. All definitions and theorems from this chapter are from [DPR+13].

Recall that we termed 'pseudo-random number generator with input' to refer that the generator is refreshed periodically with new inputs, as described informally in Section 2.8.3. As explain in Chapter 3, the security models of [DHY02, BST03, BH05] also concern pseudo-random number generator with input.

Our definition of a pseudo-random number generator with input requires that, in addition to the usual refresh and next algorithm, an algorithm named setup is set, whose objective is to generate a parameter seed (which will be the seed of a randomness extractor). As noted in Section 3.8, this algorithm is necessary to describe completely the generator operations, as it shall naturally involve a randomness extractor. Furthermore, we want the parameter seed to be public because the security of our schemes shall not rely on the secrecy of any parameter (if this secrecy is guaranteed, one can use a standard pseudo-random number generator).

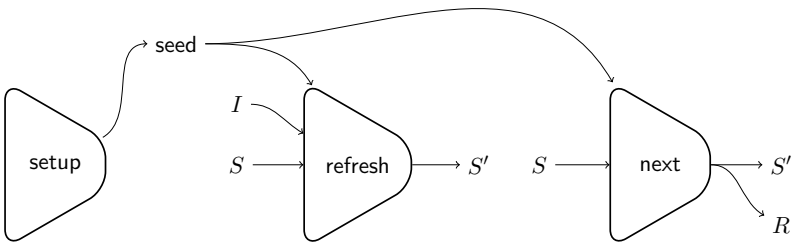

Figure 4.1 – Pseudo-Random Number Generator with Input [DPR+13]

Definition 27 (Pseudo-Random Number Generator with Input [DPR+13])**.** *A Pseudo-Random Number Generator with Input is a triple of algorithms* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *and a quadruple* $(s, n, \ell, p) \in \mathbb{N}^4$ *where:*

- setup*: it is a probabilistic algorithm that outputs some public parameters* $\mathsf{seed} \in \{0,1\}^s$ *for the generator.*

- refresh*: it is a deterministic algorithm that, given* seed $\in \{0,1\}^s$*, a state* $S \in \{0,1\}^n$ *and an input* $I \in \{0,1\}^p$*, outputs a new state* $S' \leftarrow \mathsf{refresh}(S, I) = \mathsf{refresh}(\mathsf{seed}, S, I) \in \{0,1\}^n$.

- next*: it is a deterministic algorithm that, given* seed $\in \{0,1\}^s$ *and a state* $S \in \{0,1\}^n$*, outputs a pair* $(S', R) \leftarrow \mathsf{next}(S) = \mathsf{next}(\mathsf{seed}, S)$ *where* $S' \in \{0,1\}^n$ *is the new state and* $R \in \{0,1\}^\ell$ *is the output.*

The integer s *is the* seed length*,* n *is the* state length*,* ℓ *is the* output length *and* p *is the* input length *of* $\mathcal{G}$.

Note that to simplify the algorithm description, we will omit the parameter seed when its definition is clear from the context. Recall that the previous models were based on the use of a *resilient* randomness extractor (Definition 5). As explained in Section 2.6, this class of extractor restricts the use of a bounded size finite family of randomness sources. Furthermore, we want that the random parameter seed is made public once generated. As noted in Section 2.6, we mainly have two possibilities:

1. We assume that independence between the seed and the randomness source can not be ensured. One solution is to restrict the randomness sources to use a resilient extractor: this is the solution proposed in [BST03, BH05]. One second solution would be to restrict the adversary capabilities and use *seed dependent* extractors, as in Definition 6.

2. We choose not to restrict the randomness neither the adversary capabilities. As noted in Section 2.6, as we also want that seed is public, one solution is to use *strong* extractors, as in Definition 7, as soon as independence between the seed and the randomness source can be ensured. Our model relies on this assumption.

In our adversarial model for pseudo-random number generator with input, we consider that the adversary $\mathcal{A}$ can partially control the inputs that are used to refresh the internal state of the generator. In addition, we also need that independence between the seed and the input distribution is guaranteed. We therefore propose to split the adversary into two entities: the *adversary* $\mathcal{A}$ whose task is (intuitively) to distinguish the outputs of the generator from random, and the *distribution sampler* $\mathcal{D}$ whose task is to produce inputs $I_1, I_2, \ldots$, which have high entropy *collectively*, but somehow help $\mathcal{A}$ in breaking the security of the generator. The distribution sampler aims at modeling potentially adversarial environment (or 'nature') where the generator operates. To ensure independence of the randomness sources with seed, we will require that the distribution sampler is set independently of seed. Once $\mathcal{D}$ is set, the adversary $\mathcal{A}$ has access to seed. This separation between $\mathcal{A}$ and $\mathcal{D}$ allows to clarify 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, we enforce independence between seed and the 'part' of the adversary that has control on the randomness source and we let the 'other part' having access to seed. The above discussion justifies Definition 28.

Definition 28 (Distribution Sampler)**.** *Let* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *and* $(s, n, \ell, p) \in \mathbb{N}^4$ *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* σ*, outputs a tuple* (σ', I, γ, z) *where:*

- σ' *is the new state for* $\mathcal{D}$.

- $I \in \{0,1\}^p$ *is the next input for the* refresh *algorithm.*

- γ *is some* fresh entropy estimation *of* I*, as discussed below.*

- z *is the* leakage *about* I *given to the adversary* $\mathcal{A}$.

We denote by q_r the upper bound on number of executions of $\mathcal{D}$ in our security games, and say that $\mathcal{D}$ is legitimate *if*

$$\mathbf{H}_\infty(I_k \mid I_1, \ldots, I_{k-1}, I_{k+1}, \ldots, I_{q_r}, z_1, \ldots, z_{q_r}, \gamma_1, \ldots, \gamma_{q_r}) \geq \gamma_k$$

for all $k \in \{1, \ldots, q_r\}$ where $(\sigma_k, I_k, \gamma_k, z_k) = \mathcal{D}(\sigma_{k-1})$ for $k \in \{1, \ldots, q_r\}$ and $\sigma_0 = 0$.

We now explain the reason for explicitly requiring $\mathcal{D}$ to output the entropy estimate γ_k. Most complex generators, for example the Linux generators `dev/random` and `dev/urandom`, are worried about the situation where the system might enter a prolonged state where no new entropy is inserted in the system. Correspondingly, such generators typically include some ad hoc *entropy estimation procedure* E whose goal is to block the generator from generating an output R until the state has not accumulated enough entropy γ^* (for some entropy threshold γ^*). Unfortunately, it is well-known that even approximating the entropy of a given distribution is a computationally hard problem [SV03]. This means that if we require a pseudo-random number generator with input $\mathcal{G}$ to explicitly come up with such a procedure E, we are bound to either place some significant restrictions (or assumptions) on $\mathcal{D}$, or rely on some hoc and non standard assumptions. Indeed, as part of this work we will demonstrate some attacks on the entropy estimation of the Linux generators, illustrating how hard it is to design a sound entropy estimation procedure E.

Also, observe that the design of E is anyway completely *independent* of the mathematics of the actual refresh and next procedures, meaning that the latter can and *should* be evaluated independently of the 'accuracy' of E.

In the security definition, we do not insist on any 'entropy estimation' procedure as a mandatory part of the design of a pseudo-random number generator with input, instead, we chose to place the burden of entropy estimations on $\mathcal{D}$ *itself*, which allows us to concentrate on the *provable* security of the refresh and next procedures. In particular, in our security definition we will not attempt to verify if $\mathcal{D}$'s claims are accurate, but will only require security when $\mathcal{D}$ is *legitimate*, as in Definition 28. Equivalently, we can think that the entropy estimations γ_k come from the entropy estimation procedure E (which is now 'merged' with $\mathcal{D}$), but only provide security assuming that E is correct in this estimation (which we know is hard in practice). Finally, in the security definition,

- The entropy estimates γ_k will only be used in security definitions, but not in any of the actual generator operations (which will only use the "input part" I returned by $\mathcal{D}$)
- We do not insist that a legitimate $\mathcal{D}$ can perfectly estimate the fresh entropy of its next sample I_k, but only provide a *lower bound* γ_k that $\mathcal{D}$ is "comfortable" with. For example, $\mathcal{D}$ is free to set $\gamma_k = 0$ as many times as it wants and, in this case, can even choose to leak the entire I_k to $\mathcal{A}$ via the leakage z_k. Note that setting $\gamma_k = 0$ corresponds to the bad-refresh(I_k) oracle in the earlier modelling of [BH05], described in Section 3.6, which is not explicitly provided in our new model.
- We allow $\mathcal{D}$ to inject new entropy γ_k as slowly (and maliciously!) as it wants, but will only require security when the counter c keeping track of the current "fresh" entropy in the system (intuitively, "fresh" refers to the new entropy in the system since the last state compromise) crosses some entropy threshold γ^* (since otherwise $\mathcal{D}$ gave "no reason" to expect any security).

As seen in Chapter 3, four security notions for a pseudo-random number generator with input have been proposed: *resilience* (RES), *forward security* (FWD), *backward security* (BWD) and

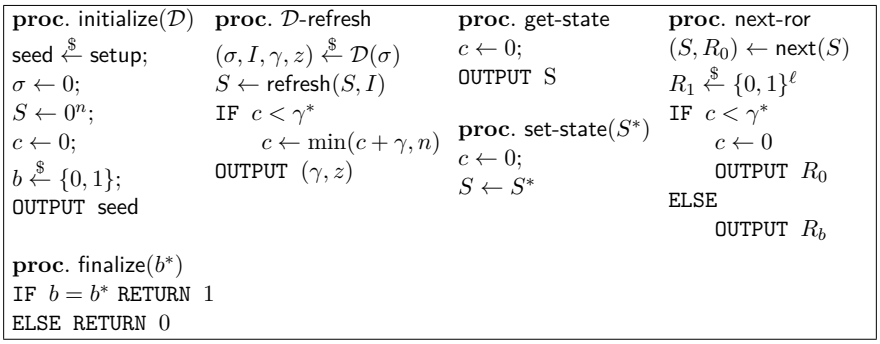

Figure 4.2 – Procedures in Security Games RES(γ^*), FWD(γ^*), BWD(γ^*), ROB(γ^*)

robustness (ROB), with the latter being the strongest notion among them. We now define the analogues of these notions in our stronger adversarial model, later comparing our model with the prior model of [BH05]. Each of the games below is parametrized by some parameter γ^* which is part of the claimed generator security, and intuitively measures the minimal "fresh" entropy in the system when security should be expected. In particular, minimizing the value of γ^* corresponds to a stronger security guarantee. When γ^* is clear from the context, we omit it for the game description (e.g., write ROB instead of ROB(γ^*)).

All four security games (RES(γ^*), FWD(γ^*), BWD(γ^*), ROB(γ^*)) are described using the game playing framework presented in Section 2.4, and share the same initialize and finalize procedures in Figure 4.2. As we mentioned, our overall adversary is modelled via a pair of adversaries $(\mathcal{A}, \mathcal{D})$, where $\mathcal{A}$ is the actual adversary and $\mathcal{D}$ is a stateful distribution sampler. We already discussed the distribution sampler $\mathcal{D}$, so we turn to the adversary $\mathcal{A}$, whose goal is to guess the correct value b picked in the initialize procedure, which also returns to $\mathcal{A}$ the public value seed, and initializes several important variables: corruption flag corrupt, "fresh entropy counter" c, state S and sampler's $\mathcal{D}$ initial state σ. In each of the games (RES, FWD, BWD, ROB), $\mathcal{A}$ has access to several oracles depicted in Figure 4.2. We briefly discuss these oracles:

- $\mathcal{D}$-refresh. This is the key procedure where the distribution sampler $\mathcal{D}$ is run, and where its output I is used to refresh the current state S. Additionally, one adds the amount of fresh entropy γ to the entropy counter c. The values of γ and the leakage z are also returned to $\mathcal{A}$. We denote by q_r the number of times $\mathcal{A}$ calls $\mathcal{D}$-refresh (and, hence, $\mathcal{D}$), and notice that by our convention (of including oracle calls into run-time calculations) the total run-time of $\mathcal{D}$ is implicitly upper bounded by the run-time of $\mathcal{A}$.

- next-ror. This procedure provides $\mathcal{A}$ with either the real-or-random challenge (provided that $c \geq \gamma^*$) or the true generator output. As a small subtlety, a "premature" call to next-ror before c crosses the γ^* resets the counter c to 0, since then $\mathcal{A}$ might learn something non-trivial about the (low-entropy) state S in this case. We denote by q_n the total number of calls to next-ror and get-next.

- get-state/set-state. These procedures provide $\mathcal{A}$ with the ability to either learn the current state S, or set it to any value S^*. In either case c is reset to 0.We denote by q_s the total number of calls to get-state and set-state.

We can now define the corresponding security notions for pseudo-random number generator with input. In the sequel we denote the 'resources' of $\mathcal{A}$ by $T = (t, q_r, q_n, q_s)$. We also use the integers k and j as follows:

- The integer k is used to identify the successive outputs $(\sigma_k, I_k, \gamma_k, z_k)$ of the distribution sampler $\mathcal{D}$ and therefore the successive inputs I_k used to update the internal state of the generator,
- The integer j is used to identify the successive states S_j of the generator, when updated with the refresh algorithm.

Definition 29 (Security of pseudo-random number generator with input [DPR+13])**.** *A pseudo-random number generator with input* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *is called* $(T = (t, q_r, q_n, q_s), \gamma^*, \varepsilon)$-robust (resp. resilient, forward-secure, backward-secure)*, 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* $\mathsf{ROB}(\gamma^*)$ *(resp,* $\mathsf{RES}(\gamma^*)$*,* $\mathsf{FWD}(\gamma^*)$*,* $\mathsf{BWD}(\gamma^*)$*) is at most* ε*, where:*

- $\mathsf{ROB}(\gamma^*)$ *is the unrestricted game where* $\mathcal{A}$ *is allowed to make the above calls.*
- $\mathsf{RES}(\gamma^*)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* get-state/set-state *(i.e.,* $q_s = 0$*).*
- $\mathsf{FWD}(\gamma^*)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* set-state *and a single call to* get-state *(i.e.,* $q_s = 1$*) which is the very last oracle call* $\mathcal{A}$ *is allowed to make.*
- $\mathsf{BWD}(\gamma^*)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* get-state *and a single call to* set-state *(i.e.,* $q_s = 1$*) which is the very first oracle call* $\mathcal{A}$ *is allowed to make.*

Hence, (a) resilience protects the security of the generator when not corrupted against arbitrary distribution samplers $\mathcal{D}$, (b) forward security protects past generator outputs in case the state S gets compromised, (c) backward security security ensures that the generator can successfully recover from state compromise, provided enough fresh entropy is injected into the system, (d) robustness ensures arbitrary combination of the above. Hence, robustness is the strongest and the resilience is the weakest of the above four notions. In particular, all our provable constructions will satisfy the robustness notion, but we will use the weaker notions to better point some of our attacks.

Examples of the entropy traces of the counter c for the procedures defined for the security game ROB are provided in Figure 4.3. Note that we illustrated two next-ror calls, the first one where $c \geq \gamma^*$ and the second one where $c < \gamma^*$.

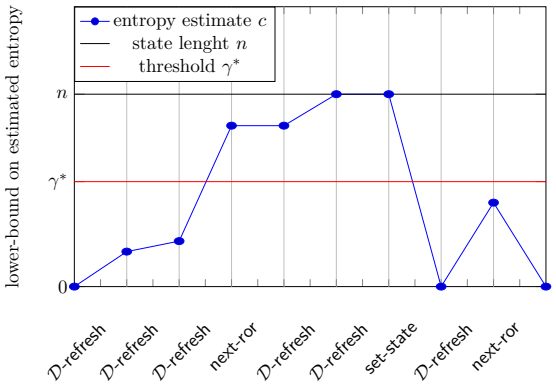

Figure 4.3 – Entropy Estimates in $\mathsf{ROB}(\gamma^*)$

Comparison with [BH05]. As described, this security models complements the previous work of [BH05] in two ways:

1. It clearly captures the notion of *entropy accumulation*.
2. It clarifies the need of independence between the public parameter seed and the source of randomness.

This security models also allows to define precisely the notion of *backward security* which is close to the *recovering security* described in the next section. In [BH05], an immediate recovery occurs any time one call is done to procedure good-refresh, which should be seen as an extreme case of the new proposed security model.

In [BH05], Barak and Halevi also insisted that the state S is indistinguishable from random once corrupt = false. While true in their specific construction, we think that demanding this property is simultaneously too restrictive and also not very well motivated as the *mandatory* part of a general security *definition*.

For example, if one considers a generator with an internal state S that includes a (never random) Boolean flag which keeps track if the last call to the generator was made to the next procedure. In particular, looking at the analysis of [BH05], the (truncated) generator $\mathbf{G}'$ inside the refresh procedure is only needed to ensure the state pseudo-randomness of their construction. In other words, if one drops (only the) state pseudo-randomness from the BH model, *the "Simplified* BH*" construction is already robust in their model*. Motivated by this, we will present a strong attack on the simplified BH construction, for *any* extractor Extract and any standard pseudo-random number generator $\mathbf{G}$.

We consider a 'simplified' construction (that we denote after the 'Simplified BH' construction, as it is derived from the robust construction described in Section 3.6.2). This construction also involves a randomness extraction function $\mathsf{Extract} : \{0,1\}^p \longrightarrow \{0,1\}^n$ and a standard pseudo-random number generator $\mathbf{G} : \{0,1\}^n \longrightarrow \{0,1\}^{n+\ell}$. As for the initial construction of [BH05], we do not define an explicit setup algorithm, and the refresh and next algorithms are given below:

- $\mathsf{refresh}(S, I) = S \oplus \mathsf{Extract}(I)$
- $\mathsf{next}(S) = \mathbf{G}(S)$

Hence the only difference between the initial construction of [BH05] is that we dropped the (truncated) generator $\mathbf{G}'$ inside the refresh procedure that is only needed to ensure the state pseudo-randomness of their construction. Consider now the simplified version of the robustness security model described in Figure 4.4, that we name the 'Simplified ROB' model. This security model is the same as the robustness security model described in Section 3.6, except that the set-state procedure is not used to challenge the adversary on its capability to distinguish the state from random.

proc. initialize $S_0 \leftarrow 0^n$; corrupt $\leftarrow$ true; $b \xleftarrow{\$} \{0,1\}$; parse $\mathcal{H}$ as $\{\mathcal{D}_i\}_{i\in I}$ seed $\xleftarrow{\$} \{0,1\}^n$; OUTPUT seed **proc**. finalize(b^*) IF $b = b^*$ RETURN 1 ELSE RETURN 0	**proc**. good-refresh(i) $x \xleftarrow{\$} \mathcal{D}_i$; $S_0 \leftarrow \mathsf{refresh}(S_0, x)$; corrupt $\leftarrow$ false; **proc**. bad-refresh(x) IF corrupt = true $S_0 \leftarrow \mathsf{refresh}(S_0, x)$; ELSE $\perp$	**proc**. set-state(S^*) corrupt $\leftarrow$ true $S \leftarrow S^*$	**proc**. next-ror $(S, R_0) \leftarrow \mathsf{next}(S)$ IF corrupt = true, RETURN R_0 ELSE $R_1 \xleftarrow{\$} \{0,1\}^\ell$ RETURN R_b

Figure 4.4 – Procedures in Security Game 'Simplified ROB($\mathcal{H}$)'

Following the proof of [BH05], one can prove that the 'Simplified BH' construction is robust in the 'Simplified ROB' model. Motivated by this, we give a very strong attack on the 'Simplified BH' construction in our stronger model, for *any* extractor Extract and generator $\mathbf{G}$. This already illustrates the main difference between our models in terms of entropy accumulation.

Consider the following very simple distribution sampler $\mathcal{D}$. At any time period, it simply sets $I = \alpha^p$ (meaning bit α concatenated p times) for a fresh and random bit α, and also sets entropy estimate $\gamma = 1$ and leakage $z = \varnothing$. Clearly, $\mathcal{D}$ is legitimate, as the min-entropy of I is 1, even conditioned on the past and the future. Hence, for any entropy threshold γ^*, the simplified BH construction must regain security after γ^* calls to the $\mathcal{D}$-refresh procedure following a state compromise. Now consider the following simple adversary $\mathcal{A}$ attacking the backward security (and, thus, robustness) of the 'Simplified BH' construction. It calls set-state(0^n), and then makes γ^* calls to $\mathcal{D}$-refresh followed by many calls to next-ror. Let us denote the value of the state S after j calls to $\mathcal{D}$-refresh by S_j, and let $Y(0) = \mathsf{Extract}(0^p)$, $Y(1) = \mathsf{Extract}(1^p)$. Then, recalling that $\mathsf{refresh}(S, I) = S \oplus \mathsf{Extract}(I)$ and $S_0 = 0^n$, we see that $S_j = Y(\alpha_1) \oplus \ldots \oplus Y(\alpha_j)$, where $\alpha_1 \ldots \alpha_j$ are random and independent bits. In particular, at any point of time there are only two possible values for S_j: if j is even, then $S_j \in \{0^n, Y(0) \oplus Y(1)\}$, and, if j is odd, then $S_j \in \{Y(0), Y(1)\}$. In other words, despite receiving γ^* random and independent bits from $\mathcal{D}$, the refresh procedure failed to accumulate more than 1 bit of entropy in the final state $S^* = S_{\gamma^*}$. In particular, after γ^* calls to $\mathcal{D}$-refresh, $\mathcal{A}$ can simply try both possibilities for S^* and easily distinguish real from random outputs with advantage arbitrarily close to 1 (by making enough calls to next-ror).

This shows that the 'Simplified BH' construction is *never* backward secure in our model, despite being secure in the 'Simplified ROB' model.

4.2 Recovering and Preserving Security

We define two properties of a pseudo-random number generator with input which are intuitively simpler to analyze than the full robustness security. We show that these two properties, taken together, imply robustness.

Recovering 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 γ^*. The *recovering* process should make the bit b involved in the next-ror procedure indistinguishable: when the internal state is considered as safe, 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 Figure 4.5.
The security game RECOV is described as follow, with an adversary $\mathcal{A}$, a sampler $\mathcal{D}$, and bounds q_r, γ^*:

1. The challenger generates a seed seed $\xleftarrow{\$}$ setup and a bit $b \xleftarrow{\$} \{0,1\}$ uniformly at random. It sets $\sigma_0 = 0$ and for $k = 1, \ldots, q_r$, 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 $\gamma_1, \ldots, \gamma_{q_r}$ and $z_1, \ldots, z_{q_r}$ to the adversary.

2. The adversary gets access to an oracle getinput which on each invocation increments $k := k + 1$ and outputs I_k.

proc. initialize($\mathcal{D}$)
seed $\xleftarrow{\$}$ setup;
$\sigma_0 \leftarrow 0$;
$b \xleftarrow{\$} \{0,1\}$;
FOR $k = 1$ TO q_r DO
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
END FOR
$k \leftarrow 0$;
OUTPUT seed, $(\gamma_k, z_k)_{k=1,\ldots,q_r}$

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. getinput
$k \leftarrow k + 1$
OUTPUT I_k

proc. set-state(S^*)
$S \leftarrow S^*$
$c \leftarrow 0$

proc. $\mathcal{D}$-refresh
$k \leftarrow k + 1$;
$S = \mathsf{refresh}(S, I_k)$;
IF $c < \gamma^*$,
  $c = \min(c + \gamma_k, n)$

proc. next-ror
$(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$
$(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$
RETURN $(S^{(b)}, R^{(b)})$, $(I_{k+1}, \ldots, I_{q_r})$

Figure 4.5 – Procedures in Security Game $\mathsf{RECOV}(q_r, \gamma^*)$

3. At some point the adversary $\mathcal{A}$ calls procedure set-state: she sets a chosen internal state $S^* \in \{0,1\}^n$. She then chooses an integer d such that $k+d \leq q_r$ and $\gamma_{k+1}+\cdots+\gamma_{k+d} \geq \gamma^*$, then calls $\mathcal{D}$-refresh d times: this procedure updates the state $S := \mathsf{refresh}(S, I_{k+j})$ and updates $c \leftarrow c + \gamma_k$ sequentially.

4. Eventually, the challenger sets $(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$ and generates $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$. It then gives $(S^{(b)}, R^{(b)})$ to the adversary, together with the next inputs $I_{k+1}, \ldots, I_{q_r}$ (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 30 (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* $\mathsf{RECOV}(q_r, \gamma^*)$ *is at most* ε*.*

Preserving Security. This security notion considers a safe internal state. After several calls to $\mathcal{D}$-refresh with known (and even chosen) inputs, the internal state should remain safe. 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 safe, 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 Figure 4.6.
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 $S \xleftarrow{\$} \{0,1\}^n$, a seed seed $\leftarrow$ setup, and a bit $b \xleftarrow{\$} \{0,1\}$ 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.

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

Figure 4.6 – Procedures in Security Game PRES

3. Eventually, the challenger sets $(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$ and generates $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$. 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 31 (Preserving Security)**.** *A pseudo-random number generator with input* (setup, refresh, next) *is said* (t, ε)*-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* ε*.*

We now show that, taken together, recovering and preserving security notions imply the full notion of robustness.

Theorem 9. *If a pseudo-random number generator with input* (setup, refresh, next) *has both* $(t, q_r, \gamma^*, \varepsilon_r)$*-recovering security and* (t, ε_p)*-preserving security, then it is* $((t', q_r, q_n, q_s), \gamma^*, q_n(\varepsilon_r + \varepsilon_p))$*-robust where* $t' \approx t$*.*

Proof. We will refer to the adversary's queries to next-ror oracle in the robustness game as "next queries". We assume that the adversary makes exactly q_n of them. We say that a next query is *uncompromised* if $c \geq \gamma^*$ during the query, and we say it is *compromised* otherwise.

We prove the robustness of GEN by reduction (a) to its recovering security and (b) to its preserving security.

We partition the uncompromised next queries into two subcategories: *preserving* and *recovering*. We say that an uncompromised next query is *preserving* if $c \geq \gamma^*$ throughout the entire period between the previous next query (if there is one) and the current one. Otherwise, we say that an uncompromised next query is *recovering*. With any recovering next query, we can associate a corresponding *most recent entropy drain (*mRED*)* query which is the most recent query to either get-state, or to set-state, or to a compromised next-ror that precedes the current next query. An mRED query must set the cumulative entropy estimate to $c = 0$. Moreover, with any recovering next query, we associate a corresponding sequence of *recovering samples* $\bar{I} = (I_{k+1}, \ldots, I_{k+d})$ which are output by all the calls to the $\mathcal{D}$-refresh oracle that precede the recovering next query, but follow the associated mRED query. It is easy to see that any such sequence of recovering samples $\bar{I}$ must satisfy the entropy requirements $\sum_{j=k+1}^{k+d} \gamma_j \geq \gamma^*$ where the j^{th} call to $\mathcal{D}$-refresh oracle outputs (I_j, γ_j, z_j).

We consider a hybrid sequence of security games, whose procedures are described in Figure 4.7:

G_0 is the initial robustness game ROB, G_i, $\mathsf{G}_{i+1/2}$ and G_{i+1} are hybrid games, all derived from the initial robustess game ROB, for $i \in \{0, \ldots, q_n - 1\}$.

Game G_0. Game G_0 is the initial real-or-random ROB security game as in Definition 29.

Games G_i and G_{i+1}. Games G_i and G_{i+1} are modifications of game G_0 that use procedures described in Figure 4.7. Procedure $\mathsf{initialize}$ sets parameters as in G_0 and a new parameter ctr to 0. Procedures $\mathsf{finalize}$, $\mathcal{D}$-$\mathsf{refresh}$, $\mathcal{M}$-$\mathsf{set\text{-}state}$, $\mathcal{M}$-$\mathsf{get\text{-}state}$ are the same as in game G_0. Procedure $\mathsf{next\text{-}ror}$ is different from game G_0: ctr is incremented and for each uncompromised next query, if $\mathsf{ctr} \leq i$ (for G_i) *or* $\mathsf{ctr} \leq i+1$ (for G_{i+1}), the challenger generates a random couple $(S_1, R_1) \in \{0,1\}^{n+\ell}$ and returns R_1 to $\mathcal{A}$. If $\mathsf{ctr} > i$ (for G_i) or $\mathsf{ctr} > i+1$ (for G_{i+1}), the challenger behaves as in game G_0.

Game $\mathsf{G}_{i+1/2}$. Game $\mathsf{G}_{i+1/2}$ is a modification of game G_i that uses procedures described in Figure 4.7. Procedure $\mathsf{initialize}$ sets parameters as in G_i and a new Boolean parameter ns to true (this parameter intends to capture the differences between *recovering* and *preserving* uncompromised next queries). Procedures $\mathsf{finalize}$ and $\mathcal{D}$-$\mathsf{refresh}$ are the same as for the previous games. Procedures $\mathsf{get\text{-}state}$ and $\mathsf{set\text{-}state}$ are different: the flag ns is set to false in these procedures (hence meaning that the next query is recovering). Finally procedure $\mathsf{next\text{-}ror}$ is also different: ctr is incremented, and for each uncompromised next query, if $\mathsf{ctr} \leq i$ or $\mathsf{ctr} = i+1$ *and* flag $\mathsf{ns} = \mathsf{true}$, the challenger generates a random couple $(S_1, R_1) \in \{0,1\}^{n+\ell}$ and returns R_1 to $\mathcal{A}$. If $\mathsf{ctr} = i+1$ or $\mathsf{ns} = \mathsf{false}$, the challenger behaves as in game G_0.

We make use of two technical propositions, that are stated below: in Proposition 2, we prove that for all $i \in \{0, \ldots, q_n - 1\}$, $|\Pr[\mathsf{G}_i = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| \leq \varepsilon_p$, and in Proposition 3, we prove that $|\Pr[\mathsf{G}_{i+\frac{1}{2}} = 1] - \Pr[\mathsf{G}_{i+1} = 1]| \leq \varepsilon_r$.

Combining the two propositions, and using the hybrid argument, we get:

$$|\Pr[\mathsf{G}_0 = 1] - \Pr[\mathsf{G}_{q_n}) = 1]| \leq q_n(\varepsilon_r + \varepsilon_p).$$

Moreover G_{q_n} is completely independent of the challenger bit b. In particular, all $\mathsf{next\text{-}ror}$ queries return a random $R \xleftarrow{\$} \{0,1\}^\ell$ independent of the challenge bit b. Therefore, we have $\Pr[\mathsf{G}_{q_n} = 1] = \frac{1}{2}$. Combining with the above, we see that the adversary's advantage in the original robustness game is:

$$\left|\Pr[\mathsf{G}_0 = 1] - \frac{1}{2}\right| \leq q_n(\varepsilon_r + \varepsilon_p).$$

□

Let us now prove Proposition 2 and Proposition 3.

Proposition 2. *Assuming that the pseudo-random number generator with input has (t, ε_p)-preserving security, then for any adversary/distinguisher $\mathcal{A}, \mathcal{D}$ running in time $t' \approx t$, we have $|\Pr[\mathsf{G}_i = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| \leq \varepsilon_p$.*

Proof. Fix adversary/sampler pair $\mathcal{A}, \mathcal{D}$ running in time t'. Note that the two games above only differ in the special case where the $(i+1)^{th}$ next query made by $\mathcal{A}$ is *preserving*. Therefore, we can assume that $\mathcal{A}$ ensures this is always the case, as it can only maximize advantage.

We define an adversary $\mathcal{A}'$ that has advantage ε_p in the preserving security game PRES. The adversary $\mathcal{A}'$ gets a value seed from its challenger and passes it to $\mathcal{A}$. Then $\mathcal{A}'$ begins running $\mathcal{A}$ and simulating all of the oracles in the robustness security game. It chooses a random "challenge bit" $b \xleftarrow{\$} \{0,1\}$. It simulates all oracle calls made by $\mathcal{A}$ until the $(i+1)^{th}$ next query as in G_i.

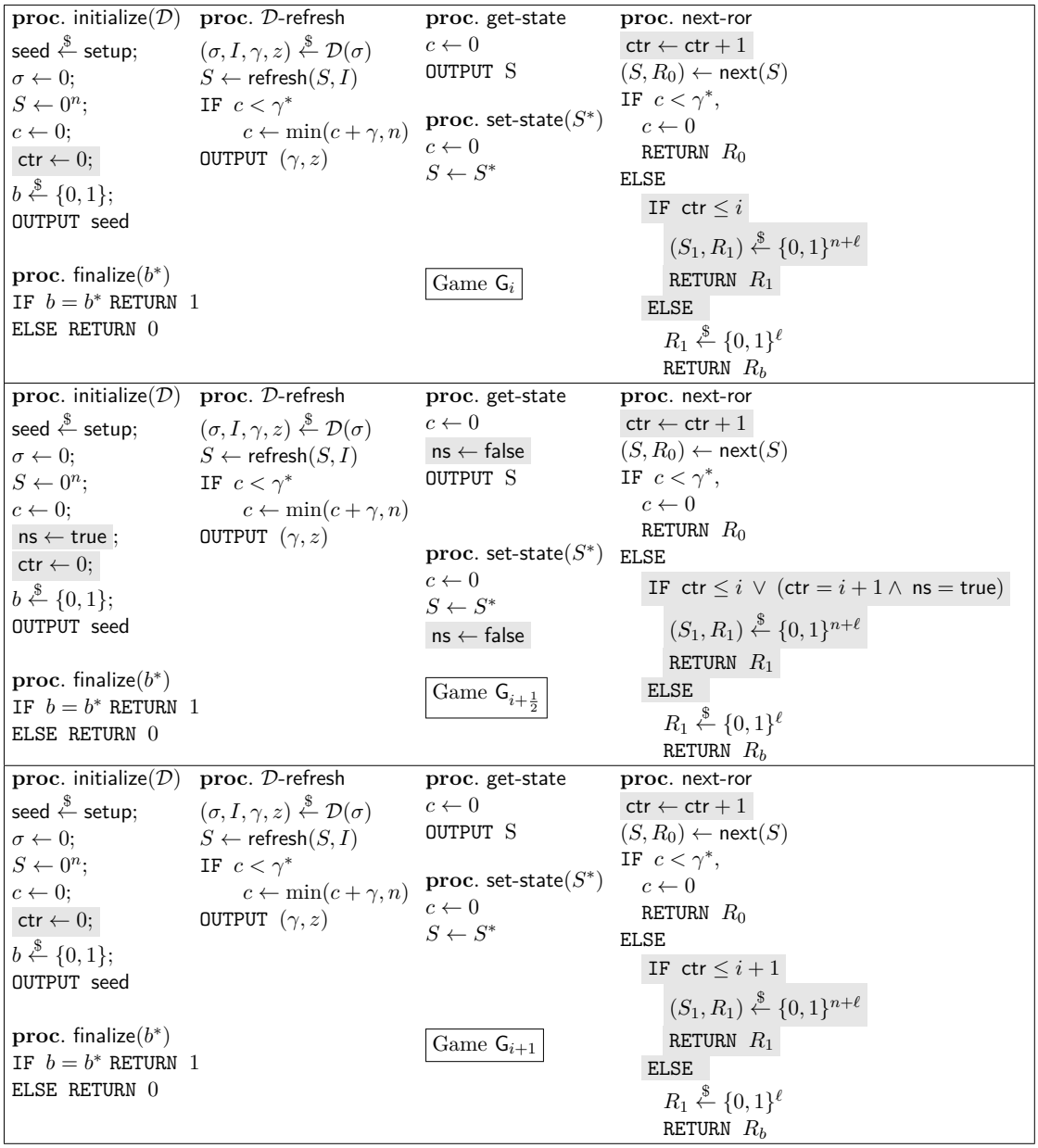

Figure 4.7 – Reductions to Preserving and Recovering Security for ROB

In particular, it simulates calls to $\mathcal{D}$-refresh using the code of the sampler $\mathcal{D}$ and updating its state. Note that $\mathcal{A}'$ has complete knowledge of the sampler state σ and the generator state S at all times.

During the $(i+1)^{th}$ next query made by $\mathcal{A}$, the adversary $\mathcal{A}'$ takes all the samples $I_1, \ldots, I_d$ which were output by $\mathcal{D}$ in between the i^{th} and $(i+1)^{th}$ next query and gives these to its challenger. It gets back a value (S^*, R_0). If the $(i+1)^{th}$ next query made by $\mathcal{A}$ is next-ror the adversary $\mathcal{A}'$ also chooses $R_1 \xleftarrow{\$} \{0,1\}^\ell$ and gives R_b to $\mathcal{A}$ where b is challenge bit randomly picked by $\mathcal{A}'$. In either case, $\mathcal{A}'$ sets the new generator state to S^* and continues running the game, simulating all future oracle calls made by $\mathcal{A}$ as in G_i. Finally, if $\mathcal{A}$ outputs the bit b^*, the adversary $\mathcal{A}'$ outputs the bit $\tilde{b}^*$ which is set to 1 if $b = b^*$.

Notice that if the challenge bit of the challenger for $\mathcal{A}'$ is $\tilde{b} = 0$ then this exactly simulates G_i for $\mathcal{A}$ and if the challenge bit is $\tilde{b} = 1$ then this exactly simulates G_{i+1}. In particular, we can think of the state immediately following the i^{th} next query as being the challenger's randomly chosen value $S_0 \xleftarrow{\$} \{0,1\}^n$, the state immediately preceding the $(i+1)^{th}$ next query being S_d which refreshes S_0 with the samples $I_1, \ldots, I_d$, and the state immediately following the query as being either $(S^*, R_0) \leftarrow \mathsf{next}(S_d)$ when $\tilde{b} = 0$ (as in G_i) or $(S^*, R_0) \xleftarrow{\$} \{0,1\}^{n+\ell}$ when $b = 1$ (as in G_{i+1}). Finally we have:

$$|\Pr[\mathsf{G}_{i+\frac{1}{2}} = 1] - \Pr[\mathsf{G}_i = 1]| = 2 \cdot (\Pr[b'^* = 1 | b' = 1] - 1) = 2 \cdot (\Pr[b'^* = b'] - 1) \leq \varepsilon_p.$$

□

Proposition 3. *Assuming that the pseudo-random number generator with input has $(t, q_r, \gamma^*, \varepsilon_r)$-recovering security, then for any adversary/distinguisher $\mathcal{A}, \mathcal{D}$ running in time $t' \approx t$, we have $|\Pr[\mathsf{G}_{i+\frac{1}{2}} = 1] - \Pr[\mathsf{G}_{i+1} = 1]| \leq \varepsilon_r$.*

Proof. Fix adversary/sampler pair $\mathcal{A}, \mathcal{D}$ running in time t'. Note that the two games above only differ in the special case where the $(i+1)^{th}$ next query made by $\mathcal{A}$ is *recovering*. Therefore, we can assume that $\mathcal{A}$ ensures this is always the case, as it can only maximize advantage.

We define an adversary $\mathcal{A}'$ such that $\mathcal{A}', \mathcal{D}$ has advantage ε_r in the recovering security game. The adversary $\mathcal{A}'$ gets a value seed from its challenger and passes it to $\mathcal{A}$. Then $\mathcal{A}'$ begins running $\mathcal{A}$ and simulating all of the oracles in the robustness security game. In particular, it chooses a random "challenge bit" $b \xleftarrow{\$} \{0,1\}$ and state $S \xleftarrow{\$} \{0,1\}^n$. It simulates all oracle calls made by $\mathcal{A}$ until right prior to the $(i+1)^{th}$ next query as in G_i. To simulate calls to $\mathcal{D}$-refresh, the adversary $\mathcal{A}'$ outputs the values γ_k, z_k that it got from its challenger in the beginning, but does not immediately update the current state S. Whenever $\mathcal{A}$ makes an oracle call to get-state, get-next, next-ror, set-state, $\mathcal{A}$' first makes sufficiently many calls to its get-refresh oracle so as to get the corresponding samples I_k that should have been sampled by these prior $\mathcal{D}$-refresh calls, and refreshes its state S accordingly before processing the current oracle call.

When $\mathcal{A}$ makes its $(i+1)^{th}$ next query, the adversary $\mathcal{A}'$ looks back and finds the *most recent entropy drain (*mRED*)* query that $\mathcal{A}$ made, and sets S_0 to the state of the generator immediately following that query. Assume $\mathcal{A}$ made d calls to $\mathcal{D}$-refresh between the mRED query and the $(i+1)^{th}$ next query (these are the "recovering samples"). Then $\mathcal{A}'$ gives (S_0, d) to its challenger and gets back (S^*, R_0) and $I_{k+d+1}, \ldots, I_{q_r}$. It chooses $R_1 \xleftarrow{\$} \{0,1\}^\ell$. If the $(i+1)^{th}$ next query made by $\mathcal{A}$ is next-ror the adversary $\mathcal{A}'$ also chooses $R_1 \xleftarrow{\$} \{0,1\}^\ell$ and gives R_b to $\mathcal{A}$, where b is challenge bit randomly picked by $\mathcal{A}'$ in the beginning. In either case, $\mathcal{A}'$ sets the new generator state to S^* and continues running the game, simulating all future oracle calls made by $\mathcal{A}$ as in G_{i+1} using the values $I_{k+d+1}, \ldots, I_{q_r}$ to simulate $\mathcal{D}$-refresh calls. Finally, if $\mathcal{A}$ outputs the bit b^*,

the adversary $\mathcal{A}'$ outputs the bit $\tilde{b}^*$ which is set to 1 if and only if $b = b^*$.

Notice that if the challenge bit of the challenger for $\mathcal{A}'$ is $\tilde{b} = 0$ then this exactly simulates $\mathsf{G}_{i+\frac{1}{2}}$ for $\mathcal{A}$ and if the challenge bit is $\tilde{b} = 1$ then this exactly simulates G_{i+1}. In particular, we can think of the state immediately following the mRED query as S_0 and the state immediately preceding the $(i+1)^{th}$ next query being S_d which refreshes S_0 with the samples $I_{k+1}, \ldots, I_{k+d}$, and the state immediately following the query as being either $(S^*, R_0) \leftarrow \mathsf{next}(S_d)$ when $\tilde{b} = 0$ (as in $\mathsf{G}_{i+\frac{1}{2}}$) or $(S^*, R_0) \xleftarrow{\$} \{0,1\}^{n+\ell}$ when $b = 1$ (as in G_{i+1}). Also, we note that $\mathcal{A}'$ is a valid adversary since the recovering samples must satisfy $\sum_{j=k+1}^{k+d} \gamma_j \geq \gamma^*$ if the $(i+1)$st next query is recovering. Finally:

$$|\Pr[\mathsf{G}_{i+1} = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| = 2 \cdot (\Pr[b'^* = 1 | b' = 1] - 1) = 2 \cdot (\Pr[b'^* = b'] - 1) \leq \varepsilon_r.$$

□

4.3 A Secure Construction

Let $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ be a standard pseudo-random generator where $m < n$. We use the notation $[y]_1^m$ to denote the first m bits of $y \in \{0,1\}^n$. Our construction of pseudo-random number generator with input has the parameters $s = 2n$ (seed length), n (state length), ℓ (output length), and $p = n$ (input length), and is defined as follows:

- $\mathsf{setup}()$: Output $\mathsf{seed} = (X, X') \xleftarrow{\$} \{0,1\}^{2n}$.
- $S' = \mathsf{refresh}(S, I)$: Given $\mathsf{seed} = (X, X')$, current state $S \in \{0,1\}^n$, and a sample $I \in \{0,1\}^n$, output: $S' := S \cdot X + I$, where all operations are over $\mathbb{F}_{2^n}$.
- $(S', R) = \mathsf{next}(S)$: Given $\mathsf{seed} = (X, X')$ and a state $S \in \{0,1\}^n$, first compute $U = [X' \cdot S]_1^m$. Then output $(S', R) = \mathbf{G}(U)$.

Notice that we are assuming each input I is in $\{0,1\}^n$. This is without loss of generality: we can take shorter inputs and pad them with 0s, or take longer inputs and break them up into n-bit chunks, calling the $\mathsf{refresh}$ procedure iteratively on each chunk.

On-line Extractor. Let's look at what happens if we start in some state S and call the $\mathsf{refresh}$ procedure d-times with the samples $I_{d-1}, \ldots, I_0$, as was done in the security games RECOV and PRES (it will be convenient to index these in reverse order). Then the new state at the end of this process will be:

$$S' := S \cdot X^d + I_{d-1} \cdot X^{d-1} + \cdots + I_1 \cdot X + I_0.$$

Let $\bar{I} := (I_{d-1}, \ldots, I_0)$ be the concatenation of all d samples. In the analysis we rely on the fact that the *polynomial evaluation* hash function defined by $h_X(\bar{I}) := \sum_{j=0}^{d-1} I_j \cdot X^j$ is $(d/2^n)$-universal meaning that the probability of any two distinct inputs colliding is at most $d/2^n$ over the random choice of X (see Section 2.3). In particular, we can think of our refresh procedure as computing this hash function in an *on-line* manner, processing the inputs I_j one-by-one without knowing the total number of future samples d, and keeping only a short local state.In particular, the updated state after the d refreshes is $S' = S \cdot X^d + h_X(\bar{I})$. Unfortunately, $h_X(\cdot)$ is not sufficiently universal to make it a good extractor, and therefore we cannot argue that S' itself is random as long as $\bar{I}$ has entropy. Therefore, we need to apply an additional hash function $h'_{X'}(Y) = [X' \cdot Y]_1^m$ which takes as input $Y \in \{0,1\}^n$ and outputs a value $h_{X'}(Y) \in \{0,1\}^m$. We show that the composition function $h^*_{X,X'}(\bar{I}) = h'_{X'}(h_X(\bar{I}))$ is a good randomness extractor. Therefore, during the evaluation of $(S'', R) = \mathsf{next}(S')$, the value

$$U = [X' \cdot S']_1^m = [X' \cdot S \cdot X^d]_1^m + h^*_{X,X'}(\bar{I})$$

is uniformly random as long as the refreshes $\bar{I}$ jointly have sufficient entropy. This is the main idea behind our construction. We formalize this via the following lemma, which provides the key to proving our main theorem.

Lemma 6. *Let d, n, m be integers, let $X, X', Y \in \mathbb{F}_{2^n}$, $\bar{I} = (I_{d-1}, \ldots, I_0) \in \mathbb{F}_{2^n}^d$. Define the hash function families:*

$$h_X(\bar{I}) := \sum_{j=0}^{d-1} I_j \cdot X^j \quad , \quad h'_{X'}(Y) := [X' \cdot Y]_1^m.$$

$$h^*_{X,X'}(\bar{I}) := h'_{X'}(h_X(\bar{I})) = \left[X' \cdot \sum_{j=0}^{d-1} I_j \cdot X^j \right]_1^m.$$

Then the hash-family $\mathcal{H} = \{h^*_{X,X'}\}$ is $2^{-m}(1 + d \cdot 2^{m-n})$-universal. In particular it is a strong (k, ε)-extractor as long as:

$$k \geq m + 2\log(1/\varepsilon) + 1, n \geq m + 2\log(1/\varepsilon) + \log(d) + 1.$$

Proof. For the first part of the lemma, fix any

$$\bar{I} = (I_{d-1}, \ldots, I_0) \neq \bar{I}' = (I'_{d-1}, \ldots, I'_0).$$

Then:

$$\begin{aligned}
\Pr_{X,X'}[h^*_{X,X'}(\bar{I}) = h_{X,X'}(\bar{I}')] &\leq \Pr_X[h_X(\bar{I}) = h_X(\bar{I}')] + \Pr_{X,X'}\left[h_{X'}(Y) = h_{X'}(Y') \;\middle|\; \begin{array}{c} Y \neq Y' \\ Y := h_X(\bar{I}), \\ Y' := h_X(\bar{I}') \end{array} \right] \\
&\leq \Pr_X\left[\sum_{j=0}^{d-1} (I_j - I'_j) \cdot X^j = 0 \right] + 2^{-m} \\
&\leq d/2^n + 2^{-m} = 2^{-m}(1 + d2^{m-n}).
\end{aligned}$$

For proving the second part, we use the fact that $h_{X,X'}$ is $2^{-m}(1+\alpha)$-universal for $\alpha = d \cdot 2^{m-n}$. Hence, it is also a (k, ε)-extractor where $\varepsilon \leq \sqrt{2^{m-k} + \alpha} = \sqrt{2^{m-k} + d2^{m-n}}$ (See Lemma 4). This is ensured by our parameter choice. □

Lemma 6 will be crucially used in establishing Theorem 10.

Theorem 10. *Let $n > m, \ell, \gamma^*$ be integers. Assume that $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a standard $(t, \varepsilon_{\mathbf{G}})$-secure pseudo-random generator. Let $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ be defined as above. Then $\mathcal{G}$ is a $((t', q_r, q_n, q_s), \gamma^*, \varepsilon)$-robust pseudo-random number generator with input where $t' \approx t$, $\varepsilon = q_n(2\varepsilon_{\mathbf{G}} + q_r^2\varepsilon_{ext} + 2^{-n+1})$ as long as $\gamma^* \geq m + 2\log(1/\varepsilon_{ext}) + 1, n \geq m + 2\log(1/\varepsilon_{ext}) + \log(q_r) + 1$.*

We present the proof below, but now make a few comments. First, it is instructive to split the security bound on ε into two parts (ignoring the "truly negligible" term $q_n \cdot 2^{-n+1}$): "computational" part $\varepsilon_{comp} = 2q_n \cdot \varepsilon_{\mathbf{G}}$ and "statistical" part $\varepsilon_{stat} = q_n q_r^2 \cdot \varepsilon_{ext}$, so that $\varepsilon \approx \varepsilon_{comp} + \varepsilon_{stat}$. Notice, the computational term ε_{comp} is already present in any "input-free" generator (or "stream cipher"), where the state S is assumed to never be compromised (so there is no $\mathsf{refresh}$ operation) and $\mathsf{next}(S) = \mathbf{G}(S)$. Also, such stream cipher has state length $n = m$. Thus, we can view the statistical term $\varepsilon_{stat} = q_n q_r^2 \cdot \varepsilon_{ext}$ and the "state overhead" $n - m = 2\log(1/\varepsilon_{ext}) + \log(q_r) + 1$ as the "price" one has to pay to additionally recover from occasional compromise (using fresh entropy gathered by the system).

In addition, to slightly reduce the number of parameters in Theorem 10, we can let k be our "security parameter" and set $q_r = q_n = q_s = 2^k$ and $\varepsilon_{ext} = 2^{-4k}$. Then we see that $\varepsilon_{stat} = 2^{3k} \cdot 2^{-4k} = 2^{-k}$, $\varepsilon_{comp} = 2^{k+1}\varepsilon_{\mathbf{G}}$ and we can set $n = m + 2\log(1/\varepsilon_{ext}) + \log(q_r) + 1 = m + 9k + 1$ and $\gamma^* = m + 2\log(1/\varepsilon_{ext}) + 1 = m + 8k + 1$. Summarizing all of these, we get Theorem 11.

Theorem 11. *Let k, m, ℓ, n be integers, where $n \geq m + 9k + 1$. Assume that $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a standard $(t, \varepsilon_{\mathbf{G}})$-secure pseudo-random generator. Then $\mathcal{G}$ is a $((t', 2^k, 2^k, 2^k), m + 8k + 1, 2^{k+1} \cdot \varepsilon_{\mathbf{G}} + 2^{-k})$-robust pseudo-random number generator with input, having n-bit state and ℓ-bit output, where $t' \approx t$.*

Coming back to our comparison with the stream ciphers (or "input-free" generators), we see that we can achieve statistical security overhead $\varepsilon_{stat} = 2^{-k}$ (with $q_r = q_n = q_s = 2^k$) at the price of state overhead $n - m = 9k + 1$ (and where entropy threshold $\gamma^* = m + 8k + 1 = n - k$).

Proof of Theorem 10 We show that $\mathcal{G}$ satisfies $(t', q_r, \gamma^*, (\varepsilon_{\mathbf{G}} + q_r^2 \varepsilon_{ext}))$-recovering security and $(t', (\varepsilon_{\mathbf{G}} + 2^{-n+1}))$-preserving security. Theorem 10 then follows directly from Theorem 9.

Proposition 4. *The pseudo-random number generator with input $\mathcal{G}$ has $(t', \varepsilon_{\mathbf{G}} + 2^{-n+1})$-preserving security.*

proc. initialize($\mathcal{D}$) $(X, X') \xleftarrow{\$}$ setup; $S_0 \xleftarrow{\$} \{0,1\}^n$; $j \leftarrow 0$; $b \xleftarrow{\$} \{0,1\}$; OUTPUT seed **proc**. finalize(b^*) IF $b = b^*$ RETURN 1 ELSE RETURN 0	**proc**. $\mathcal{D}$-refresh(I) $S_j := S_{j-1} \cdot X + I$ Game $\mathsf{G}_0 = \mathsf{PRES}$	**proc**. next-ror $U = [S_d \cdot X']_1^m$ $(S^{(0)}, R^{(0)}) \leftarrow \mathbf{G}(U)$ $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$ RETURN $(S^{(b)}, R^{(b)})$
proc. initialize($\mathcal{D}$) $(X, X') \xleftarrow{\$}$ setup; $S_0 \xleftarrow{\$} \{0,1\}^n$; $b \xleftarrow{\$} \{0,1\}$; OUTPUT seed **proc**. finalize(b^*) IF $b = b^*$ RETURN 1 ELSE RETURN 0	**proc**. $\mathcal{D}$-refresh(I) $S_j := S_{j-1} \cdot X + I$ Game G_1	**proc**. next-ror $S_d \xleftarrow{\$} \{0,1\}^n$ $U = [S_d \cdot X']_1^m$ $(S^{(0)}, R^{(0)}) \leftarrow \mathbf{G}(U)$ $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$ RETURN $(S^{(b)}, R^{(b)})$
proc. initialize($\mathcal{D}$) $(X, X') \xleftarrow{\$}$ setup; $S_0 \xleftarrow{\$} \{0,1\}^n$; $b \xleftarrow{\$} \{0,1\}$; OUTPUT seed **proc**. finalize(b^*) IF $b = b^*$ RETURN 1 ELSE RETURN 0	**proc**. $\mathcal{D}$-refresh(I) $S_j := S_{j-1} \cdot X + I$ Game G_2	**proc**. next-ror $U \xleftarrow{\$} \{0,1\}^m$ $(S^{(0)}, R^{(0)}) \leftarrow \mathbf{G}(U)$ $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$ RETURN $(S^{(b)}, R^{(b)})$
proc. initialize($\mathcal{D}$) $(X, X') \xleftarrow{\$}$ setup; $S_0 \xleftarrow{\$} \{0,1\}^n$; $b \xleftarrow{\$} \{0,1\}$; OUTPUT seed **proc**. finalize(b^*) IF $b = b^*$ RETURN 1 ELSE RETURN 0	**proc**. $\mathcal{D}$-refresh(I) $S_j := S_{j-1} \cdot X + I$ Game G_3	**proc**. next-ror $(S^{(0)}, R^{(0)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$ $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$ RETURN $(S^{(b)}, R^{(b)})$

Figure 4.8 – Preserving Security of $\mathcal{G}$

Proof. We prove the preserving security of $\mathcal{G}$ by reduction to the standard security of $\mathbf{G}$.

Let G_0 be the original security game PRES: the game outputs a bit which is set to 1 if and only if the adversary guesses the challenge bit $b^* = b$. If the initial state is $S_0 \xleftarrow{\$} \{0,1\}^n$, the seed is $\mathsf{seed} = (X, X')$, and the adversarial samples are $I_{d-1}, \ldots, I_0$ (indexed in reverse order where I_{d-1} is the earliest sample) then the refreshed state that incorporates these samples will be $S_d := S_0 \cdot X^d + \sum_{j=0}^{d-1} I_j \cdot X^j$. As long as $X \neq 0$, the value S_d is uniformly random (over the choice of S_0).
We consider the sequence of games G_0, G_1, G_2, G_3, where G_1, G_2, G_3 are all modifications of game $\mathsf{G}_0 = \mathsf{PRES}$, whose procedures are illustrated in Figure 4.8 (note that we remove the common finalize procedure in all the descriptions of the games).

In game G_1 the challenger simply picks $S_d \xleftarrow{\$} \{0,1\}^n$ uniformly at random and we have

$$|\Pr[\mathsf{G}_0 = 1] - \Pr[\mathsf{G}_1 = 1]| \leq 2^{-n}.$$

Let $U = [S_d \cdot X']_1^m$ be the value computed by the challenger during the computation $(S, R) \leftarrow \mathsf{next}(S_d)$ when the challenge bit is $b = 0$. Then, as long as $X' \neq 0$, the value U is uniformly random (over the choice S_d). Therefore, we can define G_2 where the challenger choose $U \xleftarrow{\$} \{0,1\}^n$ during this computation and we have:

$$|\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_2 = 1]| \leq 2^{-n}.$$

Finally $(S, R) = \mathsf{next}(S_d, \mathsf{seed}) = \mathbf{G}(U)$. Then (S, R) is $(t, \varepsilon_{\mathbf{G}})$ indistinguishable from uniform. Therefore we can consider a modified G_3 where the challenger just choosing (S, R) at random even when the challenge bit is $b = 0$. Since the adversary runs in time $t' \approx t$, we have:

$$|\Pr[\mathsf{G}_3 = 1] - \Pr[\mathsf{G}_2 = 1]| \leq \varepsilon_{\mathbf{G}}.$$

Since G_3 is independent of the challenge bit b, we have $\Pr[\mathsf{G}_3 = 1] = \frac{1}{2}$ and therefore

$$|\Pr[\mathsf{G}_0 = 1] - \frac{1}{2}| \leq \varepsilon_{\mathbf{G}} + 2^{-n+1}.$$

□

Proposition 5. *The pseudo-random number generator with input $\mathcal{G}$ has $(t', q_r, \gamma^*, (\varepsilon_{\mathbf{G}} + q_r^2 \varepsilon_{ext}))$-recovering security.*

Proof. We prove the recovering security of $\mathcal{G}$ (a) using that $\mathcal{H}$ is a strong randomness extractor and (b) by reduction to the standard security of $\mathbf{G}$.

Let G_0 be the original security game $\mathsf{RECOV}(q_r, \gamma^*)$: the game outputs a bit which is set to 1 if and only if the adversary guesses the challenge bit $b^* = b$.

We consider the sequence of games G_0, G_1, G_2, where G_1, G_2 are all modifications of game $\mathsf{G}_0 = \mathsf{RECOV}$, whose procedures are illustrated in Figure 4.9.

We define G_1 where, during the challenger's computation of $(S^*, R) \leftarrow \mathsf{next}(S_d)$ for the challenge bit $b = 0$, it picks $U \xleftarrow{\$} \{0,1\}^m$ uniformly at random rather than setting $U := [X' \cdot S_d]_1^m$. We argue that:

$$|\Pr[(\mathsf{G}_0) = 1] - \Pr[(\mathsf{G}_1) = 1]| \leq q_r^2 \varepsilon_{ext}.$$

The loss of q_r^2 comes from the fact that the adversary can choose the index k and the value d adaptively depending on the seed. In particular, assume that the above does not hold. Then

there must exist some values $k^*, d^* \in [q_r]$ such that the above distance is greater than ε_{ext} conditioned on the adversary making exactly k^* calls to get-refresh and choosing d^* refreshes in the game. We show that this leads to a contradiction. Fix the distribution on the subset of samples $\bar{I} = (I_{k^*+1}, \ldots, I_{k+d^*})$ output by $\mathcal{D}$ during the first step of the game, which must satisfy

$$\mathbf{H}_\infty(\bar{I} \mid \gamma_1, \ldots, \gamma_{q_r}, z_1, \ldots, z_{q_r}) \geq \gamma^*.$$

By Lemma 6, the function $h_{X,X'}(\bar{I})$ is a $(\gamma^*, \varepsilon_{ext})$-extractor, meaning that $(X, X', h_{X,X'}(\bar{I}))$ is ε_{ext}-close to (X, X', Z) where Z is random an independent of X, X'. Then, for any fixed choice of k^*, d^*, the way we compute U in G_0:

$$U := [X' \cdot S_d]_1^m = [X' \cdot S_0 X^d]_1^m + h_{X,X'}(\bar{I})$$

is ε_{exst} close to a uniformly random U as chosen in G_1. This leads to a contradiction, showing that the equation holds.

Finally, we define G_2 where, during the challenger's computation of $(S^*, R) \leftarrow \mathsf{next}(S_d)$ for the challenge bit $b = 0$, it chooses (S^*, R) uniformly at random instead of $(S^*, R) \leftarrow \mathbf{G}(U)$ as in G_1. Since the adversary runs in time $t' \approx t$, we have:

$$|\Pr[\mathsf{G}_2 = 1] - \Pr[\mathsf{G}_1 = 1]| \leq \varepsilon_{\mathbf{G}}.$$

Since G_2 is independent of the challenge bit b, we have $\Pr[\mathsf{G}_2 = 1] = \frac{1}{2}$ and therefore:

$$|\Pr[\mathsf{G}_0 = 1] - \frac{1}{2}| \leq \varepsilon_{\mathbf{G}}.$$

□

4.4 Impossibility Results

A Generic Impossibility Result. It is important to notice that there is an impossibility result when independence between the randomness source and the seed is not guaranteed. Consider any pseudo-random number generator with input $\mathcal{G}$ with an input length $p \geq 2$, consider a distribution sampler $\mathcal{D}$, where the samples I_i, $i = 1, \cdots q_r$ are such that I_0 is uniform and $[\mathsf{next}(\mathsf{refresh}(\mathsf{seed}, S_0, I))]_0 = 1$, hence $\mathbf{H}_\infty(I_0) \approx p - 1$, I_1 is uniform and $[\mathsf{next}(\mathsf{refresh}(\mathsf{seed}, S_1, I_1))]_0 = 1$, where $S_1 = \mathsf{refresh}(\mathsf{seed}, S_0, I_0)$, hence $\mathbf{H}_\infty(I_1 \mid I_0) \approx p - 1$, and generally, $[\mathsf{next}(\mathsf{refresh}(\mathsf{seed}, S_j, I_j))]_0 = 1$, where $S_j = \mathsf{refresh}(\mathsf{seed}, S_{j-1}, I_{j-1})$, and

$$\mathbf{H}_\infty(I_j \mid I_1, \ldots, I_{j-1}) \approx p - 1.$$

Let us consider an adversary $\mathcal{A}$ against the security of $\mathcal{G}$ 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. Then the first bit of the last output will always be equal to 0 and the adversary $\mathcal{A}$ breaks the robustness of the generator.

Impossibility Result for the Robust Construction. A more explicit impossibility result can also be pointed out for the secure robust construction described in Section 4.3. 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

proc. initialize($\mathcal{D}$)
$(X, X') \xleftarrow{\$}$ setup;
$\sigma_0 \leftarrow 0$;
$b \xleftarrow{\$} \{0,1\}$;
FOR $k = 1$ TO q_r DO
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
END FOR
$k \leftarrow 0$;
OUTPUT seed, $(\gamma_k, z_k)_{k=1,\ldots,q_r}$

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. getinput
$k \leftarrow k + 1$
OUTPUT I_k

proc. set-state(S^*)
$S_0 \leftarrow S^*$
$c \leftarrow 0$

proc. $\mathcal{D}$-refresh
$k \leftarrow k + 1$;
$S_j := S_{j-1} \cdot X + I_k$;
IF $c < \gamma^*$,
  $c = \min(c + \gamma_k, n)$

proc. next-ror
$U = [S_d \cdot X']_1^m$
$(S^{(0)}, R^{(0)}) \leftarrow \mathbf{G}(U)$
$(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$
RETURN $(S^{(b)}, R^{(b)})$

Game $\mathsf{G}_0 = \mathsf{RECOV}$

proc. initialize($\mathcal{D}$)
$(X, X') \xleftarrow{\$}$ setup;
$\sigma_0 \leftarrow 0$;
$b \xleftarrow{\$} \{0,1\}$;
FOR $k = 1$ TO q_r DO
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
END FOR
$k \leftarrow 0$;
OUTPUT seed, $(\gamma_k, z_k)_{k=1,\ldots,q_r}$

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. getinput
$k \leftarrow k + 1$
OUTPUT I_k

proc. set-state(S^*)
$S_0 \leftarrow S^*$
$c \leftarrow 0$

proc. $\mathcal{D}$-refresh
$k \leftarrow k + 1$;
$S_j := S_{j-1} \cdot X + I_k$;
IF $c < \gamma^*$,
  $c = \min(c + \gamma_k, n)$

proc. next-ror
$U \xleftarrow{\$} \{0,1\}^m$
$(S^{(0)}, R^{(0)}) \leftarrow \mathbf{G}(U)$
$(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$
RETURN $(S^{(b)}, R^{(b)})$

Game G_1

proc. initialize($\mathcal{D}$)
$(X, X') \xleftarrow{\$}$ setup;
$\sigma_0 \leftarrow 0$;
$b \xleftarrow{\$} \{0,1\}$;
FOR $k = 1$ TO q_r DO
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
END FOR
$k \leftarrow 0$;
OUTPUT seed, $(\gamma_k, z_k)_{k=1,\ldots,q_r}$

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. getinput
$k \leftarrow k + 1$
OUTPUT I_k

proc. set-state(S^*)
$S_0 \leftarrow S^*$
$c \leftarrow 0$

proc. $\mathcal{D}$-refresh
$k \leftarrow k + 1$;
$S_j := S_{j-1} \cdot X + I_k$;
IF $c < \gamma^*$,
  $c = \min(c + \gamma_k, n)$

proc. next-ror
$(S^{(0)}, R^{(0)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$
$(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$
RETURN $(S^{(b)}, R^{(b)})$

Game G_2

Figure 4.9 – Recovering Security of $\mathcal{G}$

$\mathcal{D}$, 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. Then after q_r calls to $\mathcal{D}$-refresh, the state of the generator is equal to:

$$S = X^{q_r-1} I_1 + X^{q_r-2} I_2 + \cdots + I_{q_r}.$$

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 $I_1, \cdots, I_{q_r}$ collectivelly contain q_r bits of entropy. Hence the adversary $\mathcal{A}$ breaks the robustness of the generator. One may argue that this kinds of attacks are made possible only because our construction does not use cryptographic primitives, however, as we now 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.

Impossibility Result for the NIST CTR_DRBG **pseudo-random number generator with input**. An explicit impossibility result can also be pointed out for the generator described

in [BK12], 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 [BK12], hence an assumption shall be made on its definition. A carefull analysis of the specification shows that a public parameter $K = \texttt{0x000102030405060708090101112131415}$ is defined in the specification, 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. The attack is similar as the attack against the 'Simplified BH, but requires the knowledge of seed.

Let us first describe the operations of CTR_DRBG. The complete description of CTR_DRBG is given in [BK12], 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 also 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 [DHY02], described in Section 3.4. As noted in the following sections, we prefer to consider the whole inputs as a sole 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 $\varnothing$ 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 initialisation value, the adversary has access to it.

The internal state of CTR_DRBG is composed with of three parts, $S = (V, K, \mathsf{ctr})$ 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).

Algorithm 1 NIST CTR_DRBG Reseed

Require: $S = (V, K, \mathsf{ctr}), I$
Ensure: $S' = (V', K', \mathsf{ctr}')$
1: $(K', V') = \mathsf{CTR\_DRBG\_update}(\mathsf{Block\_Cipher\_df}(I, 256), K, V)$
2: $\mathsf{ctr}' = 1$
3: **return** (V', K', ctr')

CTR_DRBG Reseed **Algorithm** The Reseed algorithm is described in Algorithm 1. 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.

Algorithm 2 NIST CTR_DRBG Generate

Require: $S = (V, K, \mathsf{ctr})$, n
Ensure: $S' = (V', K', \mathsf{ctr}')$, R
1: $U = \varnothing$
2: **while** $\mathsf{len}(U) < n$ **do**
3: $V' = V' + 1 \bmod 2^{128}$, $U = [U || \mathsf{AES\_ECB\_Encrypt}(\mathsf{K}', \mathsf{V}')]$
4: **end while**
5: $R = [U]^n$
6: $(K', V') = \mathsf{CTR\_DRBG\_Update}(I_a, K', V')$
7: $\mathsf{ctr}' = \mathsf{ctr} + 1$
8: **return** (V', K', ctr'), R

CTR_DRBG Generate **Algorithm** The Generate algorithm is described in Algorithm 2. It takes as input the current values for V, K, and ctr, 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 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.

Algorithm 3 NIST CTR_DRBG_Update

Require: V, K, I
Ensure: V', K'
1: $U = \varnothing$
2: **while** $\mathsf{len}(U) < (k + 128)$ **do**
3: $V' = V + 1 \bmod 2^{128}$,
4: $U = [[U || \mathsf{AES\_ECB\_Encrypt}(\mathsf{K}, \mathsf{V}')]^{\mathsf{k}+128} \oplus \mathsf{I}]$
5: $K' = [U]^k$
6: $V' = [U]_{128}$
7: **end while**
8: **return** (K', V')

CTR_DRBG_Update **Algorithm**. The two previous algorithms both rely on an internal algorithm, named CTR_DRBG_Update, described in Algorithm 3. 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.

Block_Cipher_df **Function**. The derivation function Block_Cipher_df is used in the previous algorithms. It is described in Algorithm 4. This function uses the public parameter $K =$ 0x000102030405060708090101121314 15 as a key to encrypt the input of the generator.

BCC **Function**. The BCC function is used in the previous algorithms. It is described in Algorithm 5. This function operates a bloc cipher AES_ECB_Encrypt, which corresponds to the AES in ECB mode, and chains the successive outputs.

Let us now describe the attack against the robustness of CTR_DRBG. 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: $(i+1; [I_0^i, \cdots, I_{31}^i]) \leftarrow \mathcal{D}(i)$; where $I_0, \cdots, I_{15}$ are random and $I_{16}, \cdots, I_{31} = \mathsf{AES\_ECB\_Decrypt}(K, I_0, \cdots, I_{15})$, where $K =$ 0x00 $\cdots$ 15 (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_i = 128$). 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.

Algorithm 4 NIST CTR_DRBG Block_Cipher_df

Require: I, n
Ensure: R
1: $L = \mathsf{len}(I)/8$, $N = \mathsf{len}(n)/8$, $S = [L||N||I||\mathtt{0x80}]$
2: **while** $\mathsf{len}(S) \bmod 128 \neq 0$ **do**
3: $S = S||\mathtt{0x00}$
4: **end while**
5: $U = \varnothing$, $i = 0$, $K = \mathtt{0x000102030405060708090101112131415}$
6: **while** $\mathsf{len}(U) < 256$ **do**
7: $IV = i||0$, $U = [U||\mathsf{BCC}(K, IV||S)]$, $i = i + 1$
8: **end while**
9: $K = [U]^{128}$, $X = [U]_{128}$, $V = \varnothing$
10: **while** $\mathsf{len}(V) < (k + 128)$ **do**
11: $X = \mathsf{AES\_ECB\_Encrypt}(K, X)$
12: $V = [V||X]$
13: **end while**
14: **return** $R = [V]^{128}$

Algorithm 5 NIST CTR_DRBG BCC

Require: K, $I, |I| \bmod 128 = 0$
Ensure: $R, |R| = 128$
1: $U = 0$
2: $n = |I|/128$
3: `parse` I `as` $[B_n, \cdots, B_1]$
4: **for** $i = 1$ to n **do**
5: $I = B_i \oplus U$
6: $U = \mathsf{AES\_ECB\_Encrypt}(K, I)$
7: **end for**
8: **return** $R = U$

- After $\mathcal{D}$-refresh, the Reseed algorithm is first applied: the new state is the output of $\mathsf{CTR\_DRBG\_update}(\mathsf{Block\_Cipher\_df}(I^0, 256), K, V)$ and $\mathsf{ctr} = 1$. Let us describe the algorithm $\mathsf{Block\_Cipher\_df}(I^0, 256)$: on input I^0 and 256, Block_Cipher_df calculates $L = 32$, $N = 32$, $S = [32||32||I^0||\mathtt{0x80}]$ and then $S = [32||32||I^0||\mathtt{0x80}||\mathtt{0x00}||\cdots||\mathtt{0x00}]$. Following, it calculates $\mathsf{BCC}(K, IV||S)$, for $IV = 0||0$ and $IV = 1||0$, with $K = \mathtt{0x00}\cdots\mathtt{15}$, then sets $U = \mathsf{BCC}(K, 1||0||S)||\mathsf{BCC}(K, 0||0||S)$, $K = [U]^{128}$, $X = [U]_{128}$. Let us describe the algorithm $\mathsf{BCC}(K, IV||S)$: on input $S = [32||32||I^0||\mathtt{0x80}||\mathtt{0x00}||\cdots||\mathtt{0x00}]$, $IV = 0||0$ and $K = \mathtt{0x00}\cdots\mathtt{15}$, it parses S as B_4, B_3, B_2, B_1 and calculates $I = B_1$, $U = \mathsf{AES\_ECB\_Encrypt}(K, I)$, $I = B_2 \oplus U$, $U = \mathsf{AES\_ECB\_Encrypt}(K, I)$, $I = B_3 \oplus U$, $U = \mathsf{AES\_ECB\_Encrypt}(K, I)$, $I = B_4 \oplus U$, $U = \mathsf{AES\_ECB\_Encrypt}(K, I)$. However, the input distribution os such that $B_3 = \mathsf{AES\_ECB\_Decrypt}(K, B_2)$ and therefore the output of algorithm BCC is known to $\mathcal{A}$. Hence the output of algorithm Block_Cipher_df is also known to $\mathcal{A}$ and also the output of the Reseed algorithm, although the initial input was of high entropy.
- After next-ror, the output of the generator is computed from a known state and is therefore predictable.

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

4.5 Instantiation

We now instantiate our main construction presented in Theorem 11 for various values of "security parameter" k using AES_128 in counter mode for the standard pseudo-random generator $\mathbf{G}$.

Namely, we set $m = \ell = 128$ (recall, m is the standard pseudo-random number generator input size, and ℓ in the output size), and let $\mathbf{G}(U) = \mathrm{AES}_U(0) \ldots \mathrm{AES}_U(i-1)$, where $i = \lceil (n+128)/128 \rceil$ is the number of calls to AES_128 to get one 128-bit output. Recall also from Theorem 11 that we set the state length $n = m + 9k + 1 = 9k + 129$, which gives $i = 2 + \lceil (9k+1)/128 \rceil$.

We need to set the security ε_{prg} of our counter-mode standard pseudo-random number generator in terms of the security of AES. This turns out to be a slightly subtle issue, which we discuss at the end of this section, in part because it is based on assumptions, and also because the "provable term" $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg}$ seems to be overly pessimistic and does not correspond to an actual attack. Hence, for now we will optimistically assume that, for the values of security parameter k we consider, we have $\varepsilon_{comp} \leq \varepsilon_{stat} = 2^{-k}$, so that $\varepsilon = \varepsilon_{comp} + \varepsilon_{stat} \approx \varepsilon_{stat} = 2^{-k}$.

We consider setting the security level k to three values: 40, 50 and 64. Then as $n = m+9k+1 = 9k+129$, $\gamma^* = m + 8k + 1 = 8k + 129$, and $i = 2 + \lceil (9k+1)/128 \rceil$, we get:

- For $k = 40$, we get $n = 489$, $\gamma^* = 449$, $i = 5$.
- For $k = 50$, we get $n = 579$, $\gamma^* = 529$, $i = 6$.
- For $k = 64$, we get $n = 705$, $\gamma^* = 641$, $i = 7$.

We can instantiate $\mathcal{G}$ with AES in counter mode and the fields $\mathbb{F}_{2^{489}}$ (defined by the polynom $X^{489} + X^{83} + 1$), $\mathbb{F}_{2^{579}}$ (defined by the polynom $X^{579} + X^{12} + X^9 + X^7 + 1$) and $\mathbb{F}_{2^{705}}$ (defined by the polynom $X^{705} + X^{17} + 1$). We set the output size of AES function equal to 128 bits and we describe this instantiation with $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$, where:

- $\mathsf{setup} = (X, X') \xleftarrow{\$} \{0,1\}^{489+489}$ (*resp.* $\{0,1\}^{579\times 579}$, $\{0,1\}^{705\times 705}$);
- $\mathsf{refresh}(S, I) = S \cdot X + I \in \mathbb{F}_{2^{489}}$ (*resp.* $\mathbb{F}_{2^{579}}$, $\mathbb{F}_{2^{705}}$);
- $\mathsf{next}(S) : U = [S \cdot X']_1^{128}$, $(S', R) = (\mathrm{AES}_U(0), \ldots, \mathrm{AES}_U(4))$ (*resp.* $\mathrm{AES}_U(5)$), $\mathrm{AES}_U(6)$).

Computational Term ε_{comp}. We now come back to estimating the computational term $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg}$, and our optimistic assumption that $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg} \leq \varepsilon_{stat} = 2^{-k}$, which is equivalent to $\varepsilon_{prg} \leq 2^{-2k-1}$. Since we also want the running time $t \geq q_R = 2^k$, we essentially need our pseudo-random generator $\mathbf{G}$ to be $(2^k, 2^{-2k})$-secure. However, it is easy to notice that any $(2^k, \varepsilon_{prg})$-secure standard pseudo-random number generator with an m-bit key cannot have security $\varepsilon_{prg} < 2^{k-m}$, since the attacker in time 2^k can exhaustively try 2^k out of 2^m key to achieve advantage 2^{k-m}. This means that we need to have $2^{-2k} \geq 2^{k-m}$, or $k \leq m/3$. For example, when using AES_128 in counter mode, this seems to suggest we can have $\varepsilon_{comp} \leq \varepsilon_{stat}$ only for $k \leq 42 = \lfloor 128/3 \rfloor$, which is not the case for our high and unbreakable security settings.

However, we believe that the above analysis is overly pessimistic. Indeed, in theory, if we want to use a standard pseudo-random number generator in a stream cipher mode ($(S, R) \leftarrow \mathbf{G}(S)$) for 2^k times, we can only claim "union bound" security $2^k\varepsilon_{prg}$, which, as we saw, is only possible when $k \leq m/3$. Although tight in theory, the bound does not seem to correspond to any *concrete attack* when used with most "real-world" standard pseudo-random number generators (such as AES_128 in counter mode). For example, for $k = 64$, the bound $2^k\varepsilon_{prg} \geq 2^{64} \cdot 2^{-64} = 1$, which suggests (if the bound was tight!) that one can break a stream cipher built from AES_128 in the counter mode in 2^{64} queries with advantage 1. However, we are presently not aware of any attack achieving advantage even 2^{-64}, let alone 1. To put it differently, we think that our original assumption that $\varepsilon_{comp} \leq \varepsilon_{stat}$ for $k = 64$ seems reasonable based on our current knowledge,

even though theoretical analysis suggests that there is little point to set $k > 42$.

Based on this discussion, we suggest the following recipe when instantiating our construction with a particular standard pseudo-random number generator $\mathbf{G}$. Instead of directly looking at the term $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg}$ when examining a candidate value of security parameter k, one should ask the following question instead: *based on the current knowledge, what is the largest value of k (call it k^*) so that no attacker can achieve advantage better than 2^{-k} when $\mathbf{G}$ is used in the stream cipher mode for 2^k times?* When this k^* is determined, there is no point to set $k > k^*$, as this only increases the state length n and degrades the efficiency of the generator, without increasing its security ε beyond 2^{-k^*} (as $\varepsilon_{comp} \leq 2^{-k^*}$ anyway). However, setting $k \leq k^*$ will result in final security $\varepsilon \approx 2^{-k}$ while improving the efficiency of the resulting generator (i.e., state length $n = m + 9k + 1$, $\gamma^* = m + 8k + 1$, and the complexity of refresh and next).

With this (somewhat heuristic) recipe, we believe setting $k^* = 64$ was a fair and reasonable choice when using AES_128 in counter mode to implement $\mathbf{G}$.

4.6 Benchmarks

We now present efficiency benchmarks between our construction $\mathcal{G}$ and LINUX, a pseudo-random number generator with input that we analyze in Section 7.2. These benchmarks are based on a very optimistic hypothesis concerning LINUX and even with this hypothesis, our construction $\mathcal{G}$ appears to be more efficient. As shown, a complete internal state accumulation is on average two times faster for $\mathcal{G}$ than for LINUX and a 2048-bits key generation is on average ten times faster for $\mathcal{G}$ than for LINUX.

For LINUX, we made the (optimitic) hypothesis that for the given input distribution, the mixing function of LINUX accumulates the entropy in the internal state, that is $\mathbf{H}_\infty(M(S, I)) = \mathbf{H}_\infty(S) + \mathbf{H}_\infty(I)$ if S and I are independent, and that the SHA1 function used for transfer between the pools and output is a perfect extractor, that is $\mathbf{H}_\infty(\mathsf{SHA1}(S_*)) = 160$ if $\mathbf{H}_\infty(S_*) = 160$. Of course, both of these hypotheses are extremely strong, but we make them to achieve the most optimistic (and probably unrealistic!) estimates when comparing LINUX with our construction $\mathcal{G}$.

We implemented LINUX with functions extract_buf and mix_pool_bytes that we extracted from the source code and we implemented $\mathcal{G}$ using fb_mul_lodah and fb_add from RELIC open source library [AG] (that we extended with the fields $\mathbb{F}_{2^{489}}$, $\mathbb{F}_{2^{579}}$ and $\mathbb{F}_{2^{705}}$), aes_setkey_enc and aes_crypt_ctr from PolarSSL open source library [Pol]. CPU cycle count was done using ASM instruction RDTSC. Implementation was done on a x86 Ubuntu workstation. All code was written in C, we used gcc C compiler and linker, code optimization flag O2 was used to build the code.

4.6.1 Benchmarks on the Accumulation Process

First benchmarks are done on the accumulation process. We simulated a complete accumulation of the internal state for LINUX and $\mathcal{G}$ with an input containing one bit of entropy per byte. For $\mathcal{G}$, by Theorem 10, 8 inputs of size 449 bits (*resp.* 579, 705 bits) are necessary to recover from an internal state compromise, whereas by hypothesis, for LINUX, $\lfloor 160/12 \rfloor = 13$ inputs of size 12 bits are necessary to recover from an internal state compromise and transfers need to be done between the input pool and the output pools.

For LINUX, denoting $S^t = (S_i^t, S_u^t, S_r^t)$, where S_i^t, S_u^t and S_r^t are the successive states of the input pool, the non-blocking output pool and the blocking output pool, respectively, we implemented

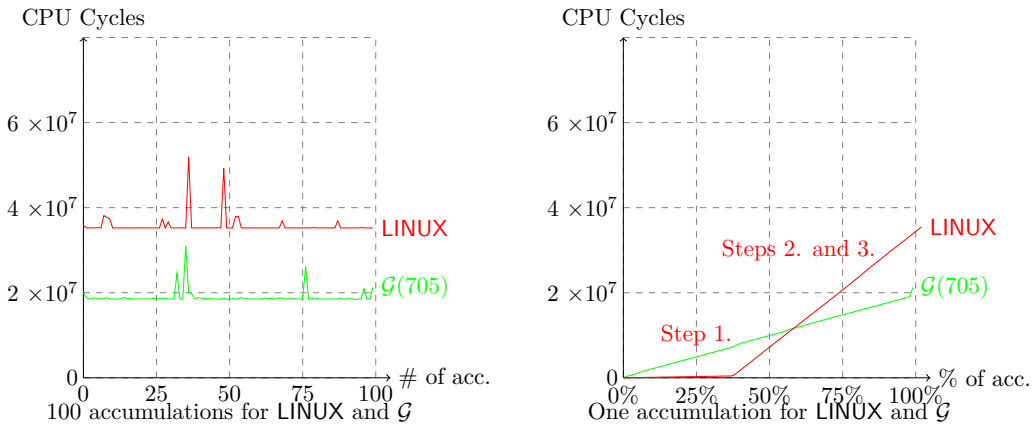

Figure 4.10 – Benchmark on the Accumulation Process

the following process, starting from a compromised internal state (S_i^0, S_u^0, S_r^0), of size 6144 bits, and using successive inputs of size 12 bytes, that we denote I^t:

1. Refresh S_i^0 with $I^0, \cdots, I^{13}$: $S_i^t = \mathsf{M}(S_i^{t-1}, I^{t-1})$. By hypothesis, $\mathbf{H}_\infty(S_i^{13}) = 168$.

2. Transfer 1024 bits from S_i^{13} to S_r. The transfer is made by blocks of 80 bits, therefore, 13 transfers are necessary. Each transfer is done in two steps: first LINUX generates from S_i^{13} an intermediate data $T_i^{13} = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_i^{13}, \mathsf{H}(S_i^{13}))$ and then it mixes it with S_r, giving the new states $S_i^{14} = \mathsf{M}(S_i^{13}, \mathsf{H}(S_i^{13}))$ and $S_r^{14} = \mathsf{M}(S_r^{13}, T_i^{13})$. Then by hypothesis, $\mathbf{H}_\infty(S_r^{13}) = 80$. After repeating these steps 12 times, by hypothesis, $\mathbf{H}_\infty(S_r^{26}) = 1024$.

3. Repeat step 2. for S_u instead of S_r. By hypothesis, $\mathbf{H}_\infty(S_u^{39}) = 1024$.

After this process, by hypothesis, $\mathbf{H}_\infty(S^{39}) = 6144$ is maximal.

For $\mathcal{G}$, denoting S^t the successive states of the internal state, we implemented the following process, starting from a compromised internal state S^0, of size 489 bits (*resp.* 579, 705 bits), and using successive inputs I^t, of size 489 bits (*resp.* 579, 705 bits): Refresh S^0 with $I^0, \cdots, I^7$: $S^i = S^{i-1} \cdot X + I^{i-1}$. After this process, by Theorem 10, $\mathbf{H}_\infty(S^8) = 489$ (*resp.* 579, 705 bits) is maximal.

The number of CPU cycles to perform these processes on LINUX and $\mathcal{G}$ (with internal state size 705 bits) are presented in Figure 4.10. We first implemented 100 complete accumulations processes for LINUX and $\mathcal{G}$ and we compared one by one each accumulation. As shown on the left part of Figure 4.10, a complete accumulation in the internal state of $\mathcal{G}$ needs on average two times less CPU cycles than a complete accumulation the internal state of LINUX. Then we analyze one accumulation in detail or LINUX and $\mathcal{G}$. As shown on the right part of Figure 4.10, a complete accumulation in the internal state of LINUX needs more CPU because of the transfers between the input pool and the two output pools done in steps 2. and 3, it also shows that the refresh function of $\mathcal{G}$ is similar as the Mixing function M of LINUX.

4.6.2 Benchmarks on the Generation Process

Second benchmarks are done on the generation process. We simulated the generation of 2048-bits keys K for LINUX and $\mathcal{G}$. For $\mathcal{G}$, 16 calls to next are necessary, as each call outputs 128 bits. For LINUX, each call to next outputs 80 bits, therefore 12 calls are first necessary, then 1024 bits need to be transfered from the input pool to the output pool, then 12 new calls to next are

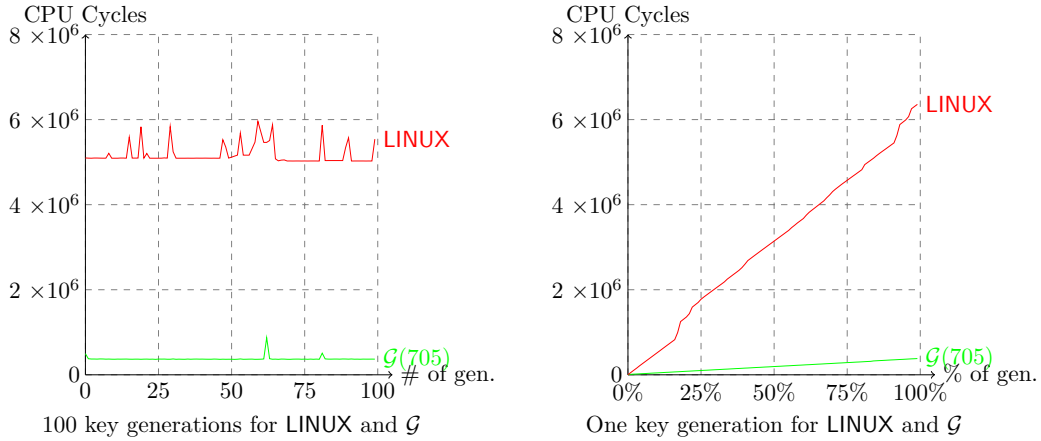

Figure 4.11 – Benchmarks on the Generation Process

necessary.

For LINUX, denoting R^t the successive ouputs, we implemented the following process, starting from an internal state (S_i^0, S_r^0, S_u^0), where we suppose at least 1024 bits of entropy are accumulated in the output pool S_r^0 and 4096 bits of entropy are accumulated in the input pool S_i^0:

1. Set $R^0 = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_r^0, \mathsf{H}(S_r^0))$
2. Repeat step 1. 12 times and set $K^0 = [R^0 || \dots || R^{12}]_1^{1024}$.
3. Transfer 1024 bits from S_i^0 to S_r^0. The transfer is made by blocks of 80 bits, therefore, 13 transfers are necessary. Each transfer is done in two steps: first LINUX generates from S_i^0 an intermediate data $T_i^0 = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_i^0, \mathsf{H}(S_i^0))$ and then it mixes it with S_r^0, giving the new states $S_i^1 = \mathsf{M}(S_i^1, \mathsf{H}(S_i^1))$ and $S_r^1 = \mathsf{M}(S_r^0, T_i^0)$. Then by hypothesis, $\mathbf{H}_\infty(S_r^1) = 80$. After repeating these steps 12 times, by hypothesis, $\mathbf{H}_\infty(S_r^{13}) = 1024$.
4. Set $R^{13} = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_r^{13}, \mathsf{H}(S_r^{13}))$
5. Repeat step 1. 12 times and set $K^1 = [R^{13} || \dots || R^{25}]_1^{1024}$.
6. Set $K = [K^0 || K^1]$

After this process, $\mathbf{H}_\infty(K) = 2048$.

For $\mathcal{G}$, we implemented the following process (using the Practical Efficiency Optimization presented in Section 4.3, starting from an internal state S^0, of size 489 bits (*resp.* 579, 705 bits), where we suppose at least $\gamma^* = 449$ (*resp.* 529, 641 bits) bits of entropy are accumulated:

1. Set $U = [S \cdot X']_1^{128}$ and $(S^1, R^0) = (\mathrm{AES}_U(0), \dots, \mathrm{AES}_U(4))$ (*resp.* $\mathrm{AES}_U(5)$), $\mathrm{AES}_U(6))$ and set the Boolean flag $\mathsf{last} = \mathsf{true}$.
2. Set $(U, R) = (\mathrm{AES}_U(0), \mathrm{AES}_U(1))$ and set $[S]_1^{128} = U$.
3. Repeat step 2. 14 times.

After this process, $\mathbf{H}_\infty(K) = 2048$.

The number of cycles to perform these processes on LINUX and $\mathcal{G}$ (with internal state size 705 bits) are presented in Figure 4.11. We first implemented the generation of 100 2048-bits keys

and we compared one by one each generation. As shown on the left part of Figure 4.11, 2048-bits key generation with $\mathcal{G}$ needs on average ten times less CPU cycles than with LINUX. Then we analyze one accumulation in detail or LINUX and $\mathcal{G}$. As shown on the right part of Figure 4.11, a 2048-bits key generation needs more CPU for LINUX.

Chapter 5

Robustness Against Memory Attacks

5.1 Model Description

In this chapter we give a syntactic formalization for security of pseudo-random number generators with input against memory attacks. All statements are part of [CR14]. We use Definition 27 for pseudo-random number generator with input in all this chapter.

We propose a modification of the *robustness* security model of Chapter 4 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, we consider the internal state as a concatenation of several binary strings (named hereafter its *decomposition*). We model 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* $\mathcal{M}$. We assume that the adversary $\mathcal{A}$ knows the *decomposition* of S and is able to choose $\mathcal{M}$ adaptively. The only differences between our security game and the original game ROB is that we replace 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.

proc. initialize($\mathcal{D}$)
seed $\xleftarrow{\$}$ setup;
$\sigma \leftarrow 0$;
$S \xleftarrow{\$} \{0,1\}^n$;
$c \leftarrow n$;
corrupt $\leftarrow$ true;
$b \xleftarrow{\$} \{0,1\}$;
OUTPUT seed

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. $\mathcal{D}$-refresh
$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$
$S \leftarrow$ refresh(S, I)
IF $c < \gamma^*$
  $c \leftarrow \min(c + \gamma, n)$
OUTPUT (γ, z)

proc. $\mathcal{M}$-set-state(S, M, J)
$S \leftarrow \mathcal{M}$-set(S, M, J)
$c \leftarrow \max(0, c - \lambda)$
IF $c < \gamma^*$,
  $c \leftarrow 0$

proc. $\mathcal{M}$-get-state(S, J)
$c \leftarrow \max(0, c - \lambda)$
IF $c < \gamma^*$,
  $c \leftarrow 0$
OUTPUT $\mathcal{M}$-get(S, J)

proc. next-ror
$(S, R_0) \leftarrow$ next(S)
IF $c < \gamma^*$,
  $c \leftarrow 0$
  OUTPUT R_0
ELSE
  $R_1 \xleftarrow{\$} \{0,1\}^\ell$
  OUTPUT R_b

Figure 5.1 – Procedures in Security Game MROB(γ^*, λ)

Definition 32 (Decomposition)**.** *A decomposition of a binary string $S \in \{0,1\}^n$ is a sequence of disjoint binary strings $(S_1, \cdots, S_k)$, such that $S = [S_1 || \cdots || S_k]$. Two binary strings S and M have the* same *decomposition if $M = [M_1 || \cdots || M_k]$ and $|S_i| = |M_i|$ for $i \in \{1, \cdots, k\}$.*

Definition 33 ($\mathcal{M}$-get / $\mathcal{M}$-set)**.** *Function $\mathcal{M}$-get takes as input a couple (S, J), where $S = [S_1 || \cdots || S_k]$ and $J \subset \{1, \cdots, k\}$, 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 $S = [S_1 || \cdots || S_k], M = [M_1 || \cdots || M_k]$ and $J \subset \{1, \cdots, k\}$, 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.

Security Model. We now describe our security model. It is adapted from the security game $\mathsf{ROB}(\gamma^*)$ that defines the *robustness* of a pseudo-random number generator with input. We describe briefly the parameters of the security game:

- Integer γ^*: Defines the minimum entropy that is required in S for the generator to be secure.
- Integer c: Defines the estimate of the amount of collected entropy.
- Integer $\lambda \leq n$: Defines the size of the mask $\mathcal{M}$.
- Boolean flag corrupt: Is et to true if $c < \gamma^*$ and false otherwise.
- Boolean b: Is used to challenge the adversary $\mathcal{A}$.

Our security game uses procedures described in Figure 5.1. 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 we initially set c to n and S to a random value, to avoid give 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. The other procedures are defined below:

- Procedure $\mathcal{D}$-refresh: $\mathcal{A}$ calls the distribution sampler $\mathcal{D}$ for a new input and uses this input to refresh $\mathcal{G}$. The estimated entropy given by $\mathcal{D}$ is used by the procedure to update the counter c ($c \leftarrow c + \gamma$) and if $c \geq \gamma^*$, then the flag corrupt is set to false.
- 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 λ, 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 λ, 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: Challenges $\mathcal{A}$ on its capability to distinguish the output of $\mathcal{G}$ from random, where the real output (R_0) of $\mathcal{G}$ is obtained with a call to algorithm next and the random string (R_1) is ampled uniformly at random by the challenger. Attacker $\mathcal{A}$ responds to the challenge with a bit b^*

The security definitions of a pseudo-random number generator with input against memory attacks is given in Definition 34.

Definition 34 (Security of a Pseudo-Random Number Generator with Input against Memory Attacks [CR14])**.** *A pseudo-random number generator with input* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *is called* $(T = (t, q_r, q_n, q_s), \gamma^*, \varepsilon)$-robust (resp. resilient, forward-secure, backward-secure)*, against memory attacks, 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 *and* q_s *calls to* $\mathcal{M}$-get-state *or* $\mathcal{M}$-set-state*, and any* legitimate *distribution sampler* $\mathcal{D}$ *inside the* $\mathcal{D}$-refresh *procedure, the advantage of* $\mathcal{A}$ *in game* $\mathsf{MROB}(\gamma^*, \lambda)$ *(resp,* $\mathsf{MRES}(\gamma^*)$*,* $\mathsf{MFWD}(\gamma^*, \lambda)$*,* $\mathsf{MBWD}(\gamma^*, \lambda)$*) is at most* ε*, where:*

- $\mathsf{MROB}(\gamma^*, \lambda)$ *is the unrestricted game where* $\mathcal{A}$ *is allowed to make the above calls and corrupt at most* λ *bits of* S*.*
- $\mathsf{MRES}(\gamma^*)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* $\mathcal{M}$-get-state/$\mathcal{M}$-set-state *(i.e.,* $q_s = 0$ *and* $\lambda = 0$*).*
- $\mathsf{MFWD}(\gamma^*, \lambda)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* $\mathcal{M}$-set-state *and a single call to* $\mathcal{M}$-get-state *(i.e.,* $q_s = 1$*) which is the very* last *oracle call* $\mathcal{A}$ *is allowed to make to corrupt* λ *bits of* S*.*
- $\mathsf{MBWD}(\gamma^*, \lambda)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* $\mathcal{M}$-get-state *and a single call to* $\mathcal{M}$-set-state *(i.e.,* $q_s = 1$*) which is the very* first *oracle call* $\mathcal{A}$ *is allowed to make to corrupt* λ *bits of* S*.*

Hence, resilience protects the security of the generator when it is not corrupted against arbitrary distribution samplers $\mathcal{D}$; forward security protects past generator outputs in case of a memory attack; backward security security ensures that the generator can successfully recover from a memory attack, provided enough fresh entropy is injected into the system; robustness ensures security against arbitrary combinations of the above.

Examples of the entropy traces for the procedures defined in our new model are provided in Figure 5.2 (which shall be compared with the traces presented in Figure 4.3). Here, calls to $\mathcal{M}$-set-state and $\mathcal{M}$-get-state only decrease the counter to λ, unless $c < \gamma^*$ (in this case c is reset to 0). Also note that as in Figure 4.3, we illustrated two next-ror calls, the first one where $c \geq \gamma^*$ and the second one where $c < \gamma^*$.

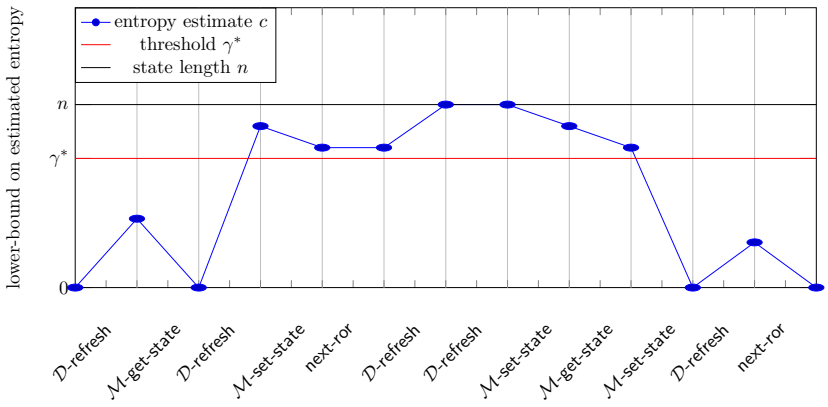

Figure 5.2 – Entropy Estimates in $\mathsf{MROB}(\gamma^*, \lambda)$

5.2 Limitation of the Initial Security Property

We show that it is possible to construct a robust pseudo-random number generator with input (Definition 29) that never resists a single bit corruption.

Consider $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ a robust pseudo-random number generator with input of internal state S, and $\mathcal{G}' = (\mathsf{setup}', \mathsf{refresh}', \mathsf{next}')$ a second pseudo-random number generator with input of internal state $S' = S||\mathsf{b}$ where b is a single bit, defined with the following algorithms:

- $\mathsf{refresh}'(S', I) = \mathsf{refresh}(S, I)||1$ *(i.e.* $S \leftarrow \mathsf{refresh}(S, I)$ and $\mathsf{b} \leftarrow 1$)

- $\mathsf{next}'(S') = \mathsf{next}(S)$ if $\mathsf{b} = 1$, $\mathsf{next}'(S') = 0$ if $\mathsf{b} = 0$

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.

To achieve our new security property, we need to define a new property named "preserving security under partial state corruption". Intuitively, it states that if the state gets partially compromised between two next calls, such that the estimated entropy inside the internal state remains above the threshold γ^*, then the generator should remain safe. Below we describe the notions of *preserving security against memory attacks* and *recovering security against memory attacks*, both adapted from the *preserving security* and *recovering security* defined in Chapter 4.

5.3 Preserving and Recovering Security Against Memory Attacks

Preserving Security Against Memory Attacks. We now describe our first security property. It states that if S_0 starts uncompromised and gets updated with calls to algorithms refresh, interleaved with calls to $\mathcal{M}$-set-state or $\mathcal{M}$-get-state, such that the state remains uncompromised, the output of next should be undistinguishable from random. The security game MPRES uses

proc. initialize($\mathcal{D}$)
seed $\xleftarrow{\$}$ setup;
$S \xleftarrow{\$} \{0,1\}^n$;
$c \leftarrow n$;
$b \xleftarrow{\$} \{0,1\}$;
$\sigma_k \leftarrow 0$;
FOR $k = 1$ TO q_r DO
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
END FOR
$k \leftarrow 0$;
OUTPUT seed, $(\gamma_k, z_k)_{k=1,\ldots,q_r}$

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. $\mathcal{D}$-refresh
$k \leftarrow k + 1$;
$S = \mathsf{refresh}(S, I_k)$;
IF $c < \gamma^*$,
  $c = \min(c + \gamma_k, n)$

proc. $\mathcal{M}$-set-state(S, M, J)
$S \leftarrow \mathcal{M}\text{-set}(S, M, J)$
$c \leftarrow \max(0, c - \lambda)$
IF $c < \gamma^*$,
  $c \leftarrow 0$

proc. $\mathcal{M}$-get-state(S, J)
$c \leftarrow \max(0, c - \lambda)$
IF $c < \gamma^*$,
  $c \leftarrow 0$
OUTPUT $\mathcal{M}\text{-get}(S, J)$

proc. next-ror
$(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$
$(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$
OUTPUT $(S^{(b)}, R^{(b)})$

Figure 5.3 – Procedures in Security Game MPRES(q_r, γ^*, λ)

procedures described in Figure 5.3. The security game MPRES is described as follows, with an adversary $\mathcal{A}$ and bounds q_r, γ^*, λ:

1. The challenger generates a seed seed $\xleftarrow{\$}$ setup and a bit $b \xleftarrow{\$} \{0,1\}$ uniformly at random. It sets $\sigma_0 = 0$ and for $k = 1, \ldots, q_r$, it computes $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$, initializes $k = 0$, computes a state S at random and sets $c = n$. It then gives back the seed and the values $\gamma_1, \ldots, \gamma_{q_r}$ and $z_1, \ldots, z_{q_r}$ to the adversary.

2. The adversary $\mathcal{A}$ gets seed and can ask as many queries as it wants to the oracle $\mathcal{D}$-refresh, the challenger updates the state $S_j := \mathsf{refresh}(S_{j-1}, I_{k+j})$ and updates $c \leftarrow c + \gamma_k$ sequentially. .

3. The challenger allows queries to the oracles $\mathcal{M}$-set-state, and $\mathcal{M}$-get-state. These queries are processed, respectively, as $\{S \leftarrow \mathcal{M}\text{-set}(S, M, J); c \leftarrow c - \lambda;$ if $c < \gamma^*,$ then $c = 0\}$

and $\{c \leftarrow c - \lambda;$ if $c < \gamma^*,$ then $c = 0\}$, with the inputs (M, J) and J provided by the adversary. These queries are answered, respectively, by noting and by $\mathcal{M}$-get(S, J);

4. Eventually, under the restriction that c never dropped to 0, the challenger sets $(S^{(0)}, R^{(0)}) \leftarrow$ next(S) and generates $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$. It then gives $(S^{(b)}, R^{(b)})$ to the adversary;

5. The adversary $\mathcal{A}$ outputs a bit b^*.

The Preserving Security Against Memory Attacks is given in Definition 35.

Definition 35 (Preserving Security Against Memory Attacks)**.** *A pseudo-random number generator with input* $\mathcal{G} =$ (setup, refresh, next) *is said* $(t, q_r, q_s, \gamma^*, \lambda, \varepsilon)$*-preserving against memory attacks if for any adversary* $\mathcal{A}$ *running within time* t*, its advantage in the above game with parameters* q_r *(number of* $\mathcal{D}$-refresh*-queries),* q_s *(number of* $\mathcal{M}$-get-state*,* $\mathcal{M}$-set-state*-queries),* γ^**, and* λ *is at most* ε*.*

Recovering Security Against Memory Attacks. We now describe our second security property. It states that if the adversary chooses a state that is later refreshed with random inputs such that sufficiently many entropy is accumulated into the internal state, the output of algorithm next should be undistinguishable from random. Note that even if $\mathcal{M}$-get-state and $\mathcal{M}$-set-state-queries are possible as in the security game MPRES, they are not allowed with compromised states, since it would make c drop to 0. In the recovery process, the entropy should almost always increase, but never drop to 0. The security game MRECOV uses procedures

proc. initialize$(\mathcal{D})$
seed $\xleftarrow{\$}$ setup;
$\sigma_0 \leftarrow 0$;
$b \xleftarrow{\$} \{0,1\}$;
FOR $k = 1$ TO q_r DO
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
END FOR
$k \leftarrow 0$;
$c \leftarrow 0$;
$S \leftarrow 0^n$;
OUTPUT seed, $(\gamma_k, z_k)_{k=1,\ldots,q_r}$

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. getinput
$k \leftarrow k + 1$
OUTPUT I_k

proc. set-state(S^*)
$S \leftarrow S^*$
$c \leftarrow 0$

proc. $\mathcal{D}$-refresh
$k \leftarrow k + 1$;
$S =$ refresh(S, I_k);
IF $c < \gamma^*$,
  $c = \min(c + \gamma_k, n)$

proc. $\mathcal{M}$-set-state(S, M, J)
$S \leftarrow \mathcal{M}$-set(S, M, J)
$c \leftarrow \max(0, c - \lambda)$
IF $c < \gamma^*$,
  $c \leftarrow 0$

proc. $\mathcal{M}$-get-state(S, J)
$c \leftarrow \max(0, c - \lambda)$
IF $c < \gamma^*$,
  $c \leftarrow 0$
OUTPUT $\mathcal{M}$-get(S, J)

proc. next-ror
$(S^{(0)}, R^{(0)}) \leftarrow$ next(S)
$(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$
OUTPUT $(S^{(b)}, R^{(b)})$, $(I_{k+1}, \ldots, I_{q_r})$

Figure 5.4 – Procedures in Security Game MRECOV(q_r, γ^*, λ)

described in Figure 5.4. The security game MRECOV is described as follows, with an adversary $\mathcal{A}$, a sampler $\mathcal{D}$ and bounds q_r, γ^*, λ:

1. The challenger generates a seed seed $\xleftarrow{\$}$ setup and a bit $b \xleftarrow{\$} \{0,1\}$ uniformly at random. It sets $\sigma_0 = 0$ and for $k = 1, \ldots, q_r$, it computes $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$, initializes $k = 0$, sets $S = 0$ and $c = 0$. It then gives back seed, S and the values $\gamma_1, \ldots, \gamma_{q_r}$ and $z_1, \ldots, z_{q_r}$ 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 $\mathcal{A}$ selects a state S^* and an integer d such that $k + d \leq q_r$ and $\gamma_{k+1} + \cdots + \gamma_{k+d} \geq \gamma^*$ and makes d calls to $\mathcal{D}$-refresh: for $j = 1, \ldots, d$, the challenger sets

the state to the chosen state and updates it with the corresponding d inputs $I_{k+1}, \cdots, I_{k+d}$ and updates $c \leftarrow c + \gamma_{k+1} + \cdots + \gamma_{k+d}$ sequentially.

4. The challenger allows queries to the oracles $\mathcal{M}$-get-state, and $\mathcal{M}$-set-state. These queries are processed, respectively, as $\{S \leftarrow \mathcal{M}\text{-set}(S, M, J); c \leftarrow c - \lambda;$ if $c < \gamma^*,$ then $c = 0\}$ and $\{c \leftarrow c - \lambda;$ if $c < \gamma^*,$ then $c = 0\}$, with the inputs (M, J) and J provided by the adversary. These queries are answered, respectively, by noting and by $\mathcal{M}\text{-get}(S, J)$;
5. Eventually, under the restriction that c never dropped to 0, the challenger sets $(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$ and generates $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$. It then gives $(S^{(b)}, R^{(b)})$ to the adversary, together with the next inputs $I_{k+1}, \ldots, I_{q_r}$ (if k was the number of refresh-queries asked up to this point);
6. The adversary $\mathcal{A}$ outputs a bit b^*.

The Recovering Security Against Memory Attacks is given in Definition 36.

Definition 36 (Recovering Security Against Memory Attacks)**.** *A pseudo-random number generator with input $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ is said $(t, q_r, q_s, \gamma^*, \lambda, \varepsilon)$-recovering against memory attacks, if for any adversary $\mathcal{A}$ and sampler $\mathcal{D}$, running within time t, its advantage in the above game with parameters q_r (number of $\mathcal{D}$-refresh-queries), q_s (number of $\mathcal{M}$-get-state / $\mathcal{M}$-set-state queries), γ^*, and λ is at most ε.*

As in Section 4.3, we prove now that the combination of recovering and preserving security, both against memory attacks, implies robustness against memory attacks.

Theorem 12. *If a pseudo-random number generator with input $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ has $(t, q_r, q_s, \gamma^*, \lambda, \varepsilon_r)$-recovering security and $(t, q_r, q_s, \gamma^*, \lambda, \varepsilon_p)$-preserving security, both against memory attacks, then it is $((t', q_r, q_s, q_n), \gamma^*, \lambda, q_n(\varepsilon_p + q_s\varepsilon_r))$-robust against memory attacks, where $t' \approx t$.*

Proof. The proof considers a hybrid sequence of security games: G_0 is the initial robustness game MROB, G_i, $\mathsf{G}_{i+\frac{1}{2}}$ and G_{i+1} are hybrid games, all derived from the initial robustess game MROB, for $i \in \{0, \ldots, q_n - 1\}$.

Game G_0. Game G_0 is the initial security game MROB as defined in Figure 5.1.

Games G_i and G_{i+1}. Games G_i and G_{i+1} are modifications of game G_0 that use the following procedures. Procedure initialize sets parameters as in G_0 and a new parameter ctr to 0. Procedures finalize, $\mathcal{D}$-refresh, $\mathcal{M}$-set-state, $\mathcal{M}$-get-state are the same as in game G_0. Procedure next-ror is different from game G_0: for each uncompromised next query, ctr is incremented, and if $\mathsf{ctr} \leq i$ (for G_i) *or* $\mathsf{ctr} \leq i + 1$ (for G_{i+1}), the challenger generates a random couple $(S_1, R_1) \in \{0,1\}^{n+\ell}$ and returns R_1 to $\mathcal{A}$. If $\mathsf{ctr} > i$ (for G_i) or $\mathsf{ctr} > i + 1$ (for G_{i+1}), the challenger behaves as in game G_0.

Game $\mathsf{G}_{i+\frac{1}{2}}$. Game $\mathsf{G}_{i+\frac{1}{2}}$ is a modification of game G_i that uses the following procedures. Procedure initialize sets parameters as in G_i and a new flag ns to true. Procedures finalize and $\mathcal{D}$-refresh are the same as for the previous games. Procedures $\mathcal{M}$-set-state and $\mathcal{M}$-get-state are different: if $c < \gamma^*$ during the procedure, the flag ns is set to false. Finally procedure next-ror is also different: for each uncompromised next query, ctr is incremented, and if $\mathsf{ctr} \leq i$ or $\mathsf{ctr} = i+1$ and flag $\mathsf{ns} = \mathsf{true}$, the challenger generates a random couple $(S_1, R_1) \in \{0,1\}^{n+\ell}$ and returns R_1 to $\mathcal{A}$. If $\mathsf{ctr} = i + 1$ or $\mathsf{ns} = \mathsf{false}$, the challenger behaves as in game G_0.

As in Section 4.3, we partition the sequence of next-ror queries done by an adversary $\mathcal{A}$ in this sequence of games into two sets: a next-ror query is said *uncompromised* if $c \geq \gamma^*$, and it is *compromised* otherwise. We then further partition the set of uncompromised next-ror queries into two subsets:

- First set : $c \geq \gamma^*$ throughout the entire period between the previous next-ror query and the current one. We name this next-ror query *preserving*.
- Second set : $c < \gamma^*$ in the period between the previous next-ror query and the current one. We name this next-ror query *recovering*.

In proposition 6, we show by reduction to the preserving security that $|\Pr[\mathsf{G}_{i+\frac{1}{2}} = 1] - \Pr[\mathsf{G}_i = 1]| \leq \varepsilon_p$ for all $i \in \{0, \ldots, q_n - 1\}$ and in proposition 7, we show by reduction to the recovering security that $|\Pr[\mathsf{G}_{i+1} = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| \leq \varepsilon_r$. Combining Propositions 6 and 7, we obtain that $|\Pr[(\mathsf{G}_0) = 1] - \Pr[(\mathsf{G}_{q_n}) = 1]| \leq q_n \cdot (\varepsilon_r + q_s \cdot \varepsilon_p)$. Moreover, G_{q_n} is independent of the challenge bit b' and therefore $\Pr[(\mathsf{G}_{q_n}) = 1] = \frac{1}{2}$, which finalizes the proof. □

Let us now prove the two reductions. We reuse the notion of mRED query introduced in Section 4.3, to identify the last query done by the adversary $\mathcal{A}$ during a recovering process, for which the the counter c has been set to 0. The main difference between the proof presented below and the one in Section 4.3 is that to identify the mRED query in Chapter 4, the adversary only needs to identify the last call to a get-state or to a set-state or to a compromised next-ror query, whereas here, the adversary needs to select between q_s $\mathcal{M}$-get-state, $\mathcal{M}$-set-state queries. This explains why the bound in Proposition 7 is equal to $q_s \cdot \varepsilon_r$ here and is equal to ε_r in Section 4.3.

Reduction to the preserving security. We build an adversary $(\mathcal{A}', \mathcal{D})$, with advantage ε_p in game MPRES, that uses $(\mathcal{A}, \mathcal{D})$ as a subroutine. In particular, it will simulate the game G_i (or $\mathsf{G}_{i+\frac{1}{2}}$) and will provide $(\mathcal{A}, \mathcal{D})$ inputs that follows the inputs distribution in the games G_i and $\mathsf{G}_{i+\frac{1}{2}}$.

Proposition 6. *Assuming that $\mathcal{G}$ has $(t, q_r, q_s, \gamma^*, \lambda, \varepsilon_p)$-preserving security against memory attacks, then for any adversary/distinguisher $\mathcal{A}, \mathcal{D}$ running in time $t' \approx t$, we have $|\Pr[\mathsf{G}_i = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| \leq \varepsilon_p$.*

Proof. Observe that if the $(i+1)^{th}$ uncompromised next-query of $(\mathcal{A}, \mathcal{D})$ is *recovering*, games G_i and $\mathsf{G}_{i+\frac{1}{2}}$ are identical, therefore we suppose that this is not the case and that the $(i+1)^{th}$ uncompromised next-query of $(\mathcal{A}, \mathcal{D})$ is *preserving*.

We construct an adversary $(\mathcal{A}', \mathcal{D})$ with advantage ε_p in game MPRES. Its challenger calls setup procedure to generate a parameter seed, a random state S' and a random bit b'. Then it generates the successive outputs of $\mathcal{D}$: $(\sigma_k, I_k, \gamma_k, z_k)_{k=1,\ldots,q_r}$, sends seed and $(\gamma_k, z_k)_{k=1,\ldots,q_r}$ to $(\mathcal{A}', \mathcal{D})$. Finally it sets the counter k to 0.

Attacker $(\mathcal{A}', \mathcal{D})$ uses the previously generated parameter seed, sets the sampler state σ to 0, picks a state S at random, sets a counter c to n and finally generates a random challenge bit b. Then $(\mathcal{A}', \mathcal{D})$ transfers seed to $(\mathcal{A}, \mathcal{D})$ and responds to the oracle queries from $(\mathcal{A}, \mathcal{D})$ in G_i, until the i^{th} uncompromised next-query. To answer these queries, $(\mathcal{A}', \mathcal{D})$ uses procedure $\mathcal{D}$-refresh, $\mathcal{M}$-get-state and $\mathcal{M}$-set-state from game MPRES:

- To answer $(\mathcal{A}, \mathcal{D})$'s $\mathcal{D}$-refresh queries, $(\mathcal{A}', \mathcal{D})$ updates S with algorithm refresh and sends the couple (γ_k, z_k) to $(\mathcal{A}, \mathcal{D})$. Finally $(\mathcal{A}', \mathcal{D})$ updates the counter c with γ_k.
- To answer $(\mathcal{A}, \mathcal{D})$'s $\mathcal{M}$-get-state queries, on input J, $(\mathcal{A}', \mathcal{D})$ computes $\mathcal{M}$-get(S, J) and sends the result to $(\mathcal{A}, \mathcal{D})$. To answer $(\mathcal{A}, \mathcal{D})$'s $\mathcal{M}$-set-state queries, on input (M, J), $(\mathcal{A}', \mathcal{D})$ updates S with function $\mathcal{M}$-get. Finally $(\mathcal{A}', \mathcal{D})$ updates the counter $c \leftarrow c - \lambda$, where $\lambda = |J|$, and if $c < \gamma^*$, $(\mathcal{A}', \mathcal{D})$ sets the counter c to 0.
- To answer compromised next-ror queries, $(\mathcal{A}', \mathcal{D})$ computes $(S_0, R_0) = \mathsf{next}(S)$ and sends R_0 to $(\mathcal{A}, \mathcal{D})$. Then $(\mathcal{A}', \mathcal{D})$ sets the counter c to 0.

- To answer the i^{th}uncompromised next-query, $(\mathcal{A}', \mathcal{D})$ generates (S_1, R_1) at random and sends R_1 to $(\mathcal{A}, \mathcal{D})$.

Recall that the $(i+1)^{th}$ uncompromised next-query of $(\mathcal{A}, \mathcal{D})$ is *preserving*. Hence between the $(i)^{th}$ and the $(i+1)^{th}$ uncompromised next queries, calls to $\mathcal{M}$-set-state, to $\mathcal{M}$-get-state and to $\mathcal{D}$-refresh are such that the counter c never decreases below γ^*, which ensures that it is never dropped to 0. Finally $(\mathcal{A}', \mathcal{D})$ calls procedure next-ror: its challenger computes $(S^0, R^0) = \mathsf{next}(S')$ and generates a random couple (S^1, R^1). It sends back a challenge couple $(S^{b'}, R^{b'})$ to $(\mathcal{A}', \mathcal{D})$. Then $(\mathcal{A}', \mathcal{D})$ responds to the $(i+1)^{th}$ uncompromised next-queries of $(\mathcal{A}, \mathcal{D})$ in game G_i. It sets $R^{b'} = R_0$, generates a random R_1 and transfers R_b to $(\mathcal{A}, \mathcal{D})$, which in returns a bit b^* to $(\mathcal{A}', \mathcal{D})$. Finally $(\mathcal{A}', \mathcal{D})$ returns 1 to $\mathcal{A}$ if $b = b^*$, and 0 otherwise and $(\mathcal{A}', \mathcal{D})$ finalizes game MPRES: it answers the bit $b'^* = 1$ if $b = b^*$. The possible cases are the following:

- If $b' = 0$, the challenger of $(\mathcal{A}', \mathcal{D})$ returns $(S^0, R^0) = \mathsf{next}(S')$. Then $(\mathcal{A}', \mathcal{D})$ sets $R_0 = R^0$, generates a random R_1 and returns R_b to $(\mathcal{A}, \mathcal{D})$. Therefore $(\mathcal{A}', \mathcal{D})$ exactly simulates the $(i+1)^{th}$ uncompromised next-queries of $\mathcal{A}$ in game G_i and $\Pr[\mathsf{G}_i = 1] = \Pr[b = b^* | b' = 0] = \Pr[b'^* = 1 | b' = 0]$.

- If $b' = 1$, the challenger of $(\mathcal{A}', \mathcal{D})$ returns $(S^1, R^1) \xleftarrow{\$} \{0,1\}^{n+\ell}$ to $(\mathcal{A}', \mathcal{D})$. Then $(\mathcal{A}', \mathcal{D})$ sets $R_0 = R^1$, generates a random R_1 and returns R_b to $(\mathcal{A}, \mathcal{D})$. Therefore $(\mathcal{A}', \mathcal{D})$ exactly simulates the $(i+1)^{th}$ uncompromised next-queries of $(\mathcal{A}, \mathcal{D})$ in game $\mathsf{G}_{i+\frac{1}{2}}$ and $\Pr[\mathsf{G}_{i+\frac{1}{2}} = 1] = \Pr[b = b^* | b' = 1] = \Pr[b'^* = 1 | b' = 1]$.

Finally the distance between games G_i and $\mathsf{G}_{i+\frac{1}{2}}$ satisfies:

$$|\Pr[\mathsf{G}_{i+\frac{1}{2}} = 1] - \Pr[\mathsf{G}_i = 1]| = 2 \cdot (\Pr[b'^* = 1 | b' = 1] - 1) = 2 \cdot (\Pr[b'^* = b'] - 1) \leq \varepsilon_p.$$

□

Reduction to the Recovering Security. We build an adversary $(\mathcal{A}', \mathcal{D})$, with advantage ε_r in game MRECOV, that uses $(\mathcal{A}, \mathcal{D})$ as a subroutine. In particular, it will simulate the game $\mathsf{G}_{i+\frac{1}{2}}$ (or G_{i+1}) and will provide $(\mathcal{A}, \mathcal{D})$ inputs that follow the inputs distribution in the games $\mathsf{G}_{i+\frac{1}{2}}$ and G_{i+1}.

Proposition 7. *Assuming that $\mathcal{G}$ has $(t, q_r, q_s, \gamma^*, \lambda, \varepsilon_r)$-recovering security against memory attacks, then for any adversary/distinguisher $\mathcal{A}, \mathcal{D}$ running in time $t' \approx t$, we have* $|\Pr[\mathsf{G}_{i+1} = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| \leq q_s \varepsilon_r$.

Proof. Observe that if the $(i+1)^{th}$ uncompromised next-query of $(\mathcal{A}, \mathcal{D})$ is *preserving*, games $\mathsf{G}_{i+\frac{1}{2}}$ and G_{i+1} are identical, therefore we suppose that this is not the case and that the $(i+1)^{th}$ uncompromised next-query of $(\mathcal{A}, \mathcal{D})$ is *recovering*.

We construct an adversary $(\mathcal{A}', \mathcal{D})$ with advantage ε_r in game MRECOV. Its challenger calls setup procedure to generate a parameter seed, generates a random state S' and a random bit b'. Then it generates the successive outputs of $\mathcal{D}$: $(\sigma_k, I_k, \gamma_k, z_k)_{k=1,\ldots,q_r}$, sends seed and $(\gamma_k, z_k)_{k=1,\ldots,q_r}$ to $(\mathcal{A}', \mathcal{D})$. Finally it sets a counter k to 0. Let $(\mathcal{A}', \mathcal{D})$ challenge $(\mathcal{A}, \mathcal{D})$: $(\mathcal{A}', \mathcal{D})$ uses the previously generated parameter seed, sets the sampler state σ to 0, sets a state S to 0^n, a counter c to 0 and finally generates a random challenge bit b. Then $(\mathcal{A}', \mathcal{D})$ sends seed to $(\mathcal{A}, \mathcal{D})$ and responds to the oracle queries from $(\mathcal{A}, \mathcal{D})$ in G_i, until the i^{th} uncompromised next-query. To answer these queries, $(\mathcal{A}', \mathcal{D})$ uses procedures getinput, $\mathcal{M}$-get-state and $\mathcal{M}$-set-state from game MRECOV:

- To answer $(\mathcal{A},\mathcal{D})$'s $\mathcal{D}$-refresh queries, $(\mathcal{A}',\mathcal{D})$ calls procedure getinput. Its challenger updates the counter k and sends the input I_k to $(\mathcal{A}',\mathcal{D})$. Then $(\mathcal{A}',\mathcal{D})$ updates S with algorithm refresh and sends the couple (γ_k, z_k) to $(\mathcal{A},\mathcal{D})$. Finally $(\mathcal{A}',\mathcal{D})$ updates the counter c with γ_k.
- To answer $(\mathcal{A},\mathcal{D})$'s $\mathcal{M}$-get-state queries, on input J, $(\mathcal{A}',\mathcal{D})$ calculates $\mathcal{M}$-get(S,J) and sends the result to $(\mathcal{A},\mathcal{D})$. To answer $(\mathcal{A},\mathcal{D})$'s $\mathcal{M}$-set-state queries, on input $(M,J$, $(\mathcal{A}',\mathcal{D})$ updates S with algorithm $\mathcal{M}$-get. Finally $(\mathcal{A}',\mathcal{D})$ updates the counter $c = c - \lambda$, where $\lambda = |J|$, and if $c < \gamma^*$, $(\mathcal{A}',\mathcal{D})$ sets the counter c to 0.
- To answer compromised next-ror queries, $(\mathcal{A}',\mathcal{D})$ calculates $(S_0, R_0) = \mathsf{next}(S)$ and sends R_0 to $(\mathcal{A},\mathcal{D})$. Then $(\mathcal{A}',\mathcal{D})$ sets the counter c to 0.
- To answer the i^{th}uncompromised next-query, $(\mathcal{A}',\mathcal{D})$ generates (S_1, R_1) at random and sends R_1 to $(\mathcal{A},\mathcal{D})$.

Recall that the $(i+1)^{th}$ uncompromised next-query of $(\mathcal{A},\mathcal{D})$ is *recovering*. Hence at least one call to a compromised $\mathcal{M}$-set-state, $\mathcal{M}$-get-state, or next-ror query was done by $(\mathcal{A},\mathcal{D})$ between the i^{th} and the $(i+1)^{th}$ uncompromised next-queries, interleaved with calls to $\mathcal{D}$-refresh. Attacker $(\mathcal{A}',\mathcal{D})$ identifies the mRED query, which is the *last* query to either $\mathcal{M}$-set-state, to $\mathcal{M}$-get-state or to a compromised next-ror done by $\mathcal{A}$ *before* the $(i+1)^{th}$ uncompromised next-query, names S_0 the state following this last query and I_{k_0} the last used input. To identify the mRED query, $(\mathcal{A},\mathcal{D})$ has at most q_s choices, as calls to $\mathcal{M}$-set-state and $\mathcal{M}$-get-state can also be done after the generator gets uncompromised. Starting from this query, the entropy is accumulated with calls to $\mathcal{D}$-refresh, hence $\mathcal{A}'$ can compute d, such that $\sum_{i=k_0+1}^{k_0+d} \gamma_i \geq \gamma^*$. Following, $(\mathcal{A}',\mathcal{D})$ divides the remaining $\mathcal{D}$-refresh queries of $(\mathcal{A},\mathcal{D})$ in two subsets : queries done with $(I_j), j = k_0{+}1, \cdots, k_0{+}d$ and queries done with $(I_j), j = k_0 + d + 1, \cdots, q_r$. Then $(\mathcal{A}',\mathcal{D})$ continues game MRECOV. Instead of getting access to the first set of inputs $(I_j), j = k_0 + 1 \cdots, k_0 + d$ as previously, with a getinput query, $(\mathcal{A}',\mathcal{D})$ makes d queries to $\mathcal{D}$-refresh. Challenger calculates the successive states $S_j = \mathsf{refresh}(S_{j-1}, I_{k_0+j})$, for $j = 1, \cdots, d$, calculates $(S^0, R^0) = \mathsf{next}(S_d)$ and generates a random couple (S^1, R^1). It sends back a challenge couple $(S^{b'}, R^{b'})$ to $\mathcal{A}'$. Then $(\mathcal{A}',\mathcal{D})$ computes S^* as the state $S^{b'}$ refreshed with inputs I_j, for $j = k_0 + d + 1, \cdots, q_r$, and responds to the $(i+1)^{th}$ uncompromised next-queries of $(\mathcal{A},\mathcal{D})$: it computes R_0 as the output of $\mathsf{next}(S^*)$, generates a random R_1 and transfers R_b to $(\mathcal{A},\mathcal{D})$, which returns with a bit b^*. Finally $(\mathcal{A}',\mathcal{D})$ returns 1 to $(\mathcal{A},\mathcal{D})$ if $b = b^*$, and 0 elsewhere and finalizes game MRECOV: it answers the bit $b'^* = 1$ if $b = b^*$ and the bit $b'^* = 0$ elsewhere. The possible cases are the following:

- If $b' = 0$, the challenger of $(\mathcal{A}',\mathcal{D})$ returns $(S^0, R^0) = \mathsf{next}(S_d)$ to $(\mathcal{A}',\mathcal{D})$. Then $(\mathcal{A}',\mathcal{D})$ computes S^* as the state S^1 refreshed with inputs I_j, for $j = k_0 + d + 1, \cdots, q_r$, computes R_0 as the output of $\mathsf{next}(S^*)$, generates a random R_1 and returns R_b to $\mathcal{A}$. Therefore $(\mathcal{A}',\mathcal{D})$ exactly simulates the $(i+1)^{th}$ uncompromised next-queries of $\mathcal{A}$ in game $\mathsf{G}_{i+\frac{1}{2}}$. Hence $\Pr[(\mathsf{G}_{i+\frac{1}{2}} = 1] = \Pr[b = b^* | b' = 0] = \Pr[b'^* = 1 | b' = 0]$.

- If $b' = 1$, the challenger of $(\mathcal{A}',\mathcal{D})$ returns $(S^1, R^1) \xleftarrow{\$} \{0,1\}^{n+\ell}$ to $(\mathcal{A}',\mathcal{D})$. Then $(\mathcal{A}',\mathcal{D})$ computes S^* as the state S^1 refreshed with inputs I_j, for $j = k_0 + d + 1, \cdots, q_r$, computes R_0 as the output of $\mathsf{next}(S^*)$, generates a random R_1 and returns R_b to $(\mathcal{A},\mathcal{D})$. Therefore $(\mathcal{A}',\mathcal{D})$ exactly simulates the $(i+1)^{th}$ uncompromised next-queries of $(\mathcal{A},\mathcal{D})$ in game G_{i+1}. Hence $\Pr[(\mathsf{G}_{i+1} = 1] = \Pr[b = b^* | b' = 1] = \Pr[b'^* = 1 | b' = 1]$.

Finally the distance between games $\mathsf{G}_{i+\frac{1}{2}}$ and G_{i+1} satisfies:

$$\Pr[\mathsf{G}_{i+1} = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| = 2 \cdot (\Pr[b'^* = 1 | b' = 1] - 1) = 2 \cdot (\Pr[b'^* = b'] - 1) \leq q_s \cdot \varepsilon_r.$$

□

5.4 A Secure Construction

Let us recall the robust construction described in Section 4.3. Let $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ be a standard pseudo-random generator where $m < n$. The robust pseudo-random number generator with input $\mathcal{G}$ has the parameters $s = 2n$ (seed length), n (state length), ℓ (output length), and $p = n$ (input length), and is defined as follows:

- $\mathsf{setup}()$: Output $\mathsf{seed} = (X, X') \xleftarrow{\$} \{0,1\}^{2n}$.
- $S' = \mathsf{refresh}(S, I)$: Given $\mathsf{seed} = (X, X')$, current state $S \in \{0,1\}^n$, and a sample $I \in \{0,1\}^n$, output: $S' := S \cdot X + I$, where all operations are over $\mathbb{F}_{2^n}$.
- $(S', R) = \mathsf{next}(S)$: Given $\mathsf{seed} = (X, X')$ and a state $S \in \{0,1\}^n$, first compute $U = [X' \cdot S]_1^m$. Then output $(S', R) = \mathbf{G}(U)$.

The following theorem extends Theorem 10.

Theorem 13. *Let $n > m, \ell, \gamma^*, \lambda$ be integers. Assume that $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a standard $(t, \varepsilon_{\mathbf{G}})$-secure pseudo-random generator. Let $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ be the pseudo-random number generator with input defined as above. Then $\mathcal{G}$ is $((t', q_r, q_n, q_s), \lambda, \gamma^*, q_n(2\varepsilon_{\mathbf{G}} + q_r^2 \cdot (1 + q_s)\varepsilon_{ext}))$-robust against memory attacks, where $t' \approx t$, as soon as $\gamma^* \geq m + q_s\lambda + 2\log(1/\varepsilon_{ext}) - 1$ and $n \geq m + q_s\lambda + \log(q_r) + 2\log(1/\varepsilon_{ext}) - 1$.*

Proof of Theorem 13. The proof has two parts, as in Section 4.3, we prove that the construction is preserving and recovering, both against memory attacks. Note that here, contrary to the Section 4.3, the strong extractor is used for the preserving and the recovering security.

Lemma 7. *Let $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ be defined as above. Then $\mathcal{G}$ has $(t, q_r, q_s, \lambda, \gamma^*, \varepsilon_{\mathbf{G}} + q_r^2\varepsilon_{ext})$-Preserving security as soon as $\gamma^* \geq m + 2\log(1/\varepsilon_{ext}) - 1$ and $n \geq m + q_s\lambda + \log(d) + 2\log(1/\varepsilon_{ext}) - 1$.*

Proof of Lemma 7. Consider games G_0, G_1 and G_2 as follows:

- G_0 is the original game MPRES applied to $\mathcal{G}$.
- G_1 is game G_0 in which the challenger computes $U \xleftarrow{\$} \{0,1\}^m$ instead of $U = [S_d \cdot X']_1^m$ inside the $\mathsf{next\text{-}ror}$ procedure.
- G_2 is game G_1 in which the challenger computes $(S^0, R^0) \xleftarrow{\$} \{0,1\}^{n+\ell}$ instead of $(S^0, R^0) \leftarrow \mathbf{G}(U)$ inside the $\mathsf{next\text{-}ror}$ procedure, when challenge bit $b = 0$.

Distance between G_0 and G_1. Recall that in the security game MPRES, adversary can ask as many queries as it wants to the oracle $\mathcal{D}$-$\mathsf{refresh}$, interleaved with queries to the oracles $\mathcal{M}$-$\mathsf{set\text{-}state}$, and $\mathcal{M}$-$\mathsf{get\text{-}state}$, where each oracle $\mathcal{D}$-$\mathsf{refresh}$ query increases the counter ($c \leftarrow c + \gamma_k$) and each $\mathcal{M}$-$\mathsf{set\text{-}state}$ or $\mathcal{M}$-$\mathsf{get\text{-}state}$ query decreases it ($c \leftarrow c - \lambda$). Then as the adversary makes at most q_s $\mathcal{M}$-$\mathsf{set\text{-}state}$ / $\mathcal{M}$-$\mathsf{get\text{-}state}$ queries, by Lemma 6, the complete sequence of calls before the call to $\mathsf{next\text{-}ror}$ followed by the m-truncation leads to a $(\gamma, \varepsilon_{ext})$-randomness extractor as soon as :

$$\gamma \geq m + 2\log(1/\varepsilon_{ext}) - 1 \text{ and } n \geq m + q_s\lambda + 2\log(1/\varepsilon_{ext}) + \log(d) - 1. \ (*)$$

Finally, as adversary $\mathcal{A}$ has q_r possibilities to choose when to start and stop the $\mathcal{D}$-$\mathsf{refresh}$ queries, there is a loss of q_r^2. With conditions above, $|\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_0 = 1]| \leq q_r^2\varepsilon_{ext}$.

Distance between G_1 and G_2. Since $\mathbf{G}$ is a $(t, \varepsilon_{\mathbf{G}})$-secure standard pseudo-random number

generator, we can replace both the output and the random state by truly random values. Then we have $|\Pr[\mathsf{G}_2 = 1] - \Pr[\mathsf{G}_1 = 1]| \leq \varepsilon_{\mathbf{G}}$.

Distance between G_0 and $\frac{1}{2}$. From the above games, one gets, $|\Pr[\mathsf{G}_2 = 1] - \Pr[\mathsf{G}_0 = 1]| \leq q_s \cdot q_r^2 \cdot \varepsilon_{ext} + \varepsilon_{\mathbf{G}}$, and as $\Pr[\mathsf{G}_2 = 1] = \frac{1}{2}$, $|\Pr[\mathsf{G}_0 = 1] - \frac{1}{2}| \leq q_r^2 \varepsilon_{ext} + \varepsilon_{\mathbf{G}}$, as soon as conditions $(*)$ on n, m, γ and λ are satisfied. □

Lemma 8. *Let $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ be defined as above. Then $\mathcal{G}$ has $(t, q_r, \gamma^*, \varepsilon_{\mathbf{G}} + q_r^2 \varepsilon_{ext})$-Recovering security as soon as $\gamma^* \geq m + q_s\lambda + 2\log(1/\varepsilon_{ext}) - 1$ and $n \geq m + \log(d) + 2\log(1/\varepsilon_{ext}) - 1$.*

Proof. Consider games G_0, $\mathsf{G}1$ and G_2 as follows:

- G_0 is the original game MRECOV applied to $\mathcal{G}$.
- G_1 is game G_0 in which the challenger computes $U \xleftarrow{\$} \{0,1\}^m$ instead of $U = [S_d \cdot X']_1^m$ inside the $\mathsf{next\text{-}output\text{-}ror}$ procedure.
- G_2 is game G_1 in which the challenger computes $(S^0, R^0) \xleftarrow{\$} \{0,1\}^{n+\ell}$ instead of $(S^0, R^0) \leftarrow \mathbf{G}(U)$ inside the $\mathsf{next\text{-}ror}$ procedure, when challenge bit $b = 0$.

Distance between G_0 and G_1. Recall that in the security game MRECOV, adversary sets a chosen state, then is allowed to make a sequence of calls to $\mathcal{D}$-$\mathsf{refresh}$ (the recovery sequence), followed by one sequence of $\mathcal{D}$-$\mathsf{refresh}$ interleaved with calls to $\mathcal{M}$-$\mathsf{set\text{-}state}$ or $\mathcal{M}$-$\mathsf{get\text{-}state}$, followed by one call to $\mathsf{next\text{-}ror}$. where each oracle $\mathcal{D}$-$\mathsf{refresh}$ query increase the counter $c \leftarrow c + \gamma_k$, and each $\mathcal{M}$-$\mathsf{set\text{-}state}$ or $\mathcal{M}$-$\mathsf{get\text{-}state}$ query decrease it $c \leftarrow c - \lambda$. Then as the adversary makes at most q_s $\mathcal{M}$-$\mathsf{set\text{-}state}$ / $\mathcal{M}$-$\mathsf{get\text{-}state}$ queries, by Lemma 6, the complete sequence of calls before the call to $\mathsf{next\text{-}ror}$ followed by the m-truncation leads to a $(\gamma, \varepsilon_{ext})$-randomness extractor as soon as :

$$\gamma \geq m + q_s\lambda + 2\log(1/\varepsilon_{ext}) - 1 \text{ and } n \geq m + 2\log(1/\varepsilon_{ext}) + \log(d) - 1. \ (**)$$

Finally, as adversary $\mathcal{A}$ has q_r possibilities to choose when to start and stop the $\mathcal{D}$-$\mathsf{refresh}$ queries and has q_s possibilities to choose q, there is a loss of $q_s q_r^2$. With conditions above, $|\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_0 = 1]| \leq q_s q_r^2 \varepsilon_{ext}$.

Distance between G_1 and G_2. Since $\mathbf{G}$ is a $(t, \varepsilon_{\mathbf{G}})$-secure standard pseudo-random generator, we can replace both the output and the random state by truly random values. And we have $|\Pr[\mathsf{G}_2 = 1] - \Pr[\mathsf{G}_1 = 1]| \leq \varepsilon_{\mathbf{G}}$.

Distance between G_0 and $\frac{1}{2}$. From the above games, one gets, $|\Pr[\mathsf{G}_2 = 1] - \Pr[\mathsf{G}_0 = 1]| \leq q_r^2 \cdot \varepsilon_{ext} + \varepsilon_{\mathbf{G}}$, and as $\Pr[\mathsf{G}_2 = 1] = \frac{1}{2}$, $|\Pr[\mathsf{G}_0 = 1] - \frac{1}{2}| \leq q_s q_r^2 \varepsilon_{ext} + \varepsilon_{\mathbf{G}}$, as soon as conditions $(**)$ are satisfied. □

Let now finalize the proof of Theorem 13. We can divide the set of next calls done by $\mathcal{A}$ between *recovering* and *preserving*. From Lemma 7 and 8, $\mathcal{G}$ has:

- $(t, q_r, q_s, \lambda, \gamma^*, \varepsilon_{\mathbf{G}} + q_r^2 \varepsilon_{ext})$-Preserving security as soon as $\gamma^* \geq m + 2\log(1/\varepsilon_{ext}) - 1$ and $n \geq m + q_s\lambda + \log(q_r) + 2\log(1/\varepsilon_{ext}) - 1$.
- $(t, q_r, q_s, \lambda, \gamma^*, \varepsilon_{\mathbf{G}} + q_r^2 \varepsilon_{ext})$-Recovering security as soon as $\gamma^* \geq m + q_s\lambda + 2\log(1/\varepsilon_{ext}) - 1$, and $n \geq m + \log(q_r) + 2\log(1/\varepsilon_{ext}) - 1$.

By Theorem 12, we get that $\mathcal{G}$ is $((t', q_r, q_n, q_s), \lambda, \gamma^*, q_n(1+q_s)(\varepsilon_{\mathbf{G}} + q_r^2\varepsilon_{ext}))$-robust against memory attacks, where $t' \approx t$, as soon as $\gamma^* \geq m + q_s\lambda + 2\log(1/\varepsilon_{ext}) - 1$ and $n \geq m + q_s\lambda + \log(q_r) + 2\log(1/\varepsilon_{ext}) - 1$.

□

As in Section 4.3, to slightly reduce the number of parameters in Theorem 13, we can let k be our "security parameter" and set $q_r = q_n = 2^k$ and $\varepsilon_{ext} = 2^{-6k}$. Then we can set $n = m + q_s\lambda + \log(q_r) + 2\log(1/\varepsilon_{ext}) - 1 = m + q_s\lambda + 11k - 1$ and $\gamma^* \geq m + q_s\lambda + 2\log(1/\varepsilon_{ext}) - 1 = m + q_s\lambda + 10k - 1$. Summarizing all of these, we get Theorem 14.

Theorem 14. *Let k, m, ℓ, n be integers, where $n \geq m + q_s\lambda + 11k - 1$. Assume that $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a standard $(t, \varepsilon_{\mathbf{G}})$-secure pseudo-random generator. Then $\mathcal{G}$ is a $((t', 2^k, 2^k, q_s), m + q_s\lambda + 10k - 1, \lambda, 2^k(1+2^k)\cdot\varepsilon_{\mathbf{G}} + 2^{-k} + 2^{-2k})$-robust pseudo-random number generator with input against memory attacks, having n-bit state and ℓ-bit output, where $t' \approx t$.*

5.5 Instantiation

Table 5.1 – Security Bounds for the Robustness of $\mathcal{G}$ against Memory Attacks

$q_s \in \{1,2,4\}$	$k \in \{40,64\}$	$\lambda = 32, 64$	n	γ^*
1	40	32	599	559
		64	631	591
	64	32	863	799
		64	895	831
2	40	32	631	591
		64	665	625
	64	32	895	831
		64	959	895
4	40	32	695	655
		64	823	783
	64	32	959	895
		64	1087	1023

We recall that our construction is based on Section 4.3: $\mathsf{refresh}(S, I) = S \cdot X + I \in \mathbb{F}_{2^n}$ and $\mathsf{next}(S) = \mathbf{G}(U)$, with $U = [X' \cdot S]_1^m$. In Section 4.3, the standard pseudo-random generator $\mathbf{G}$ is defined by $\mathbf{G}(U) = \mathsf{AES}_U(0) \| \ldots \| \mathsf{AES}_U(\nu - 1)$, where ν is the number of calls to AES with a 128-bit key U, and thus $m = 128$. For a security parameter $k = 40$, the security analysis leads to $n = 489$, $\gamma^* = 449$, and $\nu = 5$. We now apply Theorem 14 and we obtain the following bounds:

- For $k = 40$, $n = 567 + q_s\lambda$, $\gamma^* = 527 + q_s\lambda$.
- For $k = 64$, $n = 831 + q_s\lambda$, $\gamma^* = 767 + q_s\lambda$.

Finally, for $q_s \in \{1,2,4\}$, $k \in \{40,64\}$ and $\lambda \in \{32,64\}$, concrete security bounds for $\mathcal{G}$ are given in Table 5.1. Hence:

- for $q_s = 1$, $\lambda = 32$, $\mathcal{G}$ is $((t', 2^{40}, 2^{40}, 1), 32, 559, 2^{-40})$-robust against memory attacks, if $n = 599$,

- for $q_s = 4$, $\lambda = 64$, $\mathcal{G}$ is $((t', 2^{64}, 2^{64}, 4), 64, 1023, 2^{-64})$-robust against memory attacks, if $n = 1087$ (based on the discussion about the security of AES in Section 4.5).

Chapter 6

Robustness Against Side-Channel Attacks

6.1 Model Description

In this chapter we give a syntactic formalization for security of pseudo-random number generators with input against memory attacks. All statements are part of [ABP+15]. We use Definition 27 for pseudo-random number generator with input in all this chapter.

Recall that in the robustness security model ROB, in Section 4.1, 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, we give the adversary the choice of the leakage functions, that we globally name f, associated to each algorithm, or even each small block. Since we restrict our model to non-adaptive leakage, we ask the adversary to choose them beforehand. So they are provided as input to the initialize procedure by the adversary (see Figure 6.1). Then, each leakage function will be implicitly used by our two new procedures named leak-refresh and leak-next that, in addition to the usual outputs, also provide some leakage L about the manipulated data, as described in Section 3.7.1. We thus have a new parameter λ, that bounds the output length of the leakage function. Our new *Leakage-Resilient Robustness* security game $\mathsf{LROB}(\gamma^*, \lambda)$ makes use of the procedures described in Figure 6.1 and is described in details below:

- The parameter γ^*, the variable c, and the Boolean flag/function compromised are the same as for the basic robustness ROB;
- The new parameter λ sets the maximal information leakage which can be collected during the execution of operations refresh and next. Namely, for each operation (refresh or next), the leakage functions globally output at most λ bits. Such a leakage will be available when querying the leaking procedures leak-refresh and leak-next below;
- The new parameter α is an integer that models the minimal expected entropy of S after a leak-next (next with leakage) call, in a safe case (compromised is false), that is when the entropy of the internal state was assumed greater than γ^*. This captures both the creation of computational entropy during a next execution and the smaller loss of entropy caused by the leakage. We could expect $\alpha = n - \lambda$, but it may depend on the explicit construction;

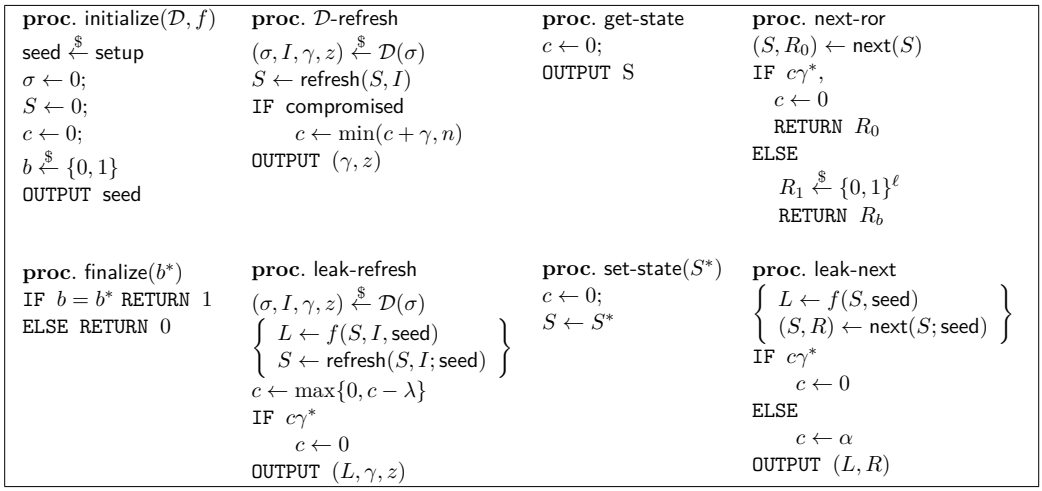

Figure 6.1 – Procedures in the Security Game LROB(γ^*, λ)

- The procedures initialize$(\mathcal{D}, f)$/finalize(b^*) initiate the security game with the additional leakage function f, check whether the adversary has won the game and output 1 in this case or 0 otherwise. As in the security game ROB, the initial state S is here set to zero (as well as the entropy counter) so that no assumption needs to be made on its initialization;
- The procedures get-state/set-state, $\mathcal{D}$-refresh, and next-ror are the same as for the basic robustness ROB;
- The procedure leak-refresh runs the refresh algorithm but additionally provides some information leakage L on the input (S, I) and seed, as above. As for the next-ror-queries, the leakage can reveal non-trivial information about a weak internal state even before the effectiveness of the refresh, and then we reduce c by λ bits. And if it drops below the threshold γ^*, it is reset to 0. Again, we could have strengthened this definition, but we preferred to keep a conservative notion. Furthermore, this strict notion is important w.r.t. our new definitions of recovering and preserving security with leakage. 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 λ bits, hence the notation $\{\ldots\}$, since they can be interleaved.
- The procedure leak-next 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 status was safe, then the new entropy estimate c is set to α, otherwise, it is reset to 0 (as for the next-ror). As above, if the next algorithm is complex, several leakage functions can be defined at each step, but the global leakage is limited to λ bits.

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. Examples of the entropy traces for the procedures defined in our new model are provided in Figure 6.2 (to be compared with the traces presented in Figure 4.3). The threshold γ^* has to be slightly higher in our new 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 λ. Now we detailed the new security game, we can define the notion of leakage-resilient robustness of a pseudo-random number generator with input.

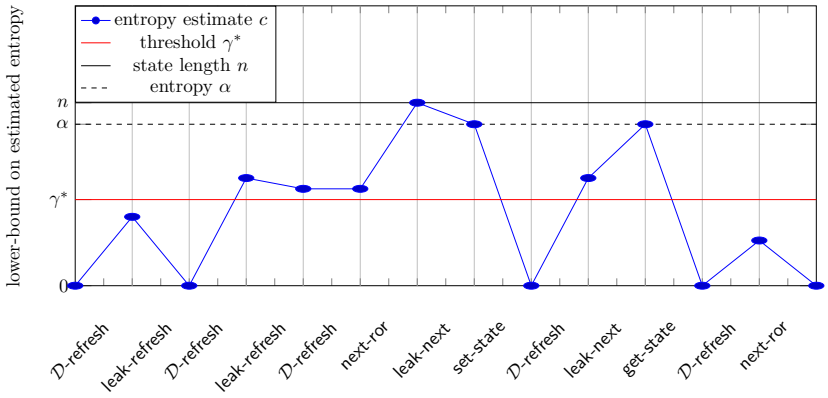

Figure 6.2 – Entropy Estimates in $\mathsf{LROB}(\gamma^*, \lambda)$

Definition 37 (Leakage-Resilient Robustness of Pseudo-Random Number Generator with Input)**.** *A pseudo-random number generator with input* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *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 after makes at most* q_r *calls to* $\mathcal{D}$-refresh/leak-refresh*,* q_n *calls to* next-ror/leak-next*, and* q_s *calls to* get-state/set-state *with a leakage bounded by* λ*, the advantage of* $\mathcal{A}$ *in game* $\mathsf{LROB}(\gamma^*, \lambda)$ *is at most* ε*.*

6.2 Analysis and Limitation of the Original Construction

Let us recall the robust construction described in Section 4.3, named $\mathcal{G}$. It makes use of a (t, ε)-secure standard pseudo-random generator $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$. The seed is a pair (X, X') of length $2n$, n is the state length, ℓ is the output length, and $p = n$ is the input length. This construction uses iterated multiplication and addition in the finite field $\mathbb{F}_{2^n}$ to refresh the internal state because it gives a proven seeded extractor that accumulates entropy, which we do not know how to do with a hash function. Plus, it is more efficient:

- $\mathsf{setup}()$ outputs $\mathsf{seed} = (X, X') \leftarrow \{0,1\}^{2n}$;
- $S' = \mathsf{refresh}(S, I; X) = S \cdot X + I$, where all operations are over $\mathbb{F}_{2^n}$;
- $(S', R) = \mathsf{next}(S; X') = \mathbf{G}(U)$, where $U = [X' \cdot S]_1^m$, the truncation of $X' \cdot S$.

Unfortunately, even a secure standard pseudo-random generator is not enough to resist to information leakage. We indeed first exhibit a counterexample, to show that the use of a standard secure pseudo-random number generator can lead to a construction that is vulnerable to side-channel attacks. But then, we prove that with a stronger security property for the standard pseudo-random number generator, namely leakage-resilience, the whole construction remains secure even in the presence of leakage.

In Section 4.5, we instantiate the generator $\mathbf{G}$ with the pseudo-random function AES in counter mode with the truncated product U as the secret key. Depending on the parameters, several calls to the pseudo-random function are required. We show hereafter that when the implementation is leaking, this construction faces vulnerabilities.

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 essentially performs a differential power analysis (DPA) attack. Then, for the above construction, during a leak-next, 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 leak-next, can thus be recovered, whereas it is considered as safe in the security game. A next-ror challenge can then be easily broken.

Furthermore, even if one uses only a few executions with the same key, with a counter as input, the adversary can predict future outputs. This vulnerability applies to AES with predictable inputs. As determined by the security games, the adversary chooses a leakage function $f_{\mathsf{next},\Pi}$ to further collect the leakage during the product and the truncation between the internal state S and the public seed X'. Assume that this function is $f_{\mathsf{next},\Pi}(S, X') =$

$$\left[\mathsf{AES}\left(\left[X' \cdot \left(\mathsf{AES}_{[X'.S]_1^m}(C_0)||\ldots||\mathsf{AES}_{[X'.S]_1^m}(C_0+\lceil\frac{n}{m}\rceil-1)\right)\right]_1^m\right)\left(C_0+\left\lceil\frac{n+\ell}{m}\right\rceil\right)\right]_1^\lambda$$

with C_0 an integer arbitrarily chosen by the attacker. With this leakage function set, the adversary makes a set-state-call and fix the counter C to C_0. This counter is a part of the global internal state. Even if this is not the random pool considered by S, this is under the control of the adversary. As the internal state is now compromised, sufficient calls to $\mathcal{D}$-refresh are made to refresh S so that its entropy increases above the threshold γ^*. Then, the attacker can ask a leak-next-query and gets back the leakage $f_{\mathsf{next},\Pi}(S, X')$ described above. Eventually, the attacker asks a challenge next-ror-query, and either gets the real output or a random one. The λ bits it got from the leakage are exactly the first λ bits of the real output. The attacker has consequently a significant advantage in the next-ror challenge.

6.3 Recovering and Preserving Security With Leakage

In this section, we adapt the notions of *recovering* and *preserving* introduced in Section 4.2 to capture side channel attacks. The former essentially deals with the capacity for the generator to accumulate the entropy from the inputs in the internal state, with the refresh algorithm, and then to recover a safe state even after being compromised. The latter deals with the quality of the internal state, even with adversarially chosen and known inputs. The quality of the internal state will then be measured by the ability of the adversary to distinguish the output randomness (by the next algorithm) from a truly random output, using one next-ror query. Since more oracles are available, contrarily to Section 4.2, our security games will be interactive and adaptive: the leak-refresh and leak-next oracles are available during the recovering and preserving sequences, and not just the $\mathcal{D}$-refresh oracle.

Recovering Security with Leakage. It 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 learning information with the collected leakage or with the output (leak-refresh, leak-next or 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 γ^*. This is the *recovering* process, which should make the bit b involved in the next-ror procedure indistinguishable: when the internal state is considered as safe, the output randomness R should look indistinguishable from random. The security game is the following, where $\mathcal{D}$ is the distribution sampler, and $f = (f_{\mathsf{refresh}}, f_{\mathsf{next}})$ denotes the union of the leakage functions related to the execution of refresh and next, both proposed by the adversary.

Even if leak-refresh and leak-next-queries are possible, they are not allowed with compromised states, since it would make c drop to 0. In the recovery process, the entropy should almost always increase, but never drop to 0.

proc. initialize($\mathcal{D}$)
seed $\xleftarrow{\$}$ setup;
$\sigma_0 \leftarrow 0$;
$S \xleftarrow{\$} \{0,1\}^n$ $b \xleftarrow{\$} \{0,1\}$;
FOR $k = 1$ TO q_r DO
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
END FOR
$k \leftarrow 0$;
OUTPUT seed, $(\gamma_k, z_k)_{k=1,\ldots,q_r}$

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. getinput
$k \leftarrow k + 1$
OUTPUT I_k

proc. set-state(S^*)
$S \leftarrow S^*$
$c \leftarrow 0$

proc. $\mathcal{D}$-refresh
$k \leftarrow k + 1$;
$S = \mathsf{refresh}(S, I_k)$;
$c = \min(c + \gamma_k, n)$

proc. leak-refresh
$k \leftarrow k + 1$;
$L = f_{\mathsf{refresh}}(S, I_k, \mathsf{seed})$;
$S = \mathsf{refresh}(S, I_k)$;
$c = \max(0, c - \lambda)$;
IF $c < \gamma^*$,
  $c = 0$
RETURN L

proc. leak-next
$L = f_{\mathsf{next}}(S, \mathsf{seed})$;
$(S, R) = \mathsf{next}(S)$;
IF $c < \gamma^*$,
  $c = 0$
ELSE $c = \alpha$
RETURN (L, R)

proc. next-ror
$(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$
$(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$
RETURN $(S^{(b)}, R^{(b)})$, $(I_{k+1}, \ldots, I_{q_r})$

Figure 6.3 – Procedures in Security Game $\mathsf{LRECOV}(q_r, q_n, \lambda, \gamma^*)$

1. The challenger generates a seed $\mathsf{seed} \xleftarrow{\$} \mathsf{setup}$ and a bit $b \xleftarrow{\$} \{0,1\}$ uniformly at random. It sets $\sigma_0 = 0$ and for $k = 1, \ldots, q_r$, 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 $\gamma_1, \ldots, \gamma_{q_r}$ and $z_1, \ldots, z_{q_r}$ 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 $\mathcal{A}$ outputs a new internal state $S_0 \in \{0,1\}^n$ and an integer d such that $k + d \leq q_r$ and $\gamma_{k+1} + \cdots + \gamma_{k+d} \geq \gamma^*$ and makes d calls to one-refresh with the d inputs $I_{k+1}, \cdots, I_{k+d}$: for $j = 1, \ldots, d$, the challenger updates the state $S_j := \mathsf{refresh}(S_{j-1}, I_{k+j})$ and updates $c \leftarrow c + \gamma_k$ sequentially.

4. The challenger allows queries to the oracles leak-refresh, and leak-next. These queries are processed, respectively, as $\{L_j = f_{\mathsf{refresh}}(S_{j-1}, I_k, \mathsf{seed}); S_j = \mathsf{refresh}(S_{j-1}, I_k); c = c - \lambda$; if $c < \gamma^*$, then $c = 0\}$, and $\{L_j = f_{\mathsf{next}}(S_{j-1}, \mathsf{seed}); (S_j, R_j) = \mathsf{next}(S_{j-1})$; if $c < \gamma^*$, then $c = 0$, else $c = \alpha\}$, with the new input I_k provided by the distribution sampler for the k^{th} refresh-query. These queries are answered, respectively, by the information leakage L, and by the information leakage L together with the randomness R;

5. Eventually, under the restriction that c never dropped to 0, the challenger sets $(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$ and generates $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$. It then gives $(S^{(b)}, R^{(b)})$ to the adversary, together with the next inputs $I_{k+1}, \ldots, I_{q_r}$ (if k was the number of refresh-queries asked up to this point);

6. The adversary $\mathcal{A}$ outputs a bit b^*.

In this game, we define the advantage of the adversary $\mathcal{A}$ as $|2\Pr[b^* = b] - 1|$. Note that we restrict our game to executions where c never dropped to 0, but one could have answered independently to b otherwise (e.g., always using $(S^{(0)}, R^{(0)})$).

Definition 38 (Recovering Security with Leakage)**.** *A pseudo-random number generator with input is said* $(t, q_r, q_n, \gamma^*, \lambda, \varepsilon)$*-recovering with leakage if for any adversary* $\mathcal{A}$ *running within time* t*, its advantage in the above game with parameters* q_r *(number of* $\mathcal{D}$-refresh *and* leak-refresh*-queries),* q_n *(number of* leak-next*-queries),* γ^**, and* λ *is at most* ε*.*

Preserving Security with Leakage. This security notion considers a safe internal state. After several calls to $\mathcal{D}$-refresh and leak-refresh with known (and even chosen) inputs, the internal state should remain safe. An initial state S_0 is generated with entropy n. Then it is refreshed with arbitrary many calls to either $\mathcal{D}$-refresh or leak-refresh, as long as the leakage does not decrease the entropy below the threshold γ^*. 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 safe, the output randomness R should look indistinguishable from random. The security game

proc. initialize($\mathcal{D}$)	**proc.** $\mathcal{D}$-refresh(I)	**proc.** leak-refresh(I)	**proc.** next-ror
seed $\xleftarrow{\$}$ setup; $S \xleftarrow{\$} \{0,1\}^n$; $b \xleftarrow{\$} \{0,1\}$; OUTPUT seed	$S = \mathsf{refresh}(S, I)$	$L = f_{\mathsf{refresh}}(S, I, \mathsf{seed})$; $S = \mathsf{refresh}(S, I)$; $c = \max(0, c - \lambda)$; IF $c < \gamma^*$, $c = 0$ RETURN L	$(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$ $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$ RETURN $(S^{(b)}, R^{(b)})$
proc. finalize(b^*) IF $b = b^*$ RETURN 1 ELSE RETURN 0		**proc.** leak-next $L = f_{\mathsf{next}}(S, \mathsf{seed})$; $(S, R) = \mathsf{next}(S)$; IF $c < \gamma^*$, $c = 0$ ELSE $c = \alpha$ RETURN (L, R)	

Figure 6.4 – Procedures in Security Game $\mathsf{LPRES}(q_r, q_n, \gamma^*, \lambda)$

is the following, where $\mathcal{D}$ is the distribution sampler, and $f = (f_{\mathsf{refresh}}, f_{\mathsf{next}})$ denotes the union of the leakage functions during the execution of refresh and next, both proposed by the adversary.

1. The challenger generates an initial state $S_0 \xleftarrow{\$} \{0,1\}^n$, a seed seed $\leftarrow$ setup, and a bit $b \xleftarrow{\$} \{0,1\}$ uniformly at random. It sets $c = n$ and then 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, leak-refresh, and leak-next, but with chosen inputs I to the refresh-queries. These queries are thus processed, respectively, as $\{S_j = \mathsf{refresh}(S_{j-1}, I)\}$, $\{L_j = f_{\mathsf{refresh}}(S_{j-1}, I, \mathsf{seed}); S_j = \mathsf{refresh}(S_{j-1}, I); c = c - \lambda;$ if $c < \gamma^*$, then $c = 0\}$, and $\{L_j = f_{\mathsf{next}}(S_{j-1}, \mathsf{seed}); (S_j, R_j) = \mathsf{next}(S_{j-1});$ if $c < \gamma^*$, then $c = 0$, else $c = \alpha\}$, with the input I provided by the adversary. Theses queries are answered, respectively, by nothing, by the information leakage L_j, and by the information leakage L_j together with the randomness R_j;

3. Eventually, under the restriction that c never dropped to 0, the challenger sets $(S^{(0)}, R^{(0)}) \leftarrow \mathsf{next}(S)$, and generates $(S^{(1)}, R^{(1)}) \xleftarrow{\$} \{0,1\}^{n+\ell}$. It then gives $(S^{(b)}, R^{(b)})$ to the adversary;

4. The adversary $\mathcal{A}$ outputs a bit b^*.

As above, we define the advantage of the adversary $\mathcal{A}$ as $|2\Pr[b^* = b] - 1|$, and the restriction that c never dropped to 0 could have been dealt another way.

Definition 39 (Preserving Security with Leakage)**.** *A pseudo-random number generator with input is said* $(t, q_r, q_n, \gamma^*, \lambda, \varepsilon)$*-preserving with leakage if for any adversary* $\mathcal{A}$ *running within time* t*, its advantage in the above game with parameters* q_r *(number of* $\mathcal{D}$-refresh *and* leak-refresh*-queries),* q_n *(number of* leak-next*-queries),* γ^**, and* λ *is at most* ε*.*

From these two security notions, one can prove the following theorem, inspired from the analysis presented in Section 4.2.

Theorem 15. *If a pseudo-random number generator with input is* $(t, q_r, q_n, \gamma^*, \lambda, \varepsilon_r)$*-recovering with leakage and* $(t, q_r, q_n, \gamma^*, \lambda, \varepsilon_p)$*-preserving with leakage then it is also* $(t', q_r, q_n, q_s, \gamma^*, \lambda, q_n \cdot (\varepsilon_r + q \cdot \varepsilon_p))$*-leakage-resilient robust where* $t' \approx t$*, where the adversary can ask at most* $q = q_r + q_n + q_s$ *queries, where* q_r *is the number of calls to* $\mathcal{D}$-refresh/leak-refresh*,* q_n *the number of calls to* next-ror/leak-next*, and* q_s *the number of calls to* get-state/set-state*.*

Proof. This proof follows the one presented in Section 4.2. It splits the leakage-resilient robustness game in preserving with leakage steps and recovering with leakage steps.

Queries. In the game of leakage-resilient robustness, we term next-*queries* the calls to the oracle next-ror. Since q_n is a bound on the next-ror and leak-next queries, this is also a bound on the next-queries. Actually, there are unsafe/compromised next-queries, when the internal state is unsafe and so the entropy estimate c is below the threshold γ^* before the query and reset to 0 after the query, and safe/uncompromised next-queries, when the internal state is safe before the next-ror-query.

For uncompromised next-queries, the output randomness R should look indistinguishable from truly random, while for compromised next-queries, there is no guarantee. As in Section 4.2, we split the uncompromised next-queries in two sets: the *preserving with leakage* queries, if the entropy estimate is above the threshold γ^* since the previous next-query; and the *recovering with leakage* queries, if the entropy estimate dropped below the threshold γ^* and has thus been reset to 0. For the latter queries, we reuse the notion of mRED (most recent entropy drain) to define the most recent query to one of the oracles get-state, set-state, leak-refresh or leak-next that reset c to 0.

Sequence of Games. Let us now define the sequence of games. Let game G_0 be the initial real-or-random game. Game G_i modifies the first i uncompromised next-queries by outputting a uniformly random R, and by setting the internal state S uniformly at random. We note that G_{q_n} is then independent of b, since all the safe next-ror-queries are answered randomly, while the unsafe next-ror-queries are anyway always answered by the real value of R. Game $\mathsf{G}_{i+\frac{1}{2}}$ acts according to the nature of the $(i+1)^{th}$ uncompromised next-query: If it is preserving, then it acts as G_{i+1}, if it is recovering, it acts as G_i. Then, one can show that

- If the generator is $(t, q_r, q_n, \gamma^*, \lambda, \varepsilon_p)$-preserving with leakage, then $|\Pr[\mathsf{G}_i = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| \leq \varepsilon_p$. This applies thanks to our security games that make c evolve the same way if the entropy of the input is assumed to be zero, when c is above γ^*;
- If the generator is $(t, q_r, q_n, \gamma^*, \lambda, \varepsilon_r)$-recovering with leakage, then $|\Pr[\mathsf{G}_{i+\frac{1}{2}} = 1] - \Pr[\mathsf{G}_{i+1} = 1]| \leq (q_{ri} + q_{ni} + q_{si}) \cdot \varepsilon_r$, where q_{ri}, q_{ni}, and q_{si} are numbers of refresh, next and state queries between the i^{th} uncompromised next-queries and the $(i+1)^{th}$ uncompromised next-query. Again, this applies since c drops to zero as soon as it decreases below γ^*,and so this cannot happen in a recovering phase.

Let us prove these two results. We first consider **the distance between the games** G_i **and** $\mathsf{G}_{i+\frac{1}{2}}$**.** We assume that we are in a *preserving with leakage* case, otherwise the two games G_i and

$\mathsf{G}_{i+\frac{1}{2}}$ are identical.

We build a reduction to the preserving with leakage security notion. We will thus define an adversary $\mathcal{A}'$ against the preserving with leakage security game. The adversary $\mathcal{A}'$ emulates the challenger for $\mathcal{A}$. It thus runs the adversary $\mathcal{A}$, to get a distribution sampler $\mathcal{D}$ and the leakage functions f. $\mathcal{A}'$ initiates its security game, and receives back the seed it transfers to $\mathcal{A}$. Then $\mathcal{A}'$ sets $S = 0^n$ and $\sigma = 0$, and simulates the answers to all the oracle calls that $\mathcal{A}$ makes in the leakage-resilient robustness security game, but altered as in G_i, until the $(i+1)^{th}$ uncompromised next-query. Note that after the i^{th} uncompromised next-query, the internal state is assumed to be the state S_0 provided by the challenger, since it has full entropy n.

Since $\mathcal{A}'$ knows the leakage functions, controls the internal state of the generator and has access to the distribution sampler $\mathcal{D}$, with the knowledge of its state, it can simulate all the calls to get-state, set-state, $\mathcal{D}$-refresh, leak-refresh, leak-next, and unsafe/compromised next-ror. The uncompromised next-queries are answered with a truly random R (as in G_i) and the internal state is renewed with a truly random S. All these new states are chosen, and thus known to $\mathcal{A}'$, until the i^{th} uncompromised next-queries. After this last query the internal state is not known any more to $\mathcal{A}'$, but is assumed to be the state S_0 provided by the challenger, which is also uniformly random.

Between the i^{th} uncompromised next-queries and the $(i+1)^{th}$ uncompromised next-query, $\mathcal{A}'$ can use its challenger to answer the queries asked by $\mathcal{A}$, but providing the inputs for the refresh queries (with or without leakage), using the distribution sampler $\mathcal{D}$, since it still knows its state σ. Indeed, during the preserving sequence, only oracles $\mathcal{D}$-refresh, leak-refresh, and leak-next are possible. At the end of this sequence, $\mathcal{A}'$ receives the challenge $(S^{(b)}, R^{(b)})$ it uses for the $(i+1)^{th}$ uncompromised next-query: it answers $\mathcal{A}$ with $R^{(b)}$ and updates the internal state of the generator with $S^{(b)}$ and continues to simulate the oracle calls made by $\mathcal{A}$ as in G_i, since it knows again the internal state of the generator and has always known the state of the distribution sampler. Eventually, $\mathcal{A}$ outputs the bit b^*. If the challenge bit b was 0 then $(S^{(b)}, R^{(b)})$ is the real value so it perfectly simulates G_i for $\mathcal{A}$. Otherwise, $(S^{(1)}, R^{(1)})$ is a random value, as in G_{i+1}. Therefore, the challenge of $\mathcal{A}'$ is exactly the same as the challenge consisting in distinguishing both games and we have,

$$|\Pr[\mathsf{G}_i = 1] - \Pr[\mathsf{G}_{i+\frac{1}{2}} = 1]| \leq \varepsilon_p.$$

We now consider **the distance between the games $\mathsf{G}_{i+\frac{1}{2}}$ and G_{i+1}.** We assume that we are in a *recovering with leakage* case, otherwise the two games are identical.

We build a reduction to the recovering with leakage security notion. We will thus define an adversary $\mathcal{A}'$ against the recovering with leakage security game. The adversary $\mathcal{A}'$ emulates the challenger for $\mathcal{A}$. It thus runs the adversary $\mathcal{A}$, to get a distribution sampler $\mathcal{D}$ and the leakage functions f. $\mathcal{A}'$ initiates its security game, and receives back the seed it transfers to $\mathcal{A}$, as well as the values $\gamma_1, \ldots, \gamma_{q_r}$ and $z_1, \ldots, z_{q_r}$. Then $\mathcal{A}'$ sets $S = 0^n$, and simulates the answers to all the oracle calls that $\mathcal{A}$ makes in the leakage-resilient robustness security game, but altered as in G_i, until the $(i+1)^{th}$ uncompromised next-query. Actually, as above, the uncompromised next-queries are answered with truly random R and the internal state is renewed with a truly random S. To simulate the calls to $\mathcal{D}$-refresh and leak-refresh queries from $\mathcal{A}$, $\mathcal{A}'$ asks for a $\mathcal{D}$-refresh query and gets back the input I_k, which allows it to evaluate the refresh algorithm itself, and even compute the leakage information. Together with the values γ_k and z_k it received above from the challenger, it can answer appropriately to $\mathcal{A}$. Since $\mathcal{A}'$ knows the internal state of the generator (and even controls it during the uncompromised next-query), it can easily simulate get-state, set-state, unsafe/compromised next-ror, and leak-next queries.

After the i^{th} uncompromised next-query, it continues the same way until the mRED query, it

has to guess among the possible queries (whose number is bounded by $q_{ri} + q_{ni} + q_{si}$, the sum of the refresh, next and σ queries between the i^{th} uncompromised next-queries and the $(i+1)^{th}$ uncompromised next-query). For this guess (which might later revealed to be incorrect), $\mathcal{A}'$ provides the current internal state, right after the mRED, as S_0 to its challenger. $\mathcal{A}'$ can use its challenger to answer the queries asked by $\mathcal{A}$. Indeed, during the recovering sequence, only oracles $\mathcal{D}$-refresh, leak-refresh, and leak-next are possible (without making c drop to 0). At the end of this sequence, $\mathcal{A}'$ receives the challenge $(S^{(b)}, R^{(b)})$ it uses for the $(i+1)^{th}$ uncompromised next-query, together the sequence of the next inputs: it answers $\mathcal{A}$ with $R^{(b)}$ and updates the internal state of the generator with $S^{(b)}$ and continues to simulate the oracle calls made by $\mathcal{A}$ as in G_i, since it knows again the internal state of the generator and knows the inputs to update it (as in the first part of the simulation). Eventually, $\mathcal{A}$ outputs the bit b^*. If the challenge bit b was 0 then $(S^{(b)}, R^{(b)})$ is the real value so it perfectly simulates G_i for $\mathcal{A}$. Otherwise, $(S^{(1)}, R^{(1)})$ is a random value, as in G_{i+1}. Therefore, the challenge of $\mathcal{A}'$ is exactly the same as the challenge consisting in distinguishing both games and we have,

$$|\Pr[\mathsf{G}_{i+\frac{1}{2}} = 1] - \Pr[\mathsf{G}_{i+1} = 1]| \leq (q_{ri} + q_{ni} + q_{si}) \cdot \varepsilon_r.$$

Combining both results, we have

$$|\Pr[\mathsf{G}_0 = 1] - \Pr[\mathsf{G}_{q_n} = 1]| \leq q_n \cdot (\varepsilon_p + (q_r + q_n + q_s) \cdot \varepsilon_r),$$

while the former game G_0 is the leakage-resilient robustness security game and the latter game G_{q_n} is independent of b. □

6.4 A Secure Construction

We slightly modify the assumption on the standard pseudo-random number generator $\mathbf{G}$, to keep the pseudo-random number generator with input $\mathcal{G}$ secure even in the presence of leakage: The standard pseudo-random number generator $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ instantiated with the truncated product $U = [X' \cdot S]_1^m$ is now required to be a (α, λ)-leakage-resilient and (t, ε)-secure standard pseudo-random number generator according to Definition 40. In that definition, λ denotes the leakage during the execution of $\mathbf{G}$, and α is the expected entropy of the output, even given the leakage.

Definition 40 (Leakage-Resilient and Secure Standard Pseudo-Random Number Generator)**.** *A standard pseudo-random number generator $\mathbf{G} : \{0,1\}^m \to \{0,1\}^N$ is (α, λ)-leakage-resilient and (t, ε)-secure if it is first a (t, ε)-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 λ-bit outputs, there exists a source $\mathcal{S}$ that outputs couples $(L, T) \in \{0,1\}^\lambda \times \{0,1\}^N$, so that the entropy of T, conditioned on L being greater than α, and the advantage with which $\mathcal{A}$ can distinguish $(f(U), \mathbf{G}(U))$ from (L, T) is bounded by ε. Note that $f(U)$ denotes the information leakage generated by f during this execution of $\mathbf{G}$ (on the inputs at the various atomic steps of the computation, that includes U and possibly some internal values).*

Based on our new assumption, Theorem 16 shows that the pseudo-random number generator with input $\mathcal{G}$ is leakage-resilient robust. The proof relies on the notions of *recovering* and *preserving* with leakage.

Theorem 16. *Let m, n, α, and γ^* be integers, such that $n > m$ and $\alpha > \gamma^*$, and $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ an $(\alpha + \ell, \lambda)$-leakage-resilient and $(t, \varepsilon_{\mathbf{G}})$-secure standard pseudo-random number generator. Then, the pseudo-random number generator with input $\mathcal{G}$ previously defined*

and instantiated with $\mathbf{G}$ *is* $(t', q_r, q_n, q_s, \gamma^*, \lambda, \varepsilon)$*-leakage-resilient robust where* $t' \approx t$*, after at most* $q = q_r + q_n + q_s$ *queries, where* q_r *is the number of* $\mathcal{D}$-refresh/leak-refresh*-queries,* q_n *the number of* next-ror/leak-next*-queries, and* q_s *the number of* get-state/set-state*-queries, where* $\varepsilon \leq q q_n \cdot \left(({q_r}^2 + 1) \cdot \varepsilon_{ext} + 3\varepsilon_{\mathbf{G}} \right)$ *and* $\varepsilon_{ext} = \sqrt{2^{m+1-\delta}}$ *for* $\delta = \min\{n - \log q_r, \gamma^* - \lambda\}$.

Following Theorem 15, we show that the pseudo-random number generator with input $\mathcal{G}$ satisfies both the recovering security with leakage and the preserving security with leakage. We also denote ε_{ext} the bias of the distribution of $U = [X' \cdot S]_1^m$ from uniform when the min-entropy of S is greater than $\gamma^* - \lambda$, and show that it can be any value greater than $\sqrt{2^{m+1-\delta}}$ for $\delta = \min\{n - \log q_r, \gamma^* - \lambda\}$. Let us recall that we denote q_r the number of calls to $\mathcal{D}$-refresh/leak-refresh, q_n the number of calls to next-ror/leak-next, and q_s the number of calls to get-state/set-state. We also denote $q = q_r + q_n + q_s$, the global number of queries.

As explained in Section 3.7 under the term *granular* model, we split the algorithm in atomic procedures, with leakage on their manipulated data. In particular, in the above construction, the refresh procedure can be considered atomic, while the next procedure should be split in two: the truncation of the product, and the standard pseudo-random number generator evaluation. As a consequence, we consider three leakage functions:

- f_{refresh} collects the leakage during the computation of the algorithm refresh, and thus takes as inputs the internal state S, the sample I and the part X of the seed;
- $f_{\mathsf{next},\Pi}$ collects the leakage in algorithm next, during the computation of the truncation of the product, and thus takes as inputs the internal state S and the part X' of the seed;
- $f_{\mathsf{next},\mathbf{G}}$ collects the leakage during the standard pseudo-random number generator evaluation. It takes as input the intermediate variable $U = [X' \cdot S]_1^m$.

Lemma 9. *The pseudo-random number generator with input* $\mathcal{G}$ *satisfies the* $(t, q_r, q_n, \gamma^*, \lambda, q_n \cdot ({q_r}^2 \cdot \varepsilon_{ext} + \varepsilon_{\mathbf{G}}))$*-recovering security with leakage.*

Proof of Lemma 9. The proof extends the one built in Section 4.3 to integrate the impact of the leakage.

Game 0 [Recovering with Leakage Security Game].

This game is the original attack game described in Section 6.3, where f is described by three leakage functions f_{refresh}, $f_{\mathsf{next},\Pi}$ and $f_{\mathsf{next},G}$. Because of the restriction for the estimated entropy not to drop to 0, a first sequence includes only $\mathcal{D}$-refresh-queries, until c gets larger than γ^*. Thereafter, the leaking procedures leak-refresh and leak-next are also allowed, in addition to the $\mathcal{D}$-refresh, as soon as c remains above γ^*. With the answer to the challenge next-ror, this game eventually outputs 1 if $b^* = b$, and we want to show that $\Pr[G_0 = 1]$ is close to 1/2.

Game 1.a [First leak-next Query: Random U].

In the first call to leak-next, we replace the truncated product U by a truly random value. Using the same approach as in Section 4.3 with Lemma 4, we can show that the sequence of inputs $(I_k)_{k=1}^d$ generated by the distribution sampler, and the polynomial evaluation followed by the m-truncation leads to a (N, ε)-randomness extractor as long as the entropy in the source $N \geq m + 2\log(1/\varepsilon) + 1$ and $n \geq m + 2\log(1/\varepsilon) + \log(d) + 1$. With the possible information leakage z_k and L_k, the sequence $(I_k)_{k=1}^d$ has a min-entropy larger than $\gamma^* - \lambda$ (because of the possible additional $f_{\mathsf{next},\Pi}(S, X')$), so we just need $m \leq \gamma^* - \lambda - 2\log(1/\varepsilon_{ext}) - 1$ and $m \leq n - 2\log(1/\varepsilon_{ext}) - \log(q_r) - 1$ to guarantee ε_{ext} indistinguishability between the real U and

a random value, with this sequence (I_k).

However, since the adversary can choose when it starts (among q_r possibility), and how long it lasts (again, q_r possibilities), there is a factor loss ${q_r}^2$. Then, we then have $|\Pr[G_0 = 1] - \Pr[G_{1.a} = 1]| \leq {q_r}^2 \cdot \varepsilon_{ext}$.

Game 1.b [First leak-next Query: Random State and Output].

In the first call to leak-next, since $f_{\mathsf{next},G}$ is fixed, and $\mathbf{G}$ is a leakage-resilient standard pseudo-random number generator, the source $\mathcal{S}$ generates indistinguishable leakage, state and random as in the previous game with a truly random U. Then, we then have $|\Pr[G_{1.a} = 1] - \Pr[G_{1.b} = 1]| \leq \varepsilon_{\mathbf{G}}$.

Game 2 [All leak-next Queries: Random States].

In an hybrid way, we replace all the leak-next outputs by $\mathcal{S}$. Then, we have $|\Pr[G_2 = 1] - \Pr[G_0 = 1]| \leq (q_n - 1) \cdot ({q_r}^2 \cdot \varepsilon_{ext} + \varepsilon_{\mathbf{G}})$.

Game 3 [The next-ror Query: Random U].

If this was the first next-query, we have already shown that U can be replaced by a truly random value. If it happens after a leak-next, the state S has enough entropy for the extractor (the global output (S, R) from the source $\mathcal{S}$ had entropy $\alpha + \ell$ when knowing the leakage, then knowing the ℓ-bit randomness R, the remaining entropy for S is above $\alpha \geq \gamma^* \geq \gamma^* - \lambda$): the truncated product in the field is a $2^{-m}(1 + 2^{m-n})$-universal, and thus a (N, ε)-randomness extractor as long as the entropy in the source is $N \geq m + 2\log(1/\varepsilon) + 1$. The above constraint is enough for a bias bounded by ε_{ext} between the real U and a random value. We then have $|\Pr[G_2 = 1] - \Pr[G_3 = 1]| \leq \varepsilon_{ext}$.

Game 4 [The next-ror Query: Random Output].

Since $\mathbf{G}$ is a $(t, \varepsilon_{\mathbf{G}})$-secure standard pseudo-random number generator, we can replace both the output and the random state by truly random values. And we have $|\Pr[G_3 = 1] - \Pr[G_4 = 1]| \leq \varepsilon_{\mathbf{G}}$.

From the above games, one gets, $|\Pr[G_0 = 1] - \Pr[G_4 = 1]| \leq q_n \cdot ({q_r}^2 \cdot \varepsilon_{ext} + \varepsilon_{\mathbf{G}})$, while $\Pr[G_4 = 1] = 1/2$, for any $\varepsilon_{ext} \geq \sqrt{2^{m+1-\delta}}$ for $\delta = \min\{n - \log q_r, \gamma^* - \lambda\}$. □

Remark 2. *This proof with leakage shows the relevance of the adaptation of the generic construction. Concretely, to ensure an internal state with enough entropy at the input of the final* next-ror*, we established two measures to limit the negative impact of the* leak-next *calls. First, the threshold γ^* was set voluntary higher than the original one in the robustness security* ROB*, to capture the leakage in the truncated product, given by the leakage function $f_{\mathsf{next},\Pi}$. Then, the generator $\mathbf{G}$ was defined with security properties whereby, in a* leak-next *call, the final output comes with an entropy at least equal to α despite the leakage.*

Lemma 10. *The pseudo-random number generator with input $\mathcal{G}$ has $(t, q_r, q_n, \gamma^*, \lambda, q_n \cdot (\varepsilon_{ext} + \varepsilon_{\mathbf{G}}) + 2^{-n})$-preserving security with leakage.*

Proof of Lemma 10. The proof extends the one built in Section 4.3 to integrate the impact of the leakage.

Game 0 [Preserving with Leakage Security Game].

This is the original preserving with leakage security game described in Section 6.3. The internal state starts uniformly at random, and then the adversary can ask $\mathcal{D}$-refresh and leak-refresh-queries with chosen inputs, and leak-next-queries before the challenge next-ror, as soon as c remains above γ^*. With the answer to this, this game eventually outputs 1 if $b^* = b$, and we want to show that $\Pr[G_0 = 1]$ is close to 1/2.

Game 1.a [First leak-next Query: Random U].

As above, in the first call to leak-next, we replace the truncated product U by a truly random value. But since the internal state started full of randomness (but it would be true with any entropy level), following $\mathcal{D}$-refresh and leak-refresh-query maintain entropy or reduce it by λ at most, but remaining above γ^*, unless $X = 0$. Then, since the truncated product in the field is a (γ^*, ε)-randomness extractor (with above constraints), the bias is bounded by ε_{ext} between the real U and a random value. Then, we have $|\Pr[G_{1.a} = 1] - \Pr[G_0 = 1]| \leq \varepsilon_{ext} + 2^{-n}$.

Game 1.b [First leak-next Query: Random State and Output].

In the first call to leak-next, since $f_{\mathsf{next},G}$ is fixed, and $\mathbf{G}$ is a leakage-resilient standard pseudo-random number generator, the source $\mathcal{S}$ generates indistinguishable leakage, state and random as in the previous game with a truly random U. We then have $|\Pr[G_{1.a} = 1] - \Pr[G_{1.b} = 1]| \leq \varepsilon_{\mathbf{G}}$.

Game 2 [All leak-next Queries: Random States].

In an hybrid way, we replace all the leak-next outputs by $\mathcal{S}$. Then, we then have $|\Pr[G_2 = 1] - \Pr[G_0 = 1]| \leq (q_n - 1) \cdot (\varepsilon_{ext} + \varepsilon_{\mathbf{G}}) + 2^{-n}$.

Game 3 [The next-ror Query: Random U].

If this was the first next-query, we have already shown that U can be replaced by a truly random value. If it happens after a leak-next, the state S has enough entropy for the extractor: the truncated product in the field is a strong (γ^*, ε)-extractor. Since the state S has an entropy larger than γ^*, the above constraint is enough for a bias bounded by ε_{ext} between the real U and a random value. Then, we have $|\Pr[G_2 = 1] - \Pr[G_3 = 1]| \leq \varepsilon_{ext}$.

Game 4 [The next-ror Query: Random Output].

Since $\mathbf{G}$ is $(t, \varepsilon_{\mathbf{G}})$-secure, we can replace both the output and the random state by truly random values. And we have $|\Pr[G_3 = 1] - \Pr[G_4 = 1]| \leq \varepsilon_{\mathbf{G}}$.

From the above games, one gets, $|\Pr[G_0 = 1] - \Pr[G_4 = 1]| \leq q_n \cdot (\varepsilon_{ext} + \varepsilon_{\mathbf{G}}) + 2^{-n}$, while $\Pr[G_4 = 1] = 1/2$. □

From above Lemmas 9 and 10, we conclude that the generator $\mathcal{G}$ satisfies $(t', q_r, q_n, q_s, \gamma^*, \lambda, \varepsilon)$-leakage-resilient robustness where $t' \approx t$, and

$$\varepsilon = q_n \cdot \Big(\Big(q_n \cdot ({q_r}^2 \cdot \varepsilon_{ext} + \varepsilon_{\mathbf{G}}) \Big) + (q_r + q_n + q_s) \cdot \big(q_n \cdot (\varepsilon_{ext} + \varepsilon_{\mathbf{G}}) + 2^{-n} \big) \Big),$$

which proves Theorem 16, since $q_n \leq q$ and $2^{-n} \leq \varepsilon_{ext}$.

6.5 Instantiations

In Section 6.2, we explained that the original instantiation presented in Section 4.5 is vulnerable to side-channel attacks, and 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 40): 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. In this section, we give three concrete instantiations for such a leakage-resilient and secure standard pseudo-random number generator $\mathbf{G}$. The two first ones are existing constructions proposed and proved leakage-resilient by Faust et al. [FPS12] and Yu and Standaert [YS13]. The third one is a new construction that we propose with a security analysis to improve the security parameters at the expense of the internal state size. Eventually, we implement the three solutions and give benchmarks together with security levels.
To instantiate the standard pseudo-random number generator $\mathbf{G}$, we need a leakage-resilient construction which can get use of a bounded part of the internal state. We recall here two leakage-resilient constructions which can be tweaked to fit these requirements at a reasonable cost. The first one is a binary tree pseudo-random function introduced by Faust, Pietzrak and Schipper at CHES 2012 [FPS12] and the second one is a sequential stateful pseudo-random number generator with minimum public randomness proposed by Yu and Standaert at CT-RSA 2013 [YS13]. We ignore the chronological order and start the description with the second instantiation since a part of it will be used to complete the first one.

Sequential Stateful Pseudo-Random Number Generator from [YS13] The stateful pseudo-random number generator of Yu and Standaert comes with an internal state made of two randomly chosen values : a secret key $K_0 \in \{0,1\}^\mu$ and a public seed $s \in \{0,1\}^\mu$. The construction is made of two stages. In the upper stage, a (non leakage-resilient) generator $\mathbf{F}'$ is processed in counter mode to expand the seed s into uniformly random values $p_0, p_1, \ldots$. In the lower stage, a (non leakage-resilient) pseudo-random function $\mathbf{F}$ generates outputs with public values p_i and updates the secret so it is never used more than twice. The parameter s can be included in our seed (under the notation X'') since it shares the same properties than X and X'. However, the current counter is varying and thus need to be stored in the deterministic part of the internal state. In the proof of [YS13], the counter is implicitly required to be different at each use since the public values p_i need to be independent. But in our model of leakage-resilient robustness, the deterministic part of the internal state can be definitively compromised by the attacker who could, in this case, set the counter to a previous value, making the public p_i not independent anymore. To thwart this issue, we suggest to extend the internal state so that the truncated part of full entropy can contain both the secret key K_0 and a uniformly random counter used only for a single execution of next. This way, no parameter can be compromised and we are back to the context of the proof made by the designers. The only difference in the security comes from the probability of collisions when using a uniformly random counter at each call.

This two-stage instantiation is illustrated in Figure 6.5. One can note that the input U is split in two slices, to initiate the secret key K_0 and the counter C, each of size μ. In order to relate these parameters with the parameters of our generator from Section 6.4 that provides an m-bit random string U as input to the gen $\mathbf{G}$, and wants to receive back an N-bit string, we set $N = n + \ell$: $\kappa = N/\mu$ blocks are generated with κ keys and the κ blocks of output and new internal state are all generated using $2\kappa - 1$ calls to $\mathbf{F}'$ and $2\kappa - 1$ calls to $\mathbf{F}$.
The security of this instantiation is almost entirely guaranteed by its designers in [YS13]. The only difference concerns the uniformly generated counter at each call to function next. Considering the additional possible collisions, Theorem 17 shows how this solution achieves the security

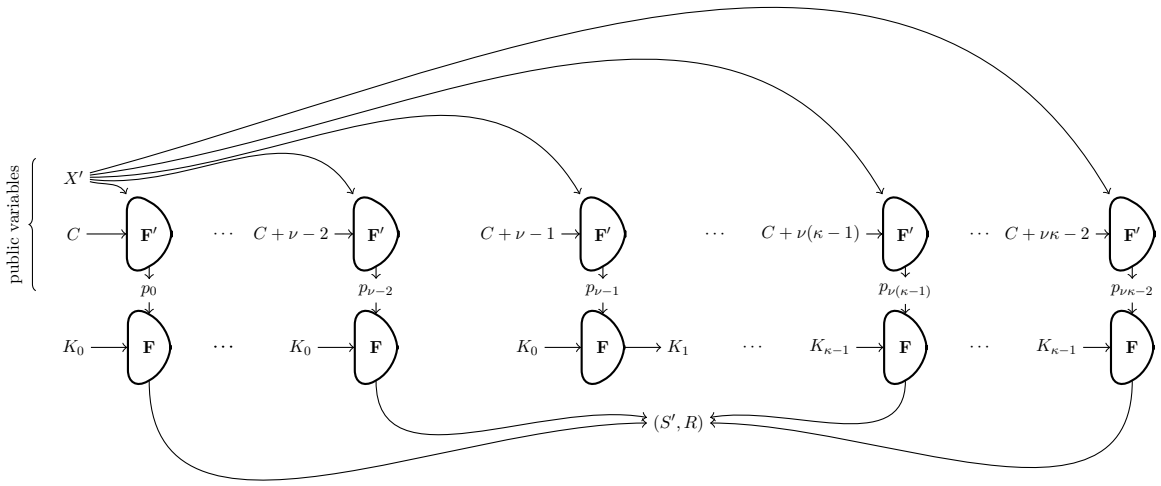

Figure 6.5 – Instantiation of Generator **G** from [YS13] with Random Input $U = (C, K_0)$

requirements in Definition 40.

Theorem 17. *Let μ and κ be parameters such that $(\nu-1)\kappa\mu = N$. Let $\mathbf{F} : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ be a $(\alpha/\kappa, \lambda)$-leakage-resilient and $(t, \nu, \varepsilon_{\mathbf{F}})$-secure pseudo-random function and $\mathbf{F}' : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ be a $(t, q(\nu\kappa-1), \varepsilon_{\mathbf{F}'})$-secure pseudo-random function, where q is a bound on the global number of executions of **G**. The instantiation proposed for **G** as described on Figure 6.5 with **F** and $\mathbf{F}'$ provides an (α, λ)-leakage-resilient and $(t, \varepsilon_{\mathbf{G}})$-secure pseudo-random number generator where $\varepsilon_{\mathbf{G}} \leq \kappa\varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'} + q^2(\nu\kappa-1)/2^\mu$.*

In the proposal, each call to **G** makes $(\nu\kappa - 1)$ calls to the pseudo-random function **F**: κ keys are used at most ν times. The inputs of **F** are generated by $\mathbf{F}'$ with the key X'' (randomly set in seed) on a counter C randomly initialized, and then incremented for each $\mathbf{F}'$ call in an execution of **G**.

The main details of the proof can be found in [YS13], including the upper stage whose validity is guaranteed in the peculiar world minicrypt introduced in [Imp95]. The only differences come from the (possible) multiple use of the same secret key: ν times instead of two and the uniformly distributed counter. They are both integrated in the generator parameters.

Note however that, for the global security, one needs all the intermediate values (p_j^i) to be distinct and unpredictable to avoid the attack described above. We thus require $\mathbf{F}'$ to be secure after $q_n(\nu\kappa - 1)$ queries and the inputs to be all distinct: by setting the $\log(\nu\kappa - 1)$ least significant bits of C to zero, we just have to avoid collisions on the $\mu - \log(\nu\kappa - 1)$ most significant bits for the q_n queries. The probability of collision is thus less than ${q_n}^2(\nu\kappa - 1)/2^\mu$. This probability can appear once and for all in the global security:

Proposition 8. *Let us consider parameters n, m, and ℓ in the construction of the pseudo-random number generator with input $\mathcal{G}$ from Section 6.2, using the standard pseudo-random*

number generator $\mathbf{G}$ *as described on Figure 6.5. Let* μ *and* κ *be parameters such that* $(\nu-1)\kappa\mu = n+\ell$*, and* $\alpha > \gamma^*$*. Let* $\mathbf{F} : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *be a* $(\alpha/\kappa, \lambda)$*-leakage-resilient and* $(t, \nu, \varepsilon_{\mathbf{F}})$*-secure pseudo-random function, and* $\mathbf{F}' : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *be a* $(t, q_n(\nu\kappa-1), \varepsilon_{\mathbf{F}'})$*-secure pseudo-random function. Then,* $\mathcal{G}$ *is* $(t, q_r, q_n, q_s, \gamma^*, \lambda, \varepsilon)$*-leakage-resilient robust after at most* $q = q_r + q_n + q_s$ *queries, where* q_r *is the number of* $\mathcal{D}$-refresh/leak-refresh*-queries,* q_n *the number of* next-ror/leak-next*-queries, and* q_s *the number of* get-state/set-state*-queries, where* $\varepsilon \leq qq_n \cdot \left((q_r{}^2+1)\cdot\sqrt{2^{m+1-\delta}} + 3(\kappa\varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'})\right) + q_n{}^2(\nu\kappa-1)/2^\mu$*, for* $\delta = \min\{n - \log q_r, \gamma^* - \lambda\}$.

It seems reasonable to have (α, λ)-leakage resilience with $\alpha = n + \ell - (\nu\kappa - 1)\lambda$: with a large γ^*, ε can be made small.

Binary Tree Pseudo-Random Function from [FPS12] The second solution was proposed by Faust et al. at CHES 2012 [FPS12]. It requires a few more calls to $\mathbf{F}'$ and $\mathbf{F}$ but depending on the inherent device, some parts can be parallelized to overtake the performances of the first solution. The initial solution does not provide sources for the required randomness. That is why we use the same upper stage as proposed in the first solution. Whereas the leakage-resilient security of this combination is provided in [YS13] and [ABF13], for the same reasons as above, we need to get use of a uniformly random counter, updated at each call to next. Figure 6.6 illustrates this second instantiation with keys K'_i used only for the generation of keys K_i.
The security of this second instantiation with the specific generation of randomness is claimed in [FPS12] and [YS13] for the keys generation and in [ABF13] for the global proposal, but for an incremental counter. The use of a uniformly random counter slightly modifies the security parameters by taking into account the probability of collisions. Theorem 18 shows the conformity with the security requirements of Definition 40.

Theorem 18. *Let* μ *and* κ *be parameters such that* $\nu\kappa\mu = N$*. Let* $\mathbf{F} : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *be a* $(\alpha/\kappa, \lambda)$*-leakage-resilient and* $(t, \nu, \varepsilon_{\mathbf{F}})$*-secure pseudo-random function and* $\mathbf{F}' : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *be a* $(t, q(2\log_2(\kappa) + \nu\kappa), \varepsilon_{\mathbf{F}'})$*-secure pseudo-random function, where* q *is a bound on the global number of executions of* $\mathbf{G}$*. The instantiation proposed for* $\mathbf{G}$ *as described in Figure 6.6 with* $\mathbf{F}$ *and* $\mathbf{F}'$ *provides an* (α, λ)*-leakage-resilient and* $(t, \varepsilon_{\mathbf{G}})$*-secure pseudo-random number generator where* $\varepsilon_{\mathbf{G}} \leq (2\kappa)\varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'} + q^2(\nu\kappa + 2\log_2(\kappa))/2^\mu$.

In the proposal, each call to $\mathbf{G}$ makes $\nu\kappa$ calls to the pseudo-random function $\mathbf{F}$: κ keys are used at most ν times. These keys are the leaves generated by a binary tree whose nodes get use of the outputs of the generator $\mathbf{F}'$ with the key X''. As mentioned in [FPS12], only two uniformly distributed inputs by tree layer are necessary. The generator $\mathbf{F}'$, executed in counter mode as done before, also provides the inputs of $\mathbf{F}$, which raises the total number of required uniformly distributed inputs to $2\log_2(\kappa) + \nu\kappa$.

The main details of the proof can be found in [FPS12] for the key generation and in [ABF13] for the global construction. However and as before, the proof does not consider such a changing counter. Using the same trick as for the previous instantiation, we get the global security:

Proposition 9. *Let us consider parameters* n*,* m*, and* ℓ *in the construction of the pseudo-random number generator with input* $\mathcal{G}$ *from from Section 6.2, using the generator* $\mathbf{G}$ *as described on Figure 6.7. Let* μ *and* κ *be parameters such that* $\nu\kappa\mu = n + \ell$*, and* $\alpha > \gamma^*$*. Let* $\mathbf{F} : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *be a* $(\alpha/\kappa, \lambda)$*-leakage-resilient and* $(t, \nu, \varepsilon_{\mathbf{F}})$*-secure pseudo-random function, and* $\mathbf{F}' : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *be a* $(t, q_n(2\log_2(\kappa) + \nu\kappa)), \varepsilon_{\mathbf{F}'})$*-secure pseudo-random function. Then,* $\mathcal{G}$ *is* $(t, q_r, q_n, q_s, \gamma^*, \lambda, \varepsilon)$*-leakage-resilient robust after at most* $q = q_r + q_n + q_s$ *queries, where* q_r *is the number of* $\mathcal{D}$-refresh/leak-refresh*-queries,* q_n *the number of* next-ror/leak-next*-queries, and* q_s *the number of* get-state/set-state*-queries, where* $\varepsilon \leq qq_n \cdot \left((q_r{}^2+1)\cdot\sqrt{2^{m+1-\delta}} + 3(2\kappa\cdot\varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'})\right) + q_n{}^2(2\log_2(\kappa)+\nu\kappa)/2^\mu$*, for* $\delta = \min\{n - \log q_r, \gamma^* - \lambda\}$.

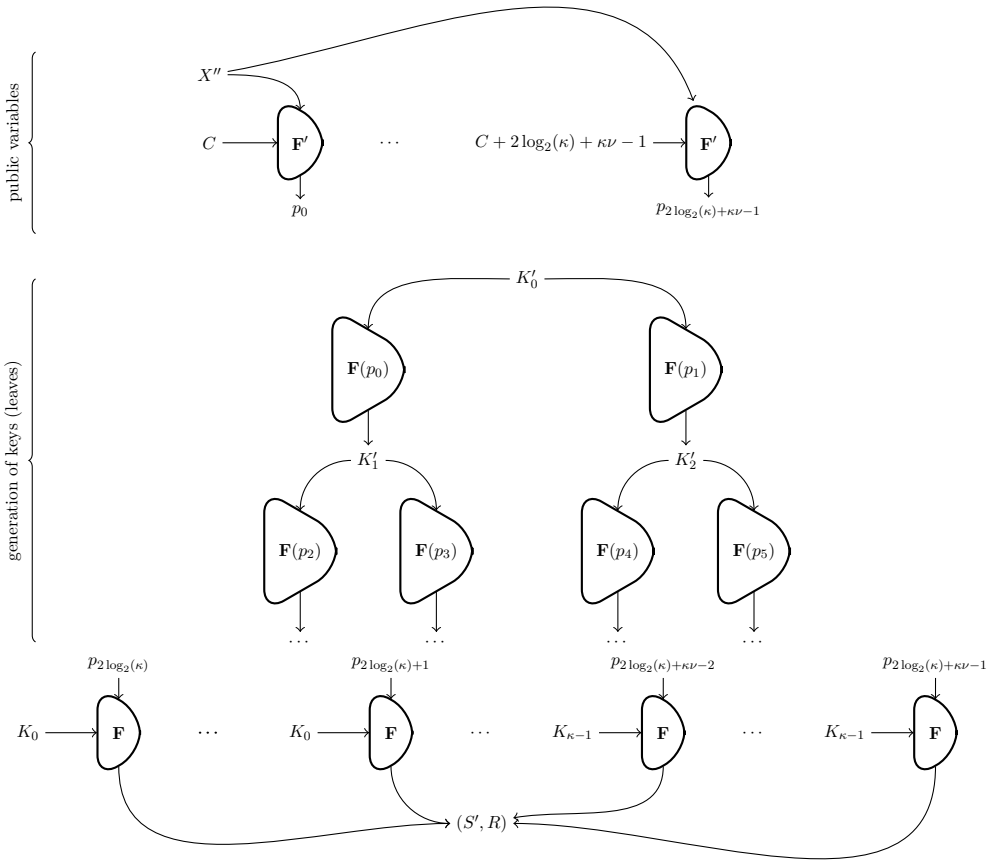

Figure 6.6 – Instantiation of Generator **G** from [FPS12] with Random Input $U = (C, K'_0)$

It seems reasonable to have (α, λ)-leakage resilience with $\alpha = n + \ell - \nu\kappa\lambda$: with a large γ^*, ε can be made small.

New Instantiation As for the existing constructions, since we cannot use a pseudo-random function with different public inputs and a single key (as shown by the first attack in Section 6.2, we follow the conclusions from [BGS15] and make use of a pseudo-random function with a regular re-keying whose frequency depends on the parameters of the inherent device. Fortunately, the number of measurements an attacker can make (which fits with the data complexity) is limited by design. To thwart the second attack described in Section 6.2 and for the needs of the proof, we continue to make use of unpredictable values as inputs of the pseudo-random function. Combining these two approaches, we recover the two stages exhibited by existing constructions. While the upper stage remains the same, we modify the lower stage to improve the security and the performances of function next (as we detail below).

The new lower stage makes several calls to a pseudo-random function $\mathbf{F} : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$, with public but uniformly distributed inputs and several distinct secret key (as in the second existing construction). The latter are directly extracted from the input value $U = [X' \cdot S]_1^m$. This pseudo-random function, denoted by **F**, is just expected to be secure with respect to a very few calls, namely ν, with the same secret key. The precise security requirements

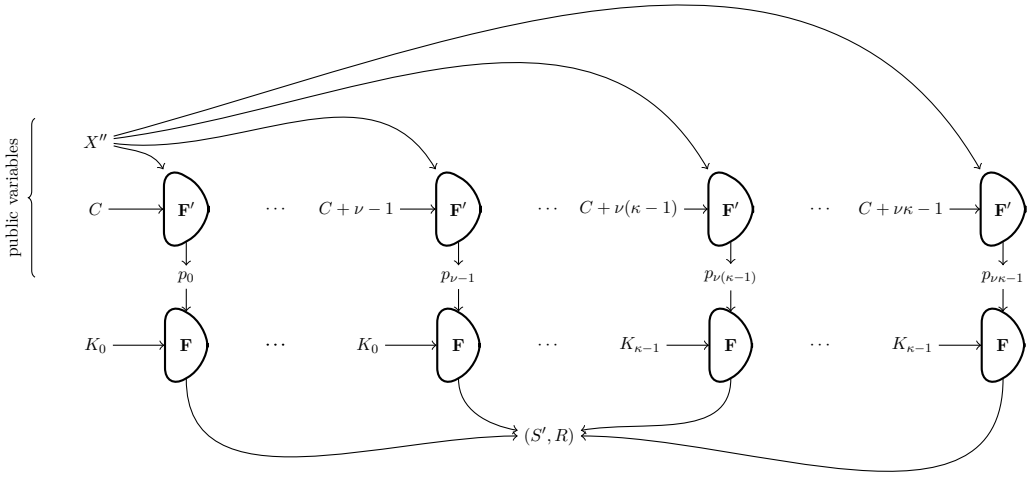

Figure 6.7 – New Instantiation of Generator $\mathbf{G}$ with Random Input $U = (C, K_0, \ldots, K_{\kappa-1})$

are formalized in Definition 41. For the sake of simplicity, we restrict this definition to keyed functions, where keys, inputs, and outputs are all μ-bit long.

Definition 41 (Leakage-Resilient Pseudo-Random Function)**.** *A pseudo-random function* $\mathbf{F} : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *is* (α, λ)*-leakage-resilient and* (t, q, ε)*-secure if it is first a* (t, q, ε)*-pseudo-random function, but in addition, for any adversary* $\mathcal{A}$*, running within time* t*, that first outputs a leakage* f *with* λ*-bit outputs, there exists a source* $\mathcal{S}$ *that outputs* $(L_i, P_i, T_i)_i \in (\{0,1\}^\lambda \times \{0,1\}^\mu \times \{0,1\}^\mu)^q$*, with a uniform distribution for the* P*'s, so that the entropy of* $(T_i)_i$*, conditioned to* $(L_i, P_i)_i$*, is greater than* α*, and the advantage with which* $\mathcal{A}$ *can distinguish the tuple* $(f(K_i, P_i), P_i, \mathbf{F}_K(P_i))_i$ *from* $(L_i, P_i, T_i)_i$ *is bounded by* ε*.*

Of course, when q is large, such a requirement implies security against DPA, but when q is small only SPA is available, which are quite limited attacks in practice. Furthermore, such an assumption is implicitly done in [YS13] with $\alpha = \mu - \lambda$, since the loss of entropy in the output is the leakage one directly gets on it.

This new two-stage instantiation is illustrated in Figure 6.7. One can note that the input U will be split in $\kappa + 1$ slices, to initiate the κ keys $K_0, \ldots, K_{\kappa-1}$, and the counter C, each of size μ. Theorem 19 shows that our proposal achieves the security requirements in Definition 40.

Theorem 19. *Let* μ *and* κ *be parameters such that* $\nu\kappa\mu = N$*. Let* $\mathbf{F} : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *be a* $(\alpha/\kappa, \lambda)$*-leakage-resilient and* $(t, \nu, \varepsilon_{\mathbf{F}})$*-secure pseudo-random function and* $\mathbf{F}' : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ *be a* $(t, q\nu\kappa, \varepsilon_{\mathbf{F}'})$*-secure pseudo-random function, where* q *is a bound on the global number of executions of* $\mathbf{G}$*. The instantiation proposed for* $\mathbf{G}$ *as described in Figure 6.7 with* $\mathbf{F}$ *and* $\mathbf{F}'$ *provides an* (α, λ)*-leakage-resilient and* $(t, \varepsilon_{\mathbf{G}})$*-secure standard pseudo-random number generator where* $\varepsilon_{\mathbf{G}} \leq \kappa \cdot \varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'} + q^2\nu\kappa/2^\mu$*.*

In the proposal, each call to $\mathbf{G}$ makes $\nu\kappa$ calls to the pseudo-random function $\mathbf{F}$: κ keys are used at most ν times. The inputs of $\mathbf{F}$ are generated by $\mathbf{F}'$ with the key X'' (randomly set in

seed) on a counter C randomly initialized, and then incremented for each $\mathbf{F}'$ call in an execution of $\mathbf{G}$.

However, for the global security, one needs all the intermediate values (p_j^i) to be distinct and unpredictable to avoid the attack described above. We thus require $\mathbf{F}'$ to be secure after $q_n\nu\kappa$ queries and the inputs to be all distinct: by setting the $\log(\nu\kappa)$ least significant bits of C to zero, we just have to avoid collisions on the $\mu - \log(\nu\kappa)$ most significant bits for the q_n queries. The probability of collision is thus less than ${q_n}^2\nu\kappa/2^\mu$. This probability can appear once and for all in the global security:

Theorem 20. *Let us consider parameters n, m, and ℓ in the construction of the pseudo-random number generator with input $\mathcal{G}$ from Section 6.2, using the generator $\mathbf{G}$ as described on Figure 6.7. Let μ and κ be parameters such that $\nu\kappa\mu = n + \ell$, and $\alpha > \gamma^*$. Let $\mathbf{F} : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ be a $(\alpha/\kappa, \lambda)$-leakage-resilient and $(t, \nu, \varepsilon_{\mathbf{F}})$-secure pseudo-random function, and $\mathbf{F}' : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\mu$ be a $(t, q_n\nu\kappa, \varepsilon_{\mathbf{F}'})$-secure pseudo-random function. Then, $\mathcal{G}$ is $(t, q_r, q_n, q_s, \gamma^*, \lambda, \varepsilon)$-leakage-resilient robust after at most $q = q_r + q_n + q_s$ queries, where q_r is the number of $\mathcal{D}$-refresh/leak-refresh-queries, q_n the number of next-ror/leak-next-queries, and q_s the number of get-state/set-state-queries, where $\varepsilon \leq qq_n \cdot \left((q_r{}^2 + 1) \cdot \sqrt{2^{m+1-\delta}} + 3(\kappa \cdot \varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'})\right) + {q_n}^2\nu\kappa/2^\mu$, for $\delta = \min\{n - \log q_r, \gamma^* - \lambda\}$.*

It seems reasonable to have (α, λ)-leakage resilience with $\alpha = n + \ell - \nu\kappa\lambda$: with a large γ^*, ε can be made small.

6.6 Benchmarks

We present some benchmarks of the construction presented in Section 4.3 and the three instantiations. Since our leakage-resilient construction is based on the one presented in Section 4.3, we use the latter as a reference when measuring efficiency. Thus, we simply implemented them on an Intel Core i7 processor to show that the new property does not significantly impact the performances. This is mainly due to the use of SPA-resistant AES implementations instead of DPA-resistant (*e.g.*, masked ones). While the target of such constructions is hardware oriented, our benchmarks rely on software implementations, as we focus on estimating the potential efficiency loss of our new construction. We used the same public cryptographic libraries that in Section 4.6 and to achieve a similar security level as the construction presented in Section 4.3, our experiments show that the tweaked binary tree construction is only less than 4 times slower. For our practical analysis, as in Section 4.6 , since it is widely used and adapted to constraint devices, we use AES with 128-bit keys to instantiate the pseudo-random functions.

We recall that our construction is based on the construction of Section 4.3: $\mathsf{refresh}(S, I) = S \cdot X + I \in \mathbb{F}_{2^n}$ and $\mathsf{next}(S) = \mathbf{G}(U)$, with $U = [X' \cdot S]_1^m$. In [DPR+13], the standard pseudo-random number generator $\mathbf{G}$ is defined by $\mathbf{G}(U) = \mathsf{AES}_U(0) \| \ldots \| \mathsf{AES}_U(\nu - 1)$, where ν is the number of calls to AES with a 128-bit key U, and thus $m = 128$. For a security parameter $k = 40$, the security analysis leads to $n = 489$, $\gamma^* = 449$, and $\nu = 5$. To achieve leakage-resilience, we need additional security requirements for the standard pseudo-random number generator $\mathbf{G}$. The three instantiations split $\mathbf{G}$ between two pseudo-random functions $\mathbf{F}$ and $\mathbf{F}'$, where $\mathbf{F}$ is used with public uniformly distributed inputs and κ different secret keys. In the existing constructions, a first key is extracted from the truncated product U and the other ones are derived through a re-keying process. In the new instantiation, all the secret keys are extracted from U. The public inputs of $\mathbf{F}$ are generated by the pseudo-random function $\mathbf{F}'$ in counter mode, with a secret initial value for the counter also extracted from U: $m = 2 \cdot 128$ for the existing constructions or $m = 128(\kappa + 1)$ for the new instantiation if both $\mathbf{F} = \mathbf{F}' = \mathsf{AES}$

Table 6.1 – Security bounds For Robustness against Side-Channel Attacks

Refs	Security Bound ϵ_G	2^{-40} Security			2^{-64} Security		
		n	keys (128)	AES calls	n	keys (256)	AES calls
[YS13]	$\kappa\epsilon_F + \epsilon'_F + q^2(\nu\kappa - 1)/2^\mu$	768	7	26	1152	5	30
[FPS12]	$2\kappa\epsilon_F + \epsilon'_F + q^2(\nu\kappa + 2\log_2(\kappa))/2^\mu$	896	4	20	1408	4	24
New	$\kappa\varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'} + q^2\nu\kappa/2^\mu$	1408	6	24	1792	5	30

with 128-bit keys. To provide the security bounds of the three constructions, we need to fix the security bounds of functions $\mathbf{F}$ and $\mathbf{F}'$. As far as we know, the best key recovery attacks on AES without leakage [BKR11] require a complexity of $2^{126.1}$ with 2^{88} data. However, our functions being executed at most twice (resp. 6 times) with the same secret keys for 2^{-40} security (resp. for 2^{-64} security), such a complexity is unreachable. We use this bound in a conservative way, to bound the security of the pseudo-random functions. As for the leakage, we give the adversary λ bits of useful information by leaking query. Nevertheless, until now it remains unclear how these λ bits of information in a single trace may reduce the security bound of the AES. In [VGS14] for instance, the authors show that a single trace on the AES might give the adversary all the required knowledge to recover the secret key, namely, when a sufficient number of noisy Hamming Weight values are available. But summing the useful information of these noisy Hamming Weight values would give a very large λ for which we cannot guarantee anything. However, we can expect either a larger amount of noise, a desynchronization of the traces or a low leaking from the inherent component which would result in a reasonable value for λ. In this case, we can fix $\varepsilon_{\mathbf{F}} = \varepsilon_{\mathbf{F}'} \approx 2^{-127}$. The resulting security bounds are given in Table 6.1 with the size n of the internal state, the number of 128 or 256-bit keys and the number of AES calls in function next, for 2^{-40} and 2^{-64} security.

The best instantiation in terms of complexity is the construction from [FPS12]. This is not surprising considering the advantageous binary shape of this function. However, if we relax the security assumptions on the AES with $\varepsilon_{\mathbf{F}} = \varepsilon_{\mathbf{F}'} = 2^{-126}$ for security of the pseudo-random functions, the conditions of the security proof are not met and therefore we cannot guarantee its security based on Corollary 20. In these specific cases, our construction seems to be the best one to use since it guarantees that the conditions of the security proof are met. Note that for 2^{-64} security, as explained below, we cannot get a provable security with 128-bits input blocks, and we need $\varepsilon_{\mathbf{F}}$ and $\varepsilon_{\mathbf{F}'}$ to be smaller than 2^{-200}, and then use AES with 256-bit keys.

Since the implementation built from [FPS12] appears to be the best one in the general case, we implement it to compare it with the benchmarks of Section 4.6. As in Section 4.6, we use fb_mul_lodah and fb_add from RELIC open source library [AG], extended with the necessary fields ($\mathbb{F}_{2^{489}}$, defined with $X^{489} + X^{83} + 1$ and $\mathbb{F}_{2^{896}}$, defined with $X^{896} + X^7 + X^5 + X^3 + 1$). We use public functions aes_setkey_enc and aes_crypt_ctr from PolarSSL open source library [Pol]. As in [DPR+13], we measure the number of CPU cycles for a recovering process and a key generation process. The CPU cycles count is done using ASM instruction RDTSC, our C code is optimized with O2 flag. We simulate a full recovery of the generetor for [FPS12] and [DPR+13] implementations, with an input containing one bit of entropy per byte. Then, 8 inputs of size 489 bits are necessary to recover from a compromise for [DPR+13], whereas, for [FPS12], 8 inputs of size 896 bits are necessary. Then we simulate the generation of 2048-bit keys that each requires

16 calls to next, as every call outputs 128 bits. Figure 6.8 gives the numbers of CPU cycles for 100 complete recovering experiments (left) and 100 key generations (right) for [DPR+13] and [FPS12]. Both processes require on average 4 times less CPU cycles to perform for [FPS12] implementation than for [DPR+13] implementation.

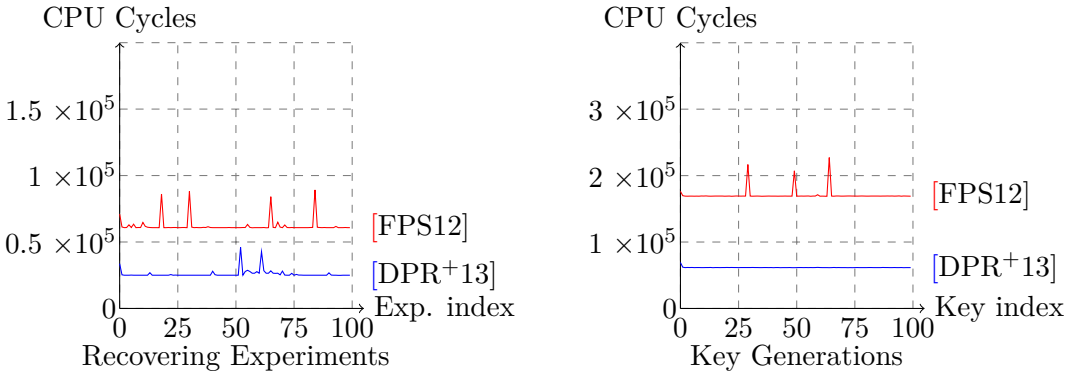

Figure 6.8 – Benchmarks Between [FPS12] and [DPR+13]

The Tweaked Binary Tree Instantiation. We first recall the constraints (Theorem 20): the quality of the pseudo-random number generator is measured by

$$\varepsilon \leq q q_n \cdot \left(({q_r}^2 + 1) \cdot \sqrt{2^{m+1-\delta}} + 3(2\kappa \cdot \varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'}) \right) + {q_n}^2 (2\log_2(\kappa) + \nu\kappa)/2^{\mu},$$

for $\delta = \min\{n - \log q_r, \gamma^* - \lambda\}$.

With $q_r = q_n = q_s = 2^k$, we get:

$$\begin{aligned} \varepsilon &\leq 3 \cdot 2^{2k} \cdot \left((2^{2k} + 1) \cdot \sqrt{2^{m+1-\delta}} + 3(2\kappa \cdot \varepsilon_{\mathbf{F}} + \varepsilon_{\mathbf{F}'}) \right) + (2\log_2(\kappa) + \nu\kappa) \cdot 2^{2k}/2^{\mu} \\ &\leq \varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_4 \end{aligned}$$

where

- $\varepsilon_1 = 2^{4k+2+(m+1-\delta)/2}$,
- $\varepsilon_2 = 18\kappa \cdot 2^{2k} \cdot \varepsilon_{\mathbf{F}}$,
- $\varepsilon_3 = 9 \cdot 2^{2k} \cdot \varepsilon_{\mathbf{F}'}$ and
- $\varepsilon_4 = 2^{2k-\mu} \cdot (2\log_2(\kappa) + \nu\kappa)$.

2^{-v} **Security.** With $m = 256$, $\mu = 128$, $\varepsilon_{\mathbf{F}} = \varepsilon_{\mathbf{F}'} \approx 2^{-127}$: $\varepsilon_1 < 2^{-v}$, as soon as $8k+2v+5+m < \delta$, which is verified for $n > 9k + 2v + 5 + m$ and $\gamma^* > n + \lambda - k$; $\varepsilon_2 < 2^{-v}$, as soon as $2k + v < 127 - \log_2(18\kappa)$; $\varepsilon_3 < 2^{-v}$, as soon as $2k + v < 127 - \log_2(9) < 123$; $\varepsilon_4 < 2^{-v}$, as soon as $2k + v < 128 - \log_2(2\log_2(\kappa) + \nu\kappa)$.

2^{-40} **Security.** For $k = v = 40$, the constraint on ε_3 is satisfied. The constraints on ε_1 are satisfied as soon as $n > 701$ and $\gamma^* > n + \lambda - 40$. With $\nu = 2$, we need $n = 256\kappa - 128 > 701$ and thus $\kappa = 4$, which ensures that the contraints on ε_2 and ε_4 are satisfied . Finally, $n = 896$ and $\gamma^* = 858$ for $\lambda \approx 2$.

2^{-64} **Security.** Unfortunately, for $k = v = 64$, one cannot get a provable security with the size of the input block $\mu = 128$, because of the collisions on the counters. In order to increase the size of the input blocks, one can XOR pseudo-random permutations to get a pseudo-random

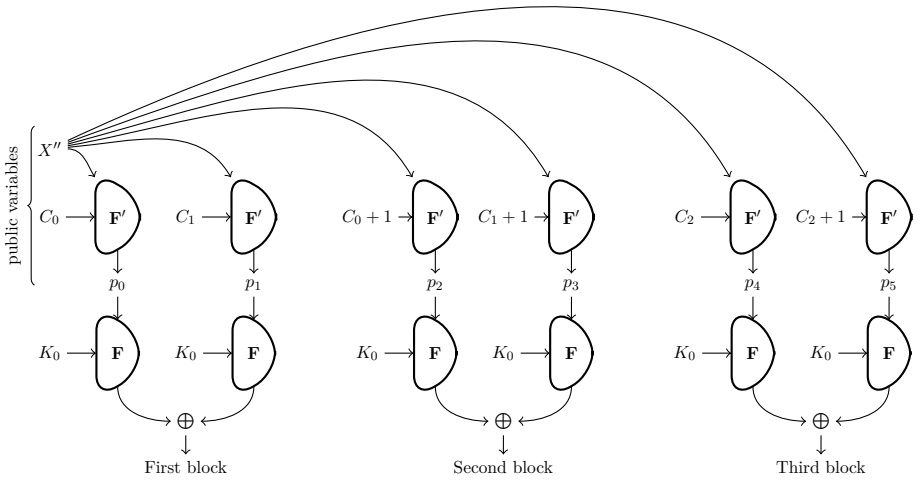

Figure 6.9 – Example of Instantiation of Generator **G** for Higher Security Bounds

function on larger inputs [Luc00]. This makes ε_4 negligible: $2^{2k-2\mu} = 2^{-128}$, and thus the factor $\nu\kappa$ will not affect it. On the other hand, to make ε_2 and ε_3 small enough, we need $\varepsilon_{\mathbf{F}}$ and $\varepsilon_{\mathbf{F}'}$ to be smaller than 2^{-200}, and then use AES with 256-bit keys. But then we have to use the same key 6 times in order to extract 384 bits (see a 3-block extraction in Figure 6.9), where κ keys are used $\nu = 6$ times, and two counters C_0 and C_1 are extracted: $m = 3 \cdot 128 = 384$, $n = 3 \times 128 \times \kappa - 128 = 384\kappa - 128$. As for the constraint on ε_1, we need $384\kappa > 1221$. We can take $\kappa = 4$. Then, $n = 1408$ and $\gamma^* = 1346$.

Chapter 7

Security Analysis

7.1 Introduction

From Security Models to Implementations. We discuss briefly some interesting common points in the security models presented in [DHY02, BH05], in the robustness model (Chapter 4) and in the robustness against memory attacks model (Chapter 5) as well as their potential use to assess the security of the implementations of pseudo-random number generators with input. All security models consider an adversarial environment for the generator. The security model of [DHY02] does not take into account an attack in which the generator is refreshed with adversarial inputs, whereas this situation is considered in [BH05] and in our two models of robustness. In [DHY02], the internal state of the generator is composed of two parts, named *key* and *initial state*; the generation algorithm takes as input both of them and updates the initial state. In concrete implementations the internal state is considered as a single entity, as modelled in [BH05] and in the two robustness models. Finally, entropy accumulation in the internal state is modelled clearly only in the two robustness models. Therefore we use two robustness models as a starting point for our analysis. Furthermore, our source code analysis shows that in certain situations, only a partial compromise of the internal state is necessary to make the generator predictable. As a partial compromise of the internal state is only captured in our model of robustness against memory attacks, we use it to identify precisely the part of the internal state that needs to be compromised to break the security of some generators.

From Implementations To Security Models. We use Definition 27 for pseudo-random number generator with input in all this chapter. This definition describes a pseudo-random number generator as a triple of algorithms $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$, where setup is a probabilistic algorithm that outputs a public parameter seed for the generator. As entropy needs to be extracted from the inputs used to refresh the generator, a *randomness extractor* is needed, ensuring that each input actually gives entropy to the generator. However, it is well known that no deterministic extractor can extract good randomness from all entropy sources and therefore a *seeded extractor* is necessary (see Section 2.6). In all implementations, no explicit extractor is defined; whereas all generators considered in this chapter use the $\mathsf{SHA1}$ hash function to mix new input into the current internal state or to generate outputs. We therefore assume for our analysis that the $\mathsf{SHA1}$ function defines a hash functions family used as an *extractor*, whose *seed* is the public parameter $\mathsf{K} = \mathsf{K}_0||\mathsf{K}_1||\mathsf{K}_2||\mathsf{K}_3$, where $\mathsf{K}_0 = \mathsf{5A827999}$, $\mathsf{K}_1 = \mathsf{6ED9EBA1}$, $\mathsf{K}_2 = \mathsf{8F1BBCDC}$, and $\mathsf{K}_3 = \mathsf{CA62C1D6}$ are the round constants defined in the $\mathsf{SHA1}$ specification [SHA95]. Hence, for all generators presented in this chapter, we assume that the algorithm setup always outputs this public parameter K, of size 128 bits and the underlying extractor is the hash function family defined in the specification [SHA95], indexed by the parameter K. We will therefore refer to the $\mathsf{SHA1}$ function in our description as H_{K}, to identity the underlying hash

function family. As a consequence, this assumption shows that our attacks on these generators are independent of the specific hash function used and are really related to their design.

Consider Algorithms $\mathsf{h_K}$, $\mathsf{H_K}$, PAD and SHA1 described in Table 7.1. These algorithms allow to describe the hash function SHA1 as a particular instance of the hash functions family $\mathsf{H_K}$, with the compression function $\mathsf{h_K}$. Note that this description is similar as the one done in [GB01].

SHA1
Require: M, $|M| < 2^{64}$
1: $\mathsf{K} = \mathsf{5A827999||6ED9EBA1||}$
2: $\quad\mathsf{8F1BBCDC||CA62C1D6}$
3: $V \leftarrow \mathsf{H_K}(M)$
4: **return** V

$\mathsf{H_K}$
Require: M, $|M| < 2^{64}$,
1: $y = \mathsf{PAD}(M)$
2: Parse y as $M_1||M_2||\cdots||M_n$, where $|M_i| = 512$ $(1 \leq i \leq n)$
3: $\mathsf{V} = \mathsf{67452301||EFCDAB89||98BADCFE||}$
4: $\quad\mathsf{10325476||C3D2E1F0}$
5: **for** $i = 1$ to n **do**
6: $\quad\mathsf{V} \leftarrow \mathsf{h_K}(M_i||\mathsf{V})$
7: **end for**
8: **return** V

PAD
Require: M, $|M| < 2^{64}$
1: $d \leftarrow (447 - |M|) \bmod 512$
2: Let ℓ be the 64-bit binary representation of $|M|$
3: $y \leftarrow M||1||0^d||\ell$
4: **return** y

Notations
$X \wedge Y$: bitwise AND of X and Y
$X \vee Y$: bitwise OR of X and Y
$X \oplus Y$: bitwise XOR of X and Y
$\neg X$: bitwise complement of X
$X + Y$: integer sum modulo 2^{32} of X and Y
$\mathsf{ROTL}^{\ell}(X)$: circular left shift of bits of X by ℓ positions $(0 \leq \ell \leq 31)$

$\mathsf{h_K}$
Require: $B||\mathsf{V}$, $|B| = 512, |\mathsf{V}| = 160$
1: Parse B as $W_0||W_1||\cdots||W_{15}$, where $|W_i| = 32$ $(0 \leq i \leq 15)$
2: Parse V as $\mathsf{V}_0||\mathsf{V}_1||\cdots||\mathsf{V}_4$, where $|V_i| = 32$ $(0 \leq i \leq 4)$
3: Parse K as $\mathsf{K}_0||\cdots||\mathsf{K}_3$, where $|\mathsf{K}_i| = 32$ $(0 \leq i \leq 3)$
4: **for** $t = 16$ to 79 **do**
5: $\quad W_t \leftarrow \mathsf{ROTL}^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$
6: **end for**
7: $A \leftarrow \mathsf{V}_0$, $B \leftarrow \mathsf{V}_1$, $C \leftarrow \mathsf{V}_2$, $D \leftarrow \mathsf{V}_3$, $E \leftarrow \mathsf{V}_4$
8: **for** $t = 0$ to 19 **do**
9: $\quad L_t \leftarrow K_0$, $L_{t+20} \leftarrow K_1$, $L_{t+40} \leftarrow K_2$, $L_{t+60} \leftarrow K_3$
10: **end for**
11: **for** $t = 0$ to 79 **do**
12: $\quad$ **if** $0 \leq t \leq 19$ **then** $f \leftarrow (B \wedge C) \vee ((\neg B) \wedge D)$ **end if**
13: $\quad$ **if** $20 \leq t \leq 39$ or $60 \leq t \leq 79$ **then** $f \leftarrow B \oplus C \oplus D$ **end if**
14: $\quad$ **if** $40 \leq t \leq 59$ **then** $f \leftarrow (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$ **end if**
15: $\quad T \leftarrow \mathsf{ROTL}^5(A) + f + E + W_t + L_t$
16: $\quad E \leftarrow D, D \leftarrow C, C \leftarrow \mathsf{ROTL}^{30}(B), B \leftarrow A, A \leftarrow T$
17: **end for**
18: $\mathsf{V}_0 \leftarrow \mathsf{V}_0 + A$, $\mathsf{V}_1 \leftarrow \mathsf{V}_1 + B$, $\mathsf{V}_2 \leftarrow \mathsf{V}_2 + C$, $\mathsf{V}_3 \leftarrow \mathsf{V}_3 + D$, $\mathsf{V}_4 \leftarrow \mathsf{V}_4 + E$
19: $\mathsf{V} \leftarrow \mathsf{V}_0||\mathsf{V}_1||\mathsf{V}_2||\mathsf{V}_3||\mathsf{V}_4$
20: **return** V

Table 7.1 – Algorithms $\mathsf{h_K}$, $\mathsf{H_K}$, PAD and SHA1

In [FPZ08], Fouque *et al.* gave an analysis of the use of the cascade construction as an entropy extractor. The cascade construction is used for iterated hash functions, such as SHA1. In particular, Fouque *et al.* showed, that the cascade mode is a good randomness extractor, if the compression function is a *pseudo-random function*. This result can be used to assess the security of the hash function family $\mathsf{H_K}$ function as a randomness extractor: assuming that the function $\mathsf{h_K}$ (which corresponds to the compression function of the $\mathsf{H_K}$ hash function family) is a *pseudo-random function*, the family $\mathsf{H_K}$ is a randomness extractor. Hence the seed of the extractor is of size 128 bits.

An Illustrative Example Let us illustrate our analysis. All implementations contain instructions that can be easily related to the refresh and next algorithms. However, while our security model considers generators that may be refreshed with potentially biased inputs, in some applications, the refresh algorithm is called just one time with a single input. Hence after this single call, the entropy contained in S (named γ^*) is bounded by the size of the input (named p). An adversary may gain information about the behavior of the environment and estimate the entropy of this single input when collected by the generator. An example of this idea is presented in [MMS13], where it was discovered that the input in the Android `SHA1PRNG` implementation

actually contains very low entropy since it was not generated by several calls to system variables. During our analysis, we discovered vulnerabilities that are complementary to this work, as we focus on the global behavior of the generators.

In our security models, an adversary can compromise the internal state (partially or totally) and the security game ensures that enough entropy is accumulated in the internal state to generate output. The OpenSSL library contains a pseudo-random number generator with an internal state of size 1072 bytes, which contains an entropy pool of size 1023 bytes and internal counters. The structure of S, that we named its *decomposition* in Section 5.1, is public for OpenSSL and known to the adversary. We show that an adversary only needs to compromise 40 bytes of the internal state and to control 23 bytes of an input of size 1023 bytes (with a legitimate distribution sampler, as described in Definition 28) to predict a future output of the generator. Hence, this shows that this pseudo-random number generator with input does not resist a single relatively small internal state compromise.

7.2 Security of Linux Generators

In Unix-like operating systems, a pseudo-random number generator with input was implemented for the first time for Linux 1.3.30 in 1994. The entropy source comes from device drivers and other sources such as latencies between user-triggered events (keystroke, disk I/O, mouse clicks, ...). It is gathered into an internal state called the *entropy pool*. The internal state keeps an estimate of the number of bits of entropy in the internal state and (pseudo-)random bits are created from the special files `/dev/random` and `/dev/urandom`. Barak and Halevi [BH05] discussed briefly the generator `/dev/random` but its conformity with their robustness security definition is not formally analyzed.

The first security analysis of these generators was given in 2006 by Gutterman, Pinkas and Reinman [GPR06]. It was completed in 2012 by Lacharme, Röck, Strubel and Videau [LRSV12]. Gutterman *et al.* [GPR06] presented an attack based on kernel version 2.6.10 for which a fix has been published in the following versions. Lacharme *et al.* [LRSV12] gives a detailed description of the operations of the generators and provides a result on the entropy preservation property of the mixing function used to refresh the internal state.

The Linux operating system contains one pseudo-random number generator with input, that has two user interfaces, named `/dev/random` and `/dev/urandom`. They are part of the kernel and used in the OS security services and some cryptographic libraries. We give a precise description[1] of this pseudo-random number generator with input in accordance with Definition 27 as a triple $\mathsf{LINUX} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ and we prove the following theorem:

Theorem 21. *The pseudo-random number generator with input* LINUX *is not robust.*

Since the actual generator LINUX does not define any seed (*i.e.* the algorithm setup always output the empty string), as mentioned above, it cannot achieve the notion of robustness. However, in the following, we additionally mount concrete attacks that would work even if LINUX had used a seed in the underlying hash function or mixing function. The attacks exploit two independent weaknesses, in the entropy estimator and the mixing functions, which would need both to be fixed in order to expect the generators to be robust.

[1] All descriptions were done by source code analysis. We refer to version 3.7.8 of the Linux kernel.

General Overview

Security Parameters. The generator LINUX uses parameters $n = 6144$, $\ell = 80$, $p = 96$. The parameter n can be modified (but requires kernel compilation), and the parameters ℓ (size of the output) and p (size of the input) are fixed. The generator outputs the requested random numbers by blocks of $\ell = 80$ bits and truncates the last block if needed.

Internal State. The internal state of generator LINUX is a triple $S = (S_i, S_u, S_r)$ where $|S_i| = 4096$ bits, $|S_u| = 1024$ bits and $|S_r| = 1024$ bits. New data is collected in S_i, which is named the *input pool*. Output is generated from S_u and S_r which are named the *output pools*. When a call to `/dev/urandom` is made, data is generated from the pool S_u and when a call to `/dev/random` is made, data is generated from the pool S_r.

Functions refresh and next. There are two refresh functions, $\mathsf{refresh}_i$ that initializes the internal state and $\mathsf{refresh}_c$ that updates it continuously. There are two next functions, next_u for `/dev/urandom` and next_r for `/dev/random`.

Mixing Function. The generator uses a *mixing function* M, described below, to mix new data in the input pool and to transfer data between the pools.

Entropy Estimator. The generator uses an *entropy estimator*, described below, to estimate the entropy of the collected input and to continuously estimates the entropy of the pools. With these estimations, the generator controls the transfers between the pools and how new input is collected. This is illustrated in Figure 7.1 and described below but at high level, the main principles are:

- New inputs are ignored when the input pool contains enough entropy. Otherwise, the estimated entropy of the input pool is increased with new input.
- Entropy estimation of the output pool is decreased on generation.
- Data is transferred from the input pool to the output pools if they require entropy.
- When the pools do not contain enough entropy, no output can be generated with `/dev/random` and it blocks whereas `/dev/urandom` always generates output.

The technical internal parameters that are the entropy estimations are named E_i (entropy estimator of S_i), E_u (of S_u), E_r (of S_r).

The $\mathsf{refresh}_i$ and $\mathsf{refresh}_c$ Functions

The generator LINUX contains two refresh functions. A first refresh function, $\mathsf{refresh}_i$, is used to generate the first internal state of the generator and the second one, $\mathsf{refresh}_c$, is used to refresh continuously the generator with new input.

Internal State Initialisation with $\mathsf{refresh}_i$. To generate the first internal state with $\mathsf{refresh}_i$, LINUX collects device-specific data using a built-in function called add_device_randomness and refreshes S_i and S_n with them. The data is derived from system calls, a call to variable jiffies, which gives the number of CPU cycles since system start-up and is represented by 32 bits, and a call to system function get_cycles, that gives the number of clock ticks since system start-up, which also returns 32 bits. The two values are xor-ed together, giving a new 32-bit input data that is generated twice for S_i and S_n and mixed for each pool. Then LINUX collects system data and refreshes the three pools S_i, S_n and S_b with them using built-in function init_std_data. The data is derived from system calls, a call to function ktime_get_real, which returns 64 bits and a call

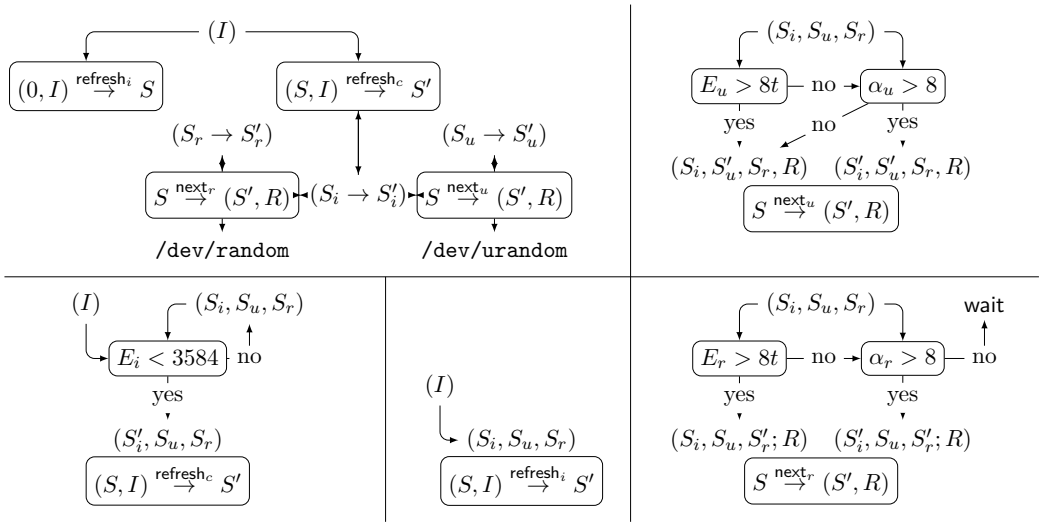

Figure 7.1 – Relations between functions and pools for LINUX

to function utsname, which returns 3120 bits. The two are concatenated, giving 3184 bits. This input data is generated for each pool and mixed with M, implemented in the built-in function mix_pool_bytes. Finally, the generated input is $I = (\text{utsname}\|\text{ktime\_get\_real}\|\text{get\_cycles} \oplus \text{jiffies})$ for S_i and S_n, and $I = (\text{utsname}\|\text{ktime\_get\_real})$ for S_r. In all cases, $\text{refresh}_i(0, I) = \mathsf{M}(0, I)$. The entropy estimator is not used during this process, so $E_i = E_u = E_r = 0$.

Algorithm 6 LINUX refresh_i

Require: $I_1 = [\text{utsname}\|\text{ktime\_get\_real}\|\text{get\_cycles} \oplus \text{jiffies}]$, $I_2 = [\text{utsname}\|\text{ktime\_get\_real}]$, $S = \varnothing$
Ensure: $S = (S_i, S_u, S_r)$
1: $S_i = \mathsf{M}(I_1, 0)$
2: $S_r = \mathsf{M}(I_2, 0)$
3: $S_u = 0$
4: **return** $S = (S_i, S_u, S_r)$

Internal State Update with refresh_c. The refresh_c function uses system events that are collected by three built-in functions: add_input_randomness, add _interrupt_randomness and add_disk_randomness. All of them call another built-in function, add_timer _randomness, which builds a 96 bits input data containing the collected event mapped to a specific value num coded in 32 bits, concatenated with jiffies and get_cycles. Finally, the generated input is then given by $I = [\text{jiffies}\|\text{get\_cycles}\|\text{num}]$. If the estimated entropy is above the default value 3584, this input is ignored (except 1 input over 4096). The entropy estimator Ent described below is used to estimate the entropy of the new input and is added to E_i.

Algorithm 7 LINUX refresh_c

Require: $I = [\text{jiffies}\|\text{get\_cycles}\|\text{num}]$, $S = (S_i, S_u, S_r)$
Ensure: $S' = (S'_i, S'_u, S'_r)$
1: **if** $E_i \geq 3584$ **then**
2: $\quad S'_i = S_i$
3: **else**
4: $\quad e = \mathsf{Ent}(I)$
5: $\quad S'_i = \mathsf{M}(I, S_i)$
6: $\quad E_i = e + E_i$
7: **end if**
8: $(S'_u, S'_r) = (S_u, S_r)$
9: **return** $S' = (S'_i, S'_u, S'_r)$

Remark 3. *Starting from version* 3.6.0 *of the kernel,* LINUX *involves a particular behavior of* add_interrupt _randomness *which collects system events and gather them in a dedicated 128 bits pool* fast_pool *without calling* add_timer _randomness*. In this case, the input is* $I =$ fast_pool*.*

For all these inputs, $\mathsf{refresh}_c(S_i, I) = \mathsf{M}(S_i, I)$ and LINUX estimates the entropy of the data collected by add_timer _randomness and estimates every input collected from fast_pool to 1 bit.

Remark 4. *Starting from version 3.2.0 of the kernel, for both* `/dev/urandom` *and* `/dev/random`*, there is an additional input specific for* `x86` *architectures for which a hardware random number generator is available. In this case, the output of the generator is mixed with* M *when this hardware random number generator is used for* $\mathsf{refresh}_i$ *and the output is mixed with the output of* LINUX *when used with* next*. For this specific architecture, denoting* I_{hd} *the input generated by the hardware random number generator,* $\mathsf{refresh}_i(S_i, I_{hd}) = \mathsf{M}(S_i, I_{hd})$ *and* $\mathsf{next}_{hd}(S) = [I_{hd}||\mathsf{next}(S)]$.

The next_u and next_r Functions

The next functions use built-in functions random_read and urandom_read that are user interfaces to read data from `/dev/random` and `/dev/urandom`, respectively. A third kernel interface, get_random_bytes(), allows to read from `/dev/urandom`. The three rely on the same built-in function extract_buf that calls the mixing function M, the hash function $\mathsf{H_K}$ described in Section 7.1 and a folding function $\mathsf{F}(w_0, \cdots, w_4) = (w_0 \oplus w_3, w_1 \oplus w_4, w_{2_{[0\cdots15]}} \oplus w_{2_{[16\cdots31]}})$.

Algorithm 8 LINUX next_r

Require: t, $S = (S_i, S_u, S_r)$
Ensure: R, $S' = (S'_i, S'_u, S'_r)$
1: $\alpha_r = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor)$
2: **if** $\alpha_r \geq 8$ **then**
3: $\quad T_i = \mathsf{F} \circ \mathsf{H_K} \circ \mathsf{M}(S_i, \mathsf{H_K}(S_i))$
4: $\quad S'_i = \mathsf{M}(S_i, \mathsf{H_K}(S_i))$
5: $\quad S^*_r = \mathsf{M}(S_r, T_i)$
6: $\quad E_i = E_i - 8\alpha_r$
7: $\quad E_r = E_r + 8\alpha_r$
8: $\quad S'_r = \mathsf{M}(S^*_r, \mathsf{H_K}(S^*_r))$
9: $\quad R = \mathsf{F} \circ \mathsf{H_K} \circ \mathsf{M}(S^*_r, \mathsf{H_K}(S^*_r))$
10: $\quad E_r = E_r - 8t$
11: **else**
12: $\quad$ Blocks until $\alpha_r \geq 8$
13: **end if**
14: $S'_u = S_u$
15: **return** R, $S' = (S'_i, S'_u, S'_r)$

Output with `/dev/random`. Let us describe the transfers when t bytes are requested from the blocking pool. If $E_r \geq 8t$, then the output is generated directly from S_r: LINUX first calculates a hash across S_r, then mixes this hash back with S_r, hashes again the output of the mixing function and folds the result in half, giving $R = \mathsf{F} \circ \mathsf{H_K} \circ \mathsf{M}(S_r, \mathsf{H_K}(S_r))$ and $S'_r = \mathsf{M}(S_r, \mathsf{H_K}(S_r))$. This decreases E_r by $8t$ and the new value is $E_r - 8t$. If $E_r < 8t$, then depending on E_i, data is transferred from S_i to S_r. Let $\alpha_r = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor)$.

- If $\alpha_r \geq 8$, then α_r bytes are transferred between S_i and S_r (so at least 8 bytes and at most 128 bytes are transferred between S_i and S_r, and S_i can contain 0 entropy. The transfer is made in two steps: first LINUX generates from S_i an intermediate data $T_i = \mathsf{F} \circ \mathsf{H_K} \circ \mathsf{M}(S_i, \mathsf{H_K}(S_i))$ and then it mixes it with S_r, giving the intermediate states $S'_i = \mathsf{M}(S_i, \mathsf{H_K}(S_i))$ and $S^*_r = \mathsf{M}(S_r, T_i)$. This decreases E_i by $8\alpha_r$ and increases E_r

by $8\alpha_r$. Finally LINUX outputs t bytes from S_r^*, this produces the final output pool $S_r' = \mathsf{M}(S_r^*, \mathsf{H}_\mathsf{K}(S_r^*))$ and $R = \mathsf{F} \circ \mathsf{H}_\mathsf{K} \circ \mathsf{M}(S_r^*, \mathsf{H}_\mathsf{K}(S_r^*))$. This decreases E_r by $8t$.

- If $\alpha_r < 8$, then LINUX blocks and waits until S_i gets refreshed with I and until $\alpha_r \geq 8$.

Output with `/dev/urandom`. Similarly, let us describe the transfers when t bytes are requested from the non-blocking pool. If $E_u \geq 8t$ then LINUX applies the same process as in the non-blocking case, outputs $R = \mathsf{F} \circ \mathsf{H}_\mathsf{K} \circ \mathsf{M}(S_u, \mathsf{H}_\mathsf{K}(S_u)))$ and sets $S_u' = \mathsf{M}(S_u, \mathsf{H}_\mathsf{K}(S_u))$. If $E_u < 8t$ then LINUX behaves differently. Let $\alpha_u = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor - 16)$:

- If $\alpha_u \geq 8$, the process is the same as in the non-blocking case, but with S_u, E_u and α_u instead of S_r, E_r and α_r.

- If $\alpha_u < 8$, then LINUX outputs the requested bytes from S_u without transferring data from S_i. Hence LINUX behaves as if $E_u \geq 8t$: $R = \mathsf{F} \circ \mathsf{H}_\mathsf{K} \circ \mathsf{M}(S_u, \mathsf{H}_\mathsf{K}(S_u))$, and $S_u' = \mathsf{M}(S_u, \mathsf{H}_\mathsf{K}(S_u))$. This decreases E_u by $8t$ and the new value is 0.

Algorithm 9 LINUX next_u

Require: t, $S = (S_i, S_u, S_r)$
Ensure: R, $S' = (S_i', S_u', S_r')$
1: $\alpha_u = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor - 16)$
2: **if** $\alpha_u \geq 8$ **then**
3: $\quad T_i = \mathsf{F} \circ \mathsf{H}_\mathsf{K} \circ \mathsf{M}(S_i, \mathsf{H}_\mathsf{K}(S_i))$
4: $\quad S_i' = \mathsf{M}(S_i, \mathsf{H}_\mathsf{K}(S_i))$
5: $\quad S_u^* = \mathsf{M}(S_u, T_i)$
6: $\quad E_i = E_i - 8\alpha_u$
7: $\quad E_u = E_u + 8\alpha_u$
8: $\quad S_u' = \mathsf{M}(S_u^*, \mathsf{H}_\mathsf{K}(S_u^*))$
9: $\quad R = \mathsf{F} \circ \mathsf{H}_\mathsf{K} \circ \mathsf{M}(S_u^*, \mathsf{H}_\mathsf{K}(S_u^*))$
10: $\quad E_u = E_u - 8t$
11: **else**
12: $\quad R = \mathsf{F} \circ \mathsf{H}_\mathsf{K} \circ \mathsf{M}(S_u, \mathsf{H}_\mathsf{K}(S_u))$
13: $\quad E_u = 0$
14: **end if**
15: $S_r' = S_r$
16: **return** R, $S' = (S_i', S_u', S_r')$

This illustrates the difference between `/dev/urandom` and `/dev/random`: If the estimated entropy of the blocking pool S_r is less than $8t$ and no transfer is done, then `/dev/random` blocks, whereas `/dev/urandom` does never block and outputs the requested t bytes from the non-blocking pool S_u.

The Entropy Estimator

A built-in estimator Ent is used to give an estimation of the entropy of the input data used to refresh S_i. It is implemented in function add_timer_randomness which is used to refresh the input pool. A timing t_n is associated with each event (system or user call) that is used to refresh the internal state. Entropy is estimated when new input data is used to refresh the internal state, entropy is not estimated using input distribution but only using the timings of the data. A description of the estimator is given in [GPR06], [LRSV12] and [GLSV12]. The estimator takes as input a sequence of inputs $I_i = [\mathsf{jiffies} \| \mathsf{get\_cycles} \| \mathsf{num}]$, it calculates differences between timings of events, where t_0, t_1, t_2, ... are the jiffies associated with each input: $\delta_i = t_i - t_{i-1}$, $\delta_i^2 = \delta_i - \delta_{i-1}$, $\delta_i^3 = \delta_i^2 - \delta_{i-1}^2$. Then, it calculates $\Delta_i = \min(|\delta_i|, |\delta_i^2|, |\delta_i^3|)$ and finally applies a logarithmic function to give the estimated entropy $H_i = 0$ if $\Delta_i < 2$, $H_i = 11$ if $\Delta_i > 2^{12}$, and $H_i = \lfloor \log_2(\Delta_i) \rfloor$ otherwise.

Algorithm 10 LINUX Entropy Estimator

Require: $I_i = [\mathsf{num}\|\mathsf{jiffies}\|\mathsf{get\_cycles}]$
Ensure: $H_i = \mathsf{Ent}(I_i)$
1: $t_i = \mathsf{jiffies}$
2: $\delta_i = t_i - t_{i-1}$
3: $\delta_i^2 = \delta_i - \delta_{i-1}$
4: $\delta_i^3 = \delta_i^2 - \delta_{i-1}^2$
5: $\Delta_i = \min(|\delta_i|, |\delta_i^2|, |\delta_i^3|)$
6: **if** $\Delta_i < 2$ **then** $H_i = 0$
7: **if** $\Delta_i > 2^{12}$ **then** $H_i = 11$
8: **else** $H_i = \lfloor \log_2(\Delta_i) \rfloor$
9: **return** $H_i = \mathsf{Ent}(I_i)$

The Folding and the Hash Functions

The folding function F and the hash function $\mathsf{H_K}$ are used when random bytes are generated by LINUX and when data is transferred from S_i to S_r or S_u. The folding function is implemented in built-in function extract_buf. It take as input five 32-bit words and output 80 bits of data. This function F is defined by $\mathsf{F}(w_0, w_1, w_2, w_3, w_4) = (w_0 \oplus w_3, w_1 \oplus w_4, w_{2_{[0\cdots15]}} \oplus w_{2_{[16\cdots31]}})$, where w_i for $i \in \{0, \ldots, 4\}$ are the input words.

The hash function $\mathsf{H_K}$ is implemented in the built-in function extract_buf by a call to a Linux system function sha_transform.

The Mixing Function

The Mixing function M is the core of generator LINUX. It is implemented in the built-in function mix_pool_bytes. It is used in two contexts, once to refresh the internal state with new input and secondly to transfer data between the input pool and the output pools. We give a complete description of M as it is used to refresh the input pool S_i, its description when it is used to transfer data between pools differs only from internal parameters.

The function M takes as input I of size one byte, the input pool S_i that is considered as a table of 128 32-bits words. It selects 7 words in S_i and mixes them with I and replaces one word of S_i with the result. The pool S_i therefore maintains an internal parameter, named k, which is used to select the word that will be modified. Another internal parameter, named d, is used in function M. This parameter is a multiple of 7 used in a rotation done at word level. We name the rotation of d bits R_d. The mixing function involves the following operations:

- The byte containing the entropy source is converted into a 32-bit word, using standard C implicit cast, and rotated by d bits. Before initialization, $d = 0$, and each time the mixing function M is used, d is incremented using k : if $k = 0 \bmod 128$ then $d = d + 14 \bmod 32$ and $d = d + 7 \bmod 32$ otherwise.

- The obtained word is xor-ed with words from the pool. If we note $S_0, \ldots, S_{127}$ the words of S_i, chosen words will be $S_{k+j \bmod 128}$ for $j \in \{0, 1, 25, 51, 76, 103\}$[2].

- The obtained word is mixed with a built-in table (named *twist table*). This table contains the binary representations of the monomials $\{0, \alpha^{32*j}\}, j = 1, \ldots 7$, in the field $(\mathbb{F}_2)/(Q)$, where $Q(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$ is the CRC32 polynomial used for Ethernet protocol [Koo02]. Denoting the primitive element α, this operation can be described as $W \to W.\alpha^3 + \mathsf{R}(\mathsf{Q}(W, \alpha^{29}).\alpha^{32}, Q)$, where $\mathsf{Q}(A, B)$ (resp. $\mathsf{R}(A, B)$) the quotient (resp. the remainder) in polynomial division A/B.

[2]Similarly, the words chosen from S_r and S_u will be $S_{k+j \bmod 32}$ for $j \in \{0, 1, 7, 14, 20, 26\}$.

- Then the word at index k in S_i is replaced by the previously generated word and k is incremented.

Algorithm 11 LINUX Mixing function

Require: I, $S = (S_0, \ldots, S_k, \ldots, S_{127})$
Ensure: S'
1: $W = \mathsf{R}_d[0||I]$
2: **if** $k = 0 \bmod 128$ **then** $d = d + 14 \bmod 32$ **else** $d = d + 7 \bmod 32$ **end if**
3: $W = W \oplus S_{k+j \bmod 128}$, $j \in \{0, 1, 25, 51, 76, 103\}$
4: $W = W.\alpha^3 + \mathsf{R}(\mathsf{Q}(W, \alpha^{29}).\alpha^{32}, Q)$
5: $S'_k = W$
6: $k = k + 1$
7: **return** $S = (S_0, \ldots, S'_k, \ldots, S_{127})$

Distributions Used for Attacks

Distributions Used in Attacks based on the Entropy Estimator As shown previously, the generator LINUX uses an internal *Entropy Estimator* on each input that continuously refreshes its internal state. We show that this estimator can be fooled in two ways. First, it is possible to define a distribution of zero entropy that the estimator will estimate of high entropy, secondly, it is possible to define a distribution of arbitrary high entropy that the estimator will estimate of zero entropy. This is due to the estimator conception: as it considers the timings of the events to estimate their entropy, regular events (but with unpredictable data) will be estimated with zero entropy, whereas irregular events (but with predictable data) will be estimated with high entropy. These two distributions are given in the following Lemma 11 and 12.

Lemma 11. *There exists a stateful distribution $\mathcal{D}_0$ such that $\mathbf{H}_\infty(\mathcal{D}_0) = 0$, whose estimated entropy by* LINUX *is high.*

Proof. Let us define the 32-bits word distribution $\mathcal{D}_0$. On input a state i, $\mathcal{D}_0$ updates its state to $i + 1$ and outputs a triple $(i + 1, [W_1^i, W_2^i, W_3^i]) \xleftarrow{\$} \mathcal{D}_0(i)$, where $W_1^0 = 2^{12}, W_1^i = ||\lfloor \cos(i).2^{20} \rfloor|| + W_1^{i-1}$, $W_2^i = W_3^i = 0$. For each state, $\mathcal{D}_0$ outputs a 12-bytes input containing 0 bit of random data, we have $\mathbf{H}_\infty(\mathcal{D}_0) = 0$ conditioned on the previous and the future outputs (*i.e.* $\mathcal{D}_0$ is legitimate only with $\gamma_i = 0$ for all i). Then $\Delta_i > 2^{12}$ and $H_i = 11$. □

Lemma 12. *There exists a stateful distribution $\mathcal{D}_1$ such that $\mathbf{H}_\infty(\mathcal{D}_1) = 64$, whose estimated entropy by* LINUX *is null.*

Proof. Let us define the 32-bits word distribution $\mathcal{D}_1$. On input a state i, $\mathcal{D}_1$ updates its state to $i+1$ and outputs a triple: $(i+1, [W_1^i, W_2^i, W_3^i]) \xleftarrow{\$} \mathcal{D}_1(i)$, where $W_i = i, W_2 \xleftarrow{\$} \mathcal{U}_{32}$ and $W_3 \xleftarrow{\$} \mathcal{U}_{32}$. For each state, $\mathcal{D}_1$ outputs a 12-bytes input containing 8 bytes of random data, we have $\mathbf{H}_\infty(\mathcal{D}_1) = 64$ conditioned on the previous and the future outputs (*i.e.* $\mathcal{D}_1$ is legitimate with $\gamma_i = 64$ for all i). Then $\delta_i = 1$, $\delta_i^2 = 0$, $\delta_{i-1}^2 = 0$, $\delta_i^3 = 0$, $\Delta_i = 0$ and $H_i = 0$. □

Distribution Used in Attack based on the Mixing Function The generator LINUX uses an internal Mixing function M, used to refresh the internal state with new input and to transfer data between the pools. It is possible to define a distribution of arbitrary high entropy for which the Mixing function is completely counter productive, i.e. the entropy of the internal state does not increase, whatever the size of the input is. This is due to the conception of the Mixing function and its linear structure. This distribution is given in Lemma 13.

Lemma 13. *There exists a stateful distribution $\mathcal{D}_2$ such that $\mathbf{H}_\infty(\mathcal{D}_2) = 1$, for which $\mathbf{H}_\infty(S) = 1$ after t* refresh*, for arbitrary high t.*

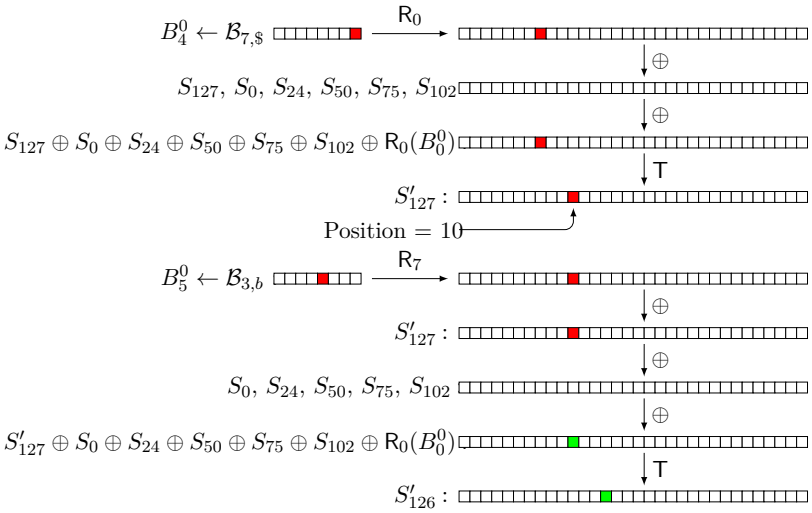

Figure 7.2 – Attack Against the Mixing Function of LINUX

Proof. Let us define the byte distributions $\mathcal{B}_{i,b}$ and $\mathcal{B}_{i,\$}$:

$$\mathcal{B}_{i,b} = \{(0, \cdots, b, \cdots, 0), b_i \leftarrow b, b_j = 0 \text{ if } i \neq j\}$$
$$\mathcal{B}_{i,\$} = \{(b_0, \cdots, b_7), b_i \xleftarrow{\$} \{0,1\}, b_j = 0 \text{ if } i \neq j\}$$

Let us define the 12 bytes distribution $\mathcal{D}_2$. On input a state i, $\mathcal{D}_2$ updates its state to $i+1$ and outputs 12 bytes:

$(i+1, [B_0^i, \ldots, B_{11}^i]) \xleftarrow{\$} \mathcal{D}_2(i)$, where $B_4^{10i} \leftarrow \mathcal{B}_{7,\$}, B_5^{10i} \leftarrow \mathcal{B}_{3,b}, B_4^{10i+2} \leftarrow \mathcal{B}_{2,b}, B_7^{10i+4} \leftarrow \mathcal{B}_{5,b}$, $B_6^{10i+6} \leftarrow \mathcal{B}_{1,b}, B_{10}^{10i+8} \leftarrow \mathcal{B}_{0,b}$, with $b = B_{4,7}^i$

For each state i, $\mathcal{D}_2$ outputs a 12-bytes input containing 1 bit of random data (for $i = 0 \bmod 10$) or 0 bit of random data (for $i \neq 0 \bmod 10$). If $d = 0$, $k = 127$ and S is known, and noting $S^t = \mathsf{refresh}(S, \mathsf{refresh}(S^{t-1}, [B_0^{t-1}, \ldots, B_{11}^{t-1}]))$, $S^t = S_0^t, \ldots, S_{127}^t$, then S^t contains 1 random bit in word S_{127}^t, at position 10, for all t. Distribution $\mathcal{D}_2$ outputs B_4^0 and B_5^0 are illustrated in Figure 7.2. □

Attacks Against the Robustness of LINUX

In this section we describe attacks on LINUX that prove Theorem 21. The first three attacks use distributions (described in Lemma 11 and Lemma 12) that fool the Entropy Estimator and the last attack uses the distribution for which the Mixing function is counter productive (described in Lemma 13). For this last attack, we show indeed that LINUX is not even backward secure.

Remark 5. *It is important to mention that our attacks do not use any computation of the adversary $\mathcal{A}$ that is correlated with the hash function family H_K. Note however that the attack based on the mixing function uses as a prerequisite that computation of the adversary $\mathcal{A}$ is done knowing the definition of the mixing function. In particular, the set $\{0, 1, 25, 51, 76, 103\}$ that is used to mix new inputs in the internal state could have been randomly chosen and selected as* seed*. In this situation, the proposed attack does not fit in the security model because the assumption about the independence between* seed *and the input distribution is not satisfied.*

Attacks Based on the Entropy Estimator As shown in Section 7.2, it is possible to build a distribution $\mathcal{D}_0$ of null entropy for which the estimated entropy is high (*cf.* Lemma 11) and

a distribution $\mathcal{D}_1$ of high entropy for which the estimated entropy is null (*cf.* Lemma 12). It is then possible to mount attacks on both `/dev/random` and `/dev/urandom`, which show that these two generators are not robust. At first we describe two attacks that use the blocking behavior on input, one attack on `/dev/random` and one attack on `/dev/urandom` and secondly we describe an attack (that works on both `/dev/random` and `/dev/urandom`) that does not use this behavior but uses the way entropy estimation evolves when data is transferred between the pools.

`/dev/random` is not robust. Let us consider an adversary $\mathcal{A}$ against the robustness of the generator `/dev/random`, and thus in the game $\mathsf{ROB}(\gamma^*)$, that makes the following oracle queries: one get-state, several next-ror, several $\mathcal{D}$-refresh and one final next-ror.

Then the state (S_i, S_r, S_u), the parameters k, d, E_i, E_u, E_r and the counter c defined in $\mathsf{ROB}(\gamma^*)$ evolve the following way:

- get-state: After a state compromise, $\mathcal{A}$ knows all parameters (but needs S_i, S_r, E_i, E_r) and $c = 0$.
- next-ror: After $\lfloor E_i/10 \rfloor + \lfloor E_r/10 \rfloor$ queries to next-ror, $E_i = E_r = 0$, $\mathcal{A}$ knows S_i and S_r and $c = 0$.
- $\mathcal{D}$-refresh: In a first stage, $\mathcal{A}$ refreshes LINUX with input from $\mathcal{D}_0$. After 300 queries, $E_i = 3584$ and $E_r = 0$. $\mathcal{A}$ knows S_i and S_r and $c = 0$.
  In a second stage, $\mathcal{A}$ refreshes LINUX with input $J \xleftarrow{\$} \mathcal{U}_{128}$. As $E_i = 3584$, these inputs are ignored as long as I contains less than 4096 bytes. After 30 queries, $\mathcal{A}$ knows S_i and S_r and $c = 3840$.
- next-ror: Since $E_r = 0$, a transfer is necessary between S_i and S_r before generating R. Since $E_i = 3584$, then $\alpha_r = 10$, such a transfer happens. But as $\mathcal{A}$ knows S_i and S_r, then $\mathcal{A}$ knows R.

Therefore, in the game $\mathsf{ROB}(\gamma^*)$ with $b = 0$, $\mathcal{A}$ obtains a 10-bytes string in the last next-ror-oracle that is predictable, whereas when $b = 1$, this event occurs only with probability 2^{-80}. It is therefore straightforward for $\mathcal{A}$ to distinguish the real and the ideal world.

The attack on `/dev/urandom` is very similar to the previous one, using the blocking behavior on input.

`/dev/urandom` is not robust. Similarly, let us consider an adversary $\mathcal{A}$ against the robustness of the generator `/dev/urandom` in the game $\mathsf{ROB}(\gamma^*)$ that makes the following oracle queries: one get-state that allows it to know S_i, S_u, E_i, E_u; $\lfloor E_i/10 \rfloor + \lfloor E_u/10 \rfloor$ next-ror, making $E_i = E_u = 0$; 100 $\mathcal{D}$-refresh with $\mathcal{D}_1$; and one next-ror, so that R will only rely on S_u as no transfer is done between S_i and S_u since $E_i = 0$. Then $\mathcal{A}$ is able to generate a predictable output R and to distinguish the real and the ideal worlds in $\mathsf{ROB}(\gamma^*)$.

Now we present a third attack, on both `/dev/random` and `/dev/urandom`, that exploits the way entropy estimation evolves when data is transferred between the input pool and the output pools. We describe it for `/dev/random`, but this attack is indeed exactly the same for `/dev/urandom`.

`/dev/random` and `/dev/urandom` are not robust. Let us consider an adversary $\mathcal{A}$ against the robustness of the generator `/dev/random`, and thus in the game $\mathsf{ROB}(\gamma^*)$, that makes the following oracle queries: one get-state, several next-ror, several $\mathcal{D}$-refresh, several next-ror, several refresh and one final next-ror.

Then the state (S_i, S_r, S_u), the parameters k, d, E_i, E_u, E_r and the counter c defined in $\mathsf{ROB}(\gamma^*)$ evolve the following way:

- get-state: After a state compromise, $\mathcal{A}$ knows all parameters (but needs S_i, S_r, E_i, E_r) and $c = 0$.

- next-ror: After $\lfloor E_i/10 \rfloor + \lfloor E_r/10 \rfloor$ queries to next-ror, $E_i = E_r = 0$, $\mathcal{A}$ knows S_i and S_r and $c = 0$.

- $\mathcal{D}$-refresh: $\mathcal{A}$ refreshes LINUX with input from $\mathcal{D}_0$. After 300 queries, $E_i = 3584$ and $E_r = 0$. $\mathcal{A}$ knows S_i and S_r and $c = 0$.

- next-ror with $t = 1$: Since $E_r = 0$, a transfer is necessary between S_i and S_r before generating R. Then $\alpha_r = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor) = 8$, $E_i = E_i - 8\alpha_r = 3534 - 64 = 3520$, $E_r = E_r + 8\alpha_r - 8t = 56$.

- next-ror with $t = 7$: Since $E_r = 56$, no transfer is necessary, and $E_r = E_r - 8t = 0$.

- Repeat the two previous queries until $E_i = 0$, and do only the first next-ror query (with $t = 1$) for the last.

- refresh with input $J \xleftarrow{\$} \mathcal{U}_{128}$. After 10 queries, $\mathcal{A}$ knows S_r and $c = 1280$.

- next-ror, with $t = 7$: Since $E_r = 56$, no transfer is necessary between S_i and S_r before generating R. But as $\mathcal{A}$ knows S_r, then $\mathcal{A}$ knows R.

Therefore, in the game $\mathsf{ROB}(\gamma^*)$ with $b = 0$, $\mathcal{A}$ obtains a 7-bytes string in the last next-ror-oracle that is predictable, whereas when $b = 1$, this event occurs only with probability 2^{-56}. It is therefore straightforward for $\mathcal{A}$ to distinguish the real and the ideal world.

Attack based on the Mixing Function. In [LRSV12], a proof of state entropy preservation is given for one iteration of the mixing function M, assuming that the input and the internal state are independent, that is: $\mathbf{H}_\infty(\mathsf{M}(S, I)) \geq \mathbf{H}_\infty(S)$ and $\mathbf{H}_\infty(\mathsf{M}(S, I)) \geq \mathbf{H}_\infty(I)$. We show that without that independence assumption and with more than one iteration of M, the generator LINUX does not recover from state compromise. This contradicts the backward security and therefore the robustness property.

LINUX is not backward secure. As shown in Section 7.2, with Lemma 13, it is possible to build an input distribution $\mathcal{D}_2$ with arbitrary high entropy such that, after several $\mathcal{D}$-refresh, $\mathbf{H}_\infty(S) = 1$. Let us consider an adversary $\mathcal{A}$ that generates an input data of distribution $\mathcal{D}_2$, and that makes the following oracle queries: set-refresh, and γ^* calls to $\mathcal{D}$-refresh followed by many calls to next-ror. Then the state (S_i, S_r, S_u), the parameters k, d, E_i, E_u, E_r and the counter c of $\mathsf{BWD}(\gamma^*)$ evolve the following way:

- set-refresh: $\mathcal{A}$ sets $S_i = 0$, $S_r = S_u = 0$, $d = 0$ and $k = 127$, and $c = 0$.

- $\mathcal{D}$-refresh: $\mathcal{A}$ refreshes LINUX with $\mathcal{D}_2$. After γ^* oracle queries, until $c \geq \gamma^*$, the new state still satisfies $\mathbf{H}_\infty(S) = 1$.

- next-ror: Since $\mathbf{H}_\infty(S) = 1$, $\mathbf{H}_\infty(R) = 1$.

Therefore, in the game $\mathsf{BWD}(\gamma^*)$ with $b = 0$, $\mathcal{A}$ always obtains an output in the last next-ror query with $\mathbf{H}_\infty(R) = 1$, whereas in $b = 1$, this event occurs only with negligible probability. It is therefore straightforward for $\mathcal{A}$ to distinguish the real and the ideal world.

7.3 Analysis of OpenSSL Generator

The OpenSSL cryptographic library contains a pseudo-random number generator with input OPENSSL which collects entropy from system calls. Is has been first analyzed by Gutmann in 1998 [Gut98] and since then no new analysis has been made. It is implemented in the source file /crypto/rand/md_rand.c, as part of the OpenSSL library. The generator takes inputs of any size and generates outputs of size 10 bytes. The generator is different depending on a choice made when building the library. This choice depends on an internal parameter named `MD_DIGEST_LENGTH`, which depends on the underlying hash function used. The choice of the hash function is made using with a dedicated flag (`USE_MD5_RAND` for the MD5 function, or `USE_SHA1_RAND` for the SHA1 function), which is by default `USE_SHA1_RAND`. Hence depending on the environment, the size of S_3 is equal to 16 bytes or 20 bytes. We assume that the SHA1 function is used in our descriptions, hence we will refer to the hash functions family H_K described in Section 7.1. We verified that our attack can be easily adapted if `USE_MD5_RAND` is chosen.

Internal State Decomposition. The internal state of OPENSSL is implemented with five fields: `state_index`, of size 32 bits, `state`, of size 1043 bytes, `md`, of size 20 bytes, `md_count_0`, `md_count_1`, each of size 64 bits. The decomposition of the internal state is given by $S = (S_1, S_2, S_3, S_4, S_5)$, where S_1, S_2, S_3, S_4, S_5 stand for `state_index`, `state`, `md`, `md_count_0`, `md_count_1`, respectively. The total size of the internal state is 8576 bits and the generator uses this decomposition as follows: field S_1 is used as an index to select bytes in S_2; S_2 and S_3 are used to collect entropy; S_4 and S_5 are counters used during the generator operations.

The refresh Algorithm. This algorithm is implemented with the instruction `ssleay_rand_add`, and fully described in Algorithm 12. It takes as input the current internal state $(S_1, S_2, S_3, S_4, S_5)$ and an input I of any size that is processed by blocks of 20 bytes. Starting with a 20-bytes block of S_2 that is indexed by S_1, consecutive blocks of S_2 are mixed with consecutive blocks of I. The mixing operation involves the hash functions family H_K. This mixing operation also involves S_3, S_4 and S_5, where S_5 is incremented for each block. When this mixing is finished, the field S_3 is xor-ed with the last calculated hash. Hence after a refresh operation, $|I|$ bits of S_2 are modified, S_3 is modified, S_1 and S_5 are incremented and S_4 is not modified.

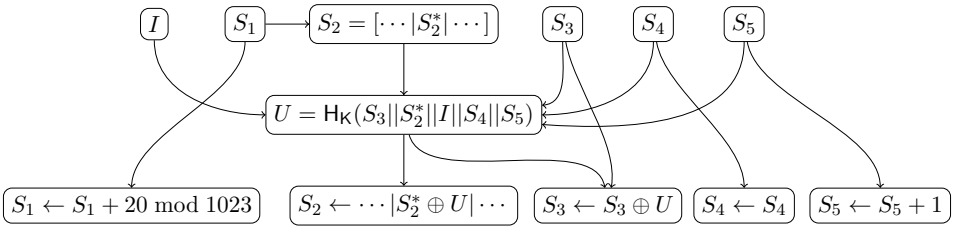

Figure 7.3 – OPENSSL refresh Algorithm

The next Algorithm. This algorithm is implemented in ssleay_rand_bytes, and described in Algorithm 13. It takes as input the current internal state $(S_1, S_2, S_3, S_4, S_5)$, mixes S_2, S_3, S_4 and S_5 together to produce the 10-byte output R and updates S_3. Only 10 bytes from S_2 are modified, that are selected using field S_1, which behaves as an index for this operation. A second mixing operation involves S_3, S_4 and S_5 to update S_3. Hence S_2 is modified sequentially by blocks of 10 bytes with successive next calls, while S_3 is completely modified, S_1 and S_4 are incremented and S_5 is not changed. As for the refresh algorithm, the two mixing operations involve the hash function family H_K.

Note that directive `ssleay_rand_bytes` takes as input an array named `buf` which is filled with the generated output, but whose content is also used as input (referenced as I in the description below). In addition, the next algorithm uses as input the current system PID and the system

Algorithm 12 OPENSSL refresh algorithm

Require: $S = (S_1, S_2, S_3, S_4, S_5)$, I
Ensure: S'
1: **while** $|I| > 0$ **do**
2: $S_2^* = S_2[S_1 \bmod 1023, \ldots, S_1 + 20 \bmod 1023]$
3: $U = \mathsf{H_K}([S_3||S_2^*||I||S_4||S_5])$
4: $S_2^* = S_2^* \oplus U$
5: $S_1 = S_1 + 20 \bmod 1023$
6: $S_5 = S_5 + 1$
7: $I = I \setminus [I]_0^{19}$
8: **end while**
9: $S_3 = S_3 \oplus U$
10: **return** $S' = (S_1, S_2, S_3, S_4, S_5)$

time. The system PID is obtained with a call to directive `getpid`, system time is obtained from a call to directive `time`, and from a call to directive `gettimeofday` (for simplicity, we refer to these two calls as "Time" in the description of the generator). These inputs during the next algorithm are not explicitly compliant with the security model that requests a strict separation between the input collection and the generation, but we mention it for completeness of the description. These calls have been explicitly set by OpenSSL community to prevent a vulnerability related to a call to the `fork` function that uses a common PID for two next calls. This vulnerability is described in [Ope13].

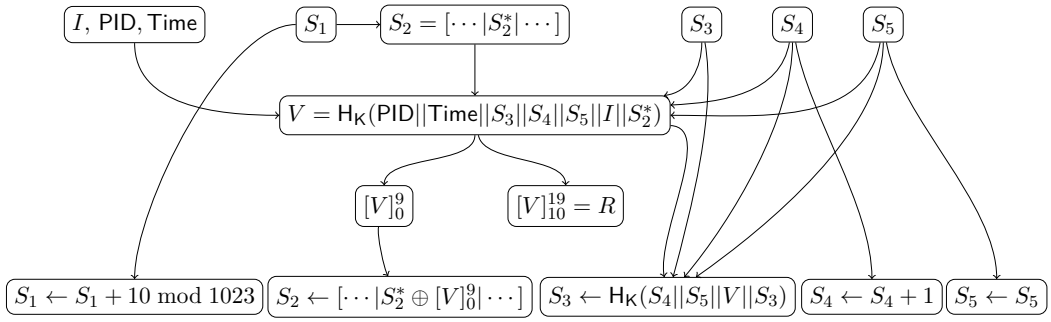

Figure 7.4 – OPENSSL next Algorithm

Algorithm 13 OPENSSL next algorithm

Require: $S = (S_1, S_2, S_3, S_4, S_5)$
Ensure: S', R
1: $S_2^* = S_2[S_1 \bmod 1023, \cdots S_1 + 10 \bmod 1023]$
2: $V = \mathsf{H_K}([\mathsf{PID}||\mathsf{Time}||S_3||S_4||S_5||I||S_2^*])$
3: $S_2^* = S_2^* \oplus V[0, \cdots, 9]$
4: $R = V[10, \cdots, 19]$
5: $S_3 = \mathsf{H_K}([S_4||S_5||V||S_3])$
6: $S_1 = S_1 + 10 \bmod 1023$
7: $S_4 = S_4 + 1$
8: **return** $S' = (S_1, S_2, S_3, S_4, S_5), R$

Attack. We mount a memory attack against the pseudo-random number generator with input OPENSSL, that is based on the internal state decomposition and the fact that this state is only partially updated refresh and next algorithms. This attack uses the field S_3, which is implemented with `md` and the field S_2– which is implemented with `state`. As described in Algorithms 12 and 13, when the generator is refreshed, the field S_3 is updated with the *last* calculated hash, whereas it is used as the entropy source for the output of the generator with 10 bytes of S_2. Suppose now that one uses an input of size 1023 bytes (which is the size of S_2 – or `state`) where the first 20 bytes and the last 3 bytes are 0, to refresh the generator. Clearly this input is independent of

the parameter seed and it is therefore legitimate to use it to refresh the generator in our security model. Suppose now that one asks for an output. This output, which only relies on the first 10 bytes of S_2 and on S_3, is predictable. Theorem 22 gives the technical details of the attack. This attack is related to the refresh function that mixes new entropy sequentially by blocks of 20 bytes in the internal state, and to the next function that also reads sequentially the internal state by blocks to produce new outputs. If a block is compromised and if the adversary controls the exact block of the input that will be mixed with the compromised block of the internal state, the output is predictable. Hence the attack points a design error of the generator, because this behavior should not be possible.

Theorem 22. *The pseudo-random number generator with input* OPENSSL *is not* backward secure against memory attacks. *To mount an attack against the generator, $\mathcal{A}$ needs to corrupt 40 bytes of the internal state.*

Proof. Define the 1023-byte distribution $\mathcal{D}$. On input a state i, $\mathcal{D}$ updates its state to $i+1$ and outputs a 1023-byte input I^i: $(i+1; [I^i_0, \cdots, I^i_{1022}]) \leftarrow \mathcal{D}(i)$; where $I^0_0 = \cdots = I^0_{19} = 0$, $I^0_{1019} = \cdots = I^0_{1022} = 0$ and all other bytes are random (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_i = 8000$). Let us define the mask $M = [M_1, M_2, M_3, M_4, M_5]$, where $M_1 = 0$, $[M_2]^{19}_0 = 0$, $M_3 = 0$, $M_4 = 0$, $M_5 = 0$ and $J = \{2, 3\}$ (*i.e.* this mask will be used to set the first 20 bytes of S_2 and S_3 to 0). 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 MBWD: one $\mathcal{M}$-set-state with S, J and M, one $\mathcal{D}$-refresh with I^0, one next-ror. Then (following refresh and next algorithm notations):

- After $\mathcal{M}$-set-state, $S_1 = 0$, $[S_2]^{19}_0 = 0^{19}$, $[S_2]^{1023}_{20}$ is random, $S_3 = 0$, $S_4 = 0$, $S_5 = 0$.
- After $\mathcal{D}$-refresh,
  - $S_1 = 0$, $[S_2]^{19}_0 = \mathsf{H}_\mathsf{K}([0||0||0||0||0])$, $[S_2]^{1023}_{20}$ is random, $S_3 = \mathsf{H}_\mathsf{K}([0||0||0||0||51])$,
  - $S_4 = 0$, $S_5 = 51$.
- After next-ror,
  - $V = \mathsf{H}_\mathsf{K}(\mathsf{PID}||\mathsf{Time}||S_3||0||51||[S_2]^{19}_0)$, $R = V^{19}_{10}$, $S_3 = \mathsf{H}_\mathsf{K}(0||51||V||\mathsf{H}_\mathsf{K}(0||0||0||0||51))$,
  - $S_1 = 10$, $S_4 = 1$.

In this last next-ror-oracle query, $\mathcal{A}$ obtains a 10-bytes string that is predictable as it only relies on PID and Time, whereas this event should occur with probability 2^{-80}. Therefore $\mathcal{A}$ can distinguish an output of OPENSSL from random in the game $\mathsf{BWD}(\gamma^*, 320)$, for all $\gamma^* \leq 8000$ and this pseudo-random number generator with input is not backward secure. □

7.4 Analysis of Android `SHA1PRNG`

In the Android system, a full Java implementation of a pseudo-random number generator with nput is provided, as part of the package `security.provider.crypto`, named `SHA1PRNG`. It has been analyzed by Michaelis *et al.* in [MMS13], where the authors identified an implementation weakness that causes the internal state to be overwritten by predictable values, decreasing its entropy to 64 bits. This generator was also debated intensively recently, due to a weakness in its initial seeding that caused a flaw in Bitcoin wallets. This weakness caused the Android community to propose a fix to the generator, that simply consists in replacing it by the one from OpenSSL, analyzed in Section 7.3. Full details about the vulnerability and the proposed fix are given in [And13]. The generator is implemented with the class `SHA1PRNG_SecureRandomImpl`

and is an inheritance from the one included in the library Apache Harmony from the package `org.apache.harmony`. It follows the method named "expansion of source bits" of IEEE standard P.1363 [BK96].

Internal State Decomposition. The internal state of the generator is implemented with the fields `seed`, of size 384 bytes, and `counter`, of size 8 bytes (many other fields are used, but they are not useful to understand the operations). Hence the decomposition of the internal state is $S = (S_1, S_2)$, where S_1, S_2 stand for `seed` and `counter` and the total size of the internal state is 392 bytes. The generator uses this decomposition as follows: S_1 contains the collected entropy and a hash of the collected entropy; S_2 contains a counter which is incremented at each output.

The refresh Algorithm. This algorithm is described in Algorithm 14. It takes as input the current internal state (S_1, S_2), an input I of any size and updates the internal state with I. It is implemented with method `engineSetSeed` as follows: the first 64 bytes of S_1 collect the consecutive inputs and the last 20 bytes of S_1 contains a hash value. Two sub-functions are used, implemented with `SHA1Impl.updateHash` and `SHA1Impl.computeHash`. Note that these two functions correspond respectively to the *update* of the internal state of $\mathsf{H_K}$ and a function that compress the input of $\mathsf{H_K}$ to a fixed length output as is defined in the specification [SHA95]. As the generator uses (wrongly, as we will see) the compression function for its operation, we will use it for its description and will refer to this function as $\mathsf{h_K}$. When the collected input fills a block of $\mathsf{h_K}$ (of size 64 bytes), the last 20 bytes of S_1 are filled with $\mathsf{h_K}$, and then the block is set to 0 and filled again. For clarity, we denote s the current collected input, h the current calculated hash in S_1 and $I^* = [I]_0^{64-|S||I|}$ in the descriptions.

Algorithm 14 Android `SHA1PRNG` refresh

Require: $S = (S_1, S_2) = ([s, \cdots, h], S_2), I$
Ensure: S'
1: **if** $|s||I| < 64$ **then** $S_1[0, \cdots, 63] = [s||I]$ **end if**
2: **if** $|s||I| = 64$ **then**
3: $S_1[0, \cdots, 63] = 0$, $S_1[328, \cdots, 347] = \mathsf{h_K}(s||I, h)$
4: **end if**
5: **if** $|s||I| > 64$ **then**
6: $S_1[0, \cdots, 63] = I \setminus I^*$, $S_1[328, \cdots, 347] = \mathsf{h_K}(s||I^*, h)$
7: **end if**
8: **return** $S' = (S_1, S_2)$

The next Algorithm. This algorithm is described in Algorithm 15. It is implemented with `engineNextBytes`. It takes as input an integer n, outputs R, of size n bytes and the updated internal state S'. Twenty successive bytes outputs are generated as follows: the algorithm appends S_1 and S_2, calculates the output with function $\mathsf{h_K}$ (the compression function) and increments the counter contained in S_2. For clarity, we suppose that n is a multiple of 20 (the implementation allows any value with intermediate arrays whose description would complicate the understanding of the algorithm) and we denote c the counter contained in S_2. We also use the same notation (s and h) used for the refresh algorithm.

Algorithm 15 Android `SHA1PRNG` next

Require: $S = (S_1, S_2) = ([s, \cdots, h], [c]), n(n \bmod 20 = 0)$
Ensure: S', R
1: **for** $i = 0$ to $n - 1$ **do**
2: $S_1[0, \cdots, 63] = [s||c]$, $S_1[328, \cdots, 347] = \mathsf{h_K}(s||c, h)$
3: $c = c + 1, S_2 = [c]$
4: $R_i = S_1[328, \cdots, 347]$
5: $i = i + 20$
6: **end for**
7: **return** $S', R = (S_1, S_2), \underset{i}{\cup} R_i$

Attack. We mount one attack against the Android SHA1PRNG taking in consideration the internal state decomposition. Our attack is possible because of the use of the compression function $\mathsf{h_K}$ instead of the hash function $\mathsf{H_K}$, both in the refresh and next algorithms. When using the compression function $\mathsf{h_K}$, the *current* hash value is used whereas the hash should be calculated with the initialization vector defined in the specification [SHA95]. Again, this attack identifies a design flaw of the generator. This attack shows that the generator is not *resilient* because the attacker only needs to refresh the generator with an input that forces S_1 to be equal to [0]. In addition, if at initialization the internal state is filled with 64 random bytes, the generator is not even pseudo-random, because no refresh is needed. The attack is demonstrated in Theorem 23.

Theorem 23. *The pseudo-random number generator with input Android* SHA1PRNG *is not resilient.*

Proof. Consider an adversary $\mathcal{A}$ against the security of the generator that chooses the following (one state) distribution $\mathcal{D}$, $\mathcal{D}(0) = I$, where I is of size ℓ, where $\ell \leq 512$ and uniformly random (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_0 = \ell$). Then $\mathcal{A}$ makes the following oracle queries in the security game RES: one $\mathcal{D}$-refresh, one first next-ror with an output R_1 of size 20 bytes, and one second next-ror, with an output R_2 of size 20 bytes. Then:

- After $\mathcal{D}$-refresh with I: $[S_1]_0^{63} = 0^{64}$ with probability 1/64, $[S_1]_{328}^{347}$ is random, $S_2 = 0$.
- After next-ror with R_0, $[S_1]_0^{63} = 0^{64}$ with probability 1/64, $R_0 = [S_1]_{328}^{347}$ and $S_2 = 1$, as S_1 is not modified.
- After next-ror with R_1, $[S_1]_0^{63} = [0||1]$, $R_0 = [S_1]_{328}^{347}$, but $[S_1]_{328}^{347} = \mathsf{h_K}(0, R_0)$ with probability 1/64.

In this last next-ror-oracle query, $\mathcal{A}$ obtains a 20-byte string that is known to $\mathcal{A}$ with probability 1/64 as it only relies on the previous output, whereas ideally, this event should occur only with probability 2^{-80}. Therefore this pseudo-random number generator with input is not resilient. □

7.5 Analysis of OpenJDK SHA1PRNG

The OpenJDK provider contains an implementation of a pseudo-random number generator with input, named SHA1PRNG, directly given in the class SecureRandom. This implementation follows the specification given in the Digital Signature Standard [DSS00]. This specification has been analyzed in [KSWH98], where the authors show that it not a *resilient* pseudo-random number generator with input. Here we present new attacks that concern partial corruption of its internal state of the implementation.

Internal State Decomposition. The internal state of the generator is implemented with three private fields, the field state, of size 20 bytes, the field remainder, of size 20 bytes and an integer remCount. The decomposition of the internal state is $S = (S_1, S_2, S_3)$, where S_1, S_2, S_3 stand for state, remainder, remCount, respectively. The generator uses this decomposition as follows: S_1 contains the collected entropy, S_2 contains random bytes before their output and S_3 is used to check if S_2 contains enough random bytes that can serve as output.

The refresh Algorithm. This algorithm is described in Algorithm 16. It is implemented with method engineSetSeed. It takes as input the current internal state $S = (S_1, S_2, S_3)$, a new input I and outputs the new internal state by mixing S_1 with I using $\mathsf{H_K}$.

The next Algorithm. This algorithm is described in Algorithms 17 and 18. It is implemented with two methods; the first one, engineNextBytes, generates the output and the second one,

Algorithm 16 OpenJDK `SHA1PRNG` refresh

Require: $S = (S_1, S_2, S_3), I$
Ensure: S'
1: $S_1 = \mathsf{H}_\mathsf{K}(S_1 || I)$
2: **return** $S' = (S_1, S_2, S_3)$

`updateState`, updates the internal state.

The method `engineNextBytes` takes as input the current internal state $S = (S_1, S_2, S_3)$ and n, the number of bytes requested. It outputs an n-byte output R and updates the internal state. The internal counter S_3 controls the update of the internal state when output is generated: if $S_3 > 0$ it means that S_2 contains some bytes that have not been used for a previous output, these bytes can be used for the current output and are then set to 0. Next, S_2 and S_1 are updated only if all bytes from S_2 have been used: at first S_2 is updated with S_1 ($S_2 = \mathsf{H}_\mathsf{K}(S_1)$) and then, S_1 is updated using `updateState` instruction.

Algorithm 17 OpenJDK `SHA1PRNG` next (`engineNextBytes`)

Require: $S = (S_1, S_2, S_3), n$
Ensure: S, R
1: $i = t = 0$
2: **if** $S_3 > 0$ **then**
3: $\quad t = \min n - i, 20 - S_3$
4: $\quad R[0, \cdots, t-1] = S_2[S_3, \cdots, S_3 + t - 1]$
5: $\quad S_2[S_3, \cdots, S_3 + t - 1] = [0]$
6: **end if**
7: **while** $i < n - 1$ **do**
8: $\quad S_2 = \mathsf{H}_\mathsf{K}(S_1)$
9: $\quad S_1 = \texttt{updateState}(S_1, S_2)$
10: $\quad t = \min n - i, 20$
11: $\quad R[i, \cdots, i + t - 1] = S_2[0, \cdots, t - 1]$
12: $\quad i \leftarrow i + t$
13: **end while**
14: $S_3 = (S_3 + n) \bmod 20$
15: **return** S_1, S_2, S_3, R

The method `updateState` is the implementation of the update algorithm specified in [DSS00]. It takes as input two binary strings S_1 and S_2 of size 20 bytes and mixes them together byte by byte.

Algorithm 18 OpenJDK `SHA1PRNG` next (`updateState`)

Require: S_1, S_2, $|S_1| = |S_2| = 160$
Ensure: S_1
1: $\ell = 1$
2: **for** $i = 0$ to 19 **do**
3: $\quad v = (S_1[i] + S_2[i] + \ell)$
4: $\quad S_1[i] = v \bmod 2^8$
5: $\quad \ell = v / 2^8$
6: **end for**
7: **return** S_1

Attack. We mount a memory attack against the OpenJDK `SHA1PRNG` taking in consideration the internal state decomposition. Our attack uses the fact that S_2 and S_3 are not updated during refresh. After a refresh, if S_3 is set by the attacker to 1, the next output will be derived from a predictable value.

Theorem 24. *The pseudo-random number generator with input OpenJDK* `SHA1PRNG` *is not* backward secure against memory attacks*. To mount an attack against the generator, $\mathcal{A}$ needs to corrupt 4 bytes of the internal state.*

Proof. Let us consider an adversary $\mathcal{A}$ against the security of the OpenJDK SHA1PRNG that chooses the distribution $\mathcal{D}$, such that $\mathcal{D}(0) = I$ where I is of size 20 bytes and random (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_0 = 160$). Then $\mathcal{A}$ makes the following oracle queries in the security game MBWD: one $\mathcal{D}$-refresh, one $\mathcal{M}$-set-state with $M = (0, 0, 1), J = \{3\}$ and one final next-ror with an output R of size 10 bytes. Then:

- After $\mathcal{D}$-refresh with I, $S_1 = \mathsf{H}_\mathsf{K}(I||0)$, $S_2 = 0$ and $S_3 = 0$.
- After one $\mathcal{M}$-set-state with $M = (0, 0, 1)$, $J = \{3\}$, $S_1 = \mathsf{H}_\mathsf{K}(I||0)$, $S_2 = 0$ and $S_3 = 1$.
- After one next-ror with $n = 10$, $S_1 = \mathsf{H}_\mathsf{K}(I||0)$, $S_2 = 0$, $S_3 = 11$ and $R = 0$.

Therefore, $\mathcal{A}$ obtains a 10-bytes string in the last next-ror-oracle query that is predictable whereas this event should occur with probability 2^{-80}. Therefore this pseudo-random number generator is not backward secure for $\gamma^* \leq 160$. Note that as the fields S_2 and S_3 are not updated during the refresh Algorithm, $\mathcal{A}$ could make sufficient calls to $\mathcal{D}$-refresh to mount a similar attack for a larger value of γ^*. □

7.6 Analysis of Bouncycastle SHA1PRNG

The Bouncy Castle Crypto package is a Java implementation of cryptographic algorithms; our analysis refers to release 1.5 [Bou]. The implementation of several pseudo-random number generators with input is provided in the package `org.bouncycastle.crypto.prng`, where the implementation of the pseudo-random number generator with input SHA1PRNG is in the class `DigestRandomGenerator`. The implementation combines a cryptographic hash function (which is by default H_K) with internal instructions that are used to update the internal state of the generator. In our source code analysis, we identified firstly a weakness related to the decomposition of the internal state, and secondly a weakness due to an incomplete state update during the refresh algorithm. These weaknesses have neither been identified in [MMS13], nor by the Bouncycastle community.

Internal State Decomposition. The internal state of the generator is implemented with the following fields: `seed` of size 160 bits, `state` of size 160 bits, `seedCounter` of size 64 bits, and field `stateCounter`, of size 64 bits. The two first contain the collected entropy and the two last are counters that are used for its operations. The total size of the internal state is 448 bits and its decomposition is $S = (S_1, S_2, S_3, S_4)$, where S_1, S_2, S_3, S_4 stand for `seed`, `state`, `seedCounter`, `stateCounter`, respectively.

The refresh Algorithm. This algorithm is fully described in Algorithm 19. It takes as input the current internal state (S_1, S_2, S_3, S_4) and an input I; it outputs a new internal state where only S_1 is updated. It is implemented with the method `addSeedMaterial`.

Algorithm 19 Bouncycastle SHA1PRNG refresh

Require: $S = (S_1, S_2, S_3, S_4)$, I
Ensure: S'
1: $S_1 = \mathsf{H}_\mathsf{K}(S_1||I)$
2: **return** $S' = (S_1, S_2, S_3, S_4)$

The next Algorithm. This algorithm is described in Algorithms 20 and 21. It is implemented with the method `NextBytes`. It takes as input an integer n, the current the internal state (S_1, S_2, S_3, S_4) and outputs an n-byte string R. The output R is derived from S_2, while an internal method, named `generateState` is used to update the state.

Algorithm 20 Bouncycastle `SHA1PRNG` next (`NextBytes`)

Require: $S = (S_1, S_2, S_3, S_4)$, n
Ensure: S'
1: $S =$ `generateState`(S)
2: $j = n$
3: **for** $i = 0$ to j **do**
4: **if** j $= 20$ **then**
5: $S =$ `generateState`(S)
6: $j = 0$
7: **end if**
8: $R[i] = S_2[i]$
9: $i = i + 1$
10: **end for**
11: **return** $S' = (S_1, S_2, S_3, S_4)$, R

Algorithm 21 Bouncycastle `SHA1PRNG` next: (`generateState`)

Require: $S = (S_1, S_2, S_3, S_4)$
Ensure: S'
1: $S_4 = S_4 + 1$
2: $S_2 = \mathsf{H_K}(S_4||S_2||S_1)$
3: **if** $S_3 \bmod 10 = 0$ **then**
4: $S_3 = S_3 + 1$
5: $S_1 = \mathsf{H_K}(S_1||S_3)$
6: **end if**
7: **return** $S' = (S_1, S_2, S_3, S_4)$

The `generateState` instruction increments the counters S_3 and S_4 and calculates the new values of S_1 and S_2 accordingly.

Attack. We mount an attack against the Bouncycastle `SHA1PRNG` taking in consideration the internal state decomposition. This attack is similar as the attack against [DSS00] described in [KSWH98]: the attacker uses a previously generated output as an input to corrupt the generator: our attack shows that Bouncycastle `SHA1PRNG` is not *resilient*.

Theorem 25. *The pseudo-random number generator with input Bouncycastle* `SHA1PRNG` *is not resilient.*

Proof. Consider an adversary $\mathcal{A}$ against the resilience of the generator that chooses the following (2-state) distribution $\mathcal{D}(0) = (1, I, 160, \varnothing)$ and $\mathcal{D}(1) = (2, J, 0, 160)$, where I and J are of size 20 bytes, I is random and J is known by $\mathcal{A}$ (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_0 = 160$ and $\gamma_1 = 0$). Then $\mathcal{A}$ makes the following oracle queries in the security game RES: one $\mathcal{D}$-refresh, two next-ror with two outputs R_1 and R_2, both of size 20 bytes, one $\mathcal{D}$-refresh, and one third next-ror, with one output R_3 of size 20 bytes. Then:

- After one $\mathcal{D}$-refresh with I, $S_1 = \mathsf{H_K}(I||0)$, $S_2 = 0$, $S_3 = 1$, $S_4 = 1$.
- After one next-ror, with $|R_1| = 20$, S_1 remains the same, $S_2 = \mathsf{H_K}(S_4||S_2||S_1) = \mathsf{H_K}(2||0||S_1)$, $S_3 = 1$, $S_4 = 2$, $R_1 = S_2$.
- After one second next-ror, with $|R_2| = 20$, S_1 remains the same, $S_2 = \mathsf{H_K}(S_4||S_2||S_1) = \mathsf{H_K}(3||R_1||S_1)$, $R_2 = S_2$.
- After one $\mathcal{D}$-refresh with $J = [3||R_1]$, $S_1 = \mathsf{H_K}(J||S_1) = \mathsf{H_K}(3||R_1||S_1) = R_2$.
- After one last next-ror with $|R_3| = 20$, S_1 remains the same, $S_2 = \mathsf{H_K}(S_4||S_2||S_1) = \mathsf{H_K}(4||R_2||R_2)$, $R_3 = S_2$.

Therefore, $\mathcal{A}$ obtains a 20-byte string in the last next-ror-oracle that is predictable ($R_3 = \mathsf{H_K}(4||R_2||R_2)$), whereas this event should occur with probability 2^{-80}. Therefore the pseudo-random number generator with input Bouncycastle `SHA1PRNG` is not resilient. □

7.7 Analysis of IBM SHA1PRNG

We analyze the pseudo-random number generator with input implemented in IBM's Java Virtual Machine. Besides Oracle's Java Virtual Machine, IBM implements its own JVM with some differences (in particular in performance) compared to Oracle's JVM. We analyze the IBM SDK Version 7 Service Refresh 7 which contains a security enhancement in the generator reported by Sethi in [IBM14]. We focus on the `SecureRandom` implementation of the crypto provider `IBMSecureRandom` located in the package `com.ibm.securerandom.provider`. This implementation consists of a main entropy pool and a mixing function which internally relies on the hash function family $\mathsf{H_K}$ to update the pool.

Internal State Decomposition. The internal state of IBM's generator is self-contained in the field `state` of size 680 bits. For convenience, we refer the field `state` as the set $S = (S_1||S_2||S_3||S_4||S_5||S_6||S_7)$. The generator uses this decomposition as follows: S_1 contains the number of bytes that has been used from the output pool, $S_2 = 0$, S_3 is the output, S_4 is a first entropy pool, S_5 are 5 different internal counters, S_6 is a second entropy pool and S_7 is a flag indicating whether the input is provided or not. The initial state is $S_1 = 0, S_2 = 0, S_3 = 0, S_4 = 0, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 = \mathsf{false}$ and it relies on the internal function `reverse` that simply reverses binary the content of the input.

The refresh algorithm. This algorithm is described in Algorithm 22. It takes as input the current internal state $(S_1, S_2, S_3, S_4, S_5, S_6, S_7)$, a input I and outputs the new internal state by mixing S_4 with I using $\mathsf{H_K}$. It is implemented with the method `engineSetSeed`.

Algorithm 22 IBM SHA1PRNG refresh

Require: $S = (S_1, S_2, S_3, S_4, S_5, S_6, S_7), I$
Ensure: S'
1: **if** $|I| > 320$ **then**
2: $\quad S_6 = \mathsf{H_K}(I)$
3: **end if**
4: $\bar{I} =$ `reverse(I)`
5: $S_4 = S_4 \oplus \bar{I}$
6: $S_7 = \mathsf{true}$
7: $S_1 = |S_3|$
8: **return** $S' = (S_1, S_2, S_3, S_4, S_5, S_6, S_7)$

The next algorithm. This algorithm is described in Algorithms 23 and 24. It is implemented with the methods `engineNextBytes` and `updateEntropyPool`. It takes as input the current internal state S and n, the number of bytes requested. It outputs an n-byte R and a new value for the internal state. It relies on S_1 to generate the output as follows: if $S_1 < |S_4|$ it means that S_3 still contains bytes that have not been used in a previous output. When S_1 reaches the size of the entropy pool (i.e. $S_1 = |S_4|$), S_3 and S_4 are updated to produce a fresh output. First entropy is added by the internal method `updateEntropyPool` and then the output pool $S_3 = \mathsf{H_K}(S_3||S_4||S_5||S_6[1])$ is updated. The instruction `time` returns the timestamp, δ is another timestamp value, and `firstTime` is an internal flag in order to ensure that S_3 is indeed filled. This procedure is repeated for each $|S_3|$ bytes.[3]

Attack. We mount an attack similar to the attack on OpenJDK `SHA1PRNG`. As in the refresh algorithm the internal state is not completely updated, an attacker can set the byte $S_1 = 0$ and make the counter of non-used bytes start reading again from $S_3[0]$. Notice that we need at least 3 bytes to set $S_1, S_5[4], S_5[5]$ properly otherwise the algorithm will force to add entropy; on the other hand, once all parameters are set up, an attacker just needs to corrupt 1 integer (4 bytes) to make the output predictable.

[3] In practice, the size of the output pool is 20-byte.

Algorithm 23 IBM `SHA1PRNG` next (`engineNextBytes`)

Require: $S = (S_1, S_2, S_3, S_4, S_5, S_6, S_7), n$
Ensure: S', R
1: **if** `firstTime` $=$ `true` **then**
2: **if** $S_1 = |S_3|$ **then**
3: $(S_4, S_5, S_7) =$ `updateEntropyPool`(S)
4: **end if**
5: $S_3 = \mathsf{H}_\mathsf{K}(S_3||S_4||S_5[0]||S_5[1])$
6: $S_1 = 0$
7: **end if**
8: $R = S_3[S_1, \ldots, n]$
9: $S_1 = 1 + n$
10: **return** $S' = (S_1, S_2, S_3, S_4, S_5, S_6, S_7), R$

Algorithm 24 IBM `SHA1PRNG` next (`updateEntropyPool`)

Require: $S = (S_1, S_2, S_3, S_4, S_5, S_6, S_7), I$
Ensure: S_4, S_5, S_7
1: **if** $S_5[1] > 0$ **and** $S_7 = FALSE$ **then**
2: **for** $S_5[0]$ **to** $S_5[0] + 20$ **do**
3: **if** `time` $\geq S_5[4] + S_5[5]$ **then**
4: $S_4 = S_4 \oplus I$
5: $S_5[4] = \delta$
6: $S_5[5] + S_5[2] +$ `time`
7: $S_5[0] + 1$
8: **end if**
9: **end for**
10: **end if**
11: **return** (S_4, S_5, S_7)

Theorem 26. *The pseudo-random number generator IBM* `SHA1PRNG` *is not backward secure. To mount an attack against the generator, $\mathcal{A}$ needs to corrupt 4 bytes of the internal state.*

Proof. Consider an adversary $\mathcal{A}$ against the security of IBM `SHA1PRNG` that chooses a distribution $\mathcal{D}$, such that $\mathcal{D}(0) = I$ where I is of size 20 bytes and random (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_0 = 160$). Then $\mathcal{A}$ makes the following oracle queries in the security game MBWD: one $\mathcal{D}$-refresh, one with an output of size 10 bytes, next-ror one $\mathcal{M}$-set-state with $M = (0, 0, 0, 0, 0, 0, 0), J = \{3\}$ and one final next-ror with an output of size 10 bytes. Then:

- After $\mathcal{D}$-refresh with I, $S_1 = |S_3|, S_2 = 0, S_3 = 0, S_4 = 0 \oplus I, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 = TRUE$.
- After one next-ror with $n = 10$, $S_1 = 10, S_2 = 0, S_3 = \mathsf{H}_\mathsf{K}(0||0 \oplus I||0||128), S_4 = 0 \oplus I, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 = TRUE$. $R = S_3[0, \ldots, 10]$. The output R is random.
- After one $\mathcal{M}$-set-state with $M = (0, 0, 0, 0, 0, 0, 0)$, $J = \{1\}$, $S_1 = 1, S_2 = 0, S_3 = \mathsf{H}_\mathsf{K}(0||0 \oplus I||0||128), S_4 = 0 \oplus I, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 = TRUE$.
- After one next-ror with $n = 10$, $S_1 = 10, S_2 = 0, S_3 = \mathsf{SHA1}(0||0 \oplus I||0||128), S_4 = 0 \oplus I, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 = TRUE$ and $R = S_3[0, \ldots, 10]$.

Therefore, $\mathcal{A}$ obtains a 10-byte string in the last next-ror-oracle query that is exactly the same as the previous next-ror-oracle query, whereas ideally, this event occurs only with probability 2^{-80}. Therefore the pseudo-random number generator IBM `SHA1PRNG` is not backward secure for $\gamma^* \leq 160$. Note that as the fields S_2 and S_3 are not updated during the refresh Algorithm, $\mathcal{A}$ could make sufficient calls to $\mathcal{D}$-refresh to mount a similar attack for a larger value of γ^*. □

Chapter 8

Conclusion and Perspectives

Security Models. The robustness model from [DPR+13] has a limitation, the seed dependence of the distribution used to generate inputs. Our proposed constructions crucially rely on the independence between the distribution sampler and seed, and we have shown that full seed dependence is impossible. Finding the right (realistic and, yet, provably secure) balance between these extremes is an important subject for further research. In [DSSW14], Dodis, Shamir, Stephens-Davidowitz and Wichs made some initial progress along these lines by introducing 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. They proved that the original construction of [DPR+13] can be extended in this model.

Security Analysis. Currently there are numerous implementations of pseudo-random number generators with input from different providers, and most of them rely on internal directives and parameters that are poorly documented or even undocumented. However, a flaw in the design can cause serious damages in cryptographic protocols, and vulnerabilities can be exploited by adversaries. Therefore widely used generators shall be analysed in a strong security model as the ones we propose in this thesis. For example, the design of the pseudo-random number generators with input in the Windows system relies on the Fortuna pseudo-random number generator [FSK10], which has been analyzed in [DSSW14], however the analysis of its implementation remains to be done. Similarly, the pseudo-random number in the BSD operating system is based on a former version of the Fortuna generator and shall be assessed in a strong security model. Open-source security products, or open-source cryptographic libraries shall also be assessed, as they are widely used in practice. Recently, a vulnerability has been discovered in the Truecrypt software, related to the improper initialisation of the pseudo-random number generator [Tru15]. A careful assessment of this generator in a strong security model would allow to point out new potential weaknesses or to ensure security.

Implementations. For the implementations of our robust construction, we used the RELIC open source library [AG], and the PolarSSL open source library [Pol]. In an industrial perspective, one could propose an optimized implementation of our robust construction for operating systems or security applications.

List of Figures

List of Algorithms

List of Tables

Bibliography

[ABF13] Michel Abdalla, Sonia Belaïd, and Pierre-Alain Fouque. Leakage-resilient symmetric encryption via re-keying. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 471–488, Santa Barbara, California, US, August 20–23, 2013. Springer, Berlin, Germany. 4, 44, 45, 105

[ABP+15] Michel Abdalla, Sonia Belaïd, David Pointcheval, Sylvain Ruhault, and Damien Vergnaud. Robust Pseudo-Random Number Generators with Input Secure Against Side-Channel Attacks - Extended Version. Cryptology ePrint Archive, 2015. 6, 91

[AG] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/. 73, 109, 135

[AGV09] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 474–495. Springer, Berlin, Germany, March 15–17, 2009. 6, 44

[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. 5

[And13] Some SecureRandom Thoughts, Aug 14st, 2013, 2013. http://android-developers.blogspot.fr/2013/08/some-securerandom-thoughts.html. 127

[ANS85] ANSI X9.17 (revised). American National Standard for Financial Institution Key Management (Wholesale), American Bankers Association, 1985. 27, 36

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 2001. 4

[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. 93, 106

[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. v, 2, 3, 5, 6, 18, 23, 28, 36, 39, 40, 41, 42, 43, 44, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 113, 115, 137

[BK96] Burton and Kaliski. Ieee p1363: A standard for rsa, diffie-hellman, and elliptic-curve cryptography (abstract). In T. Mark A. Lomas, editor, *Security Protocols Workshop*, volume 1189 of *Lecture Notes in Computer Science*, pages 117–118. Springer, 1996. 128

[BK12] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A, 2012. 3, 69

[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, Berlin, Germany. 23

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371, Seoul, South Korea, December 4–8, 2011. Springer, Berlin, Germany. 109

[Bou] The bouncy castle crypto package is a java implementation of cryptographic algorithms. http://www.bouncycastle.org/. 131

[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, Berlin, Germany. 9, 11

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Berlin, Germany. 4

[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, Berlin, Germany. v, 2, 3, 18, 22, 28, 36, 37, 38, 39, 41, 43, 48, 49, 51, 52

[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, Berlin, Germany. v, 3, 22, 28, 31, 32, 33, 43, 47, 48, 49, 137

[CG85] Benny Chor and Oded Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity (extended abstract). In *26th Annual Symposium on Foundations of Computer Science*, pages 429–442, Portland, Oregon, October 21–23, 1985. IEEE Computer Society Press. 12

[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. 6, 7, 77, 78

[CVE08] CVE-2008-0166. CVE, 2008. 5

[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. 5

[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, Berlin, Germany. v, 3, 22, 27, 34, 35, 36, 37, 38, 43, 47, 48, 49, 51, 69, 113, 137

[DKL09] Yevgeniy Dodis, Yael Tauman Kalai, and Shachar Lovett. On cryptography with auxiliary input. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 621–630, Bethesda, Maryland, USA, May 31 – June 2, 2009. ACM Press. 44

[DLW06] Giovanni Di Crescenzo, Richard J. Lipton, and Shabsi Walfish. Perfectly secure password protocols in the bounded retrieval model. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 225–244, New York, NY, USA, March 4–7, 2006. Springer, Berlin, Germany. 4

[DP08] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *49th Annual Symposium on Foundations of Computer Science*, pages 293–302, Philadelphia, Pennsylvania, USA, October 25–28, 2008. IEEE Computer Society Press. 4, 44

[DP10] Yevgeniy Dodis and Krzysztof Pietrzak. Leakage-resilient pseudorandom functions and side-channel attacks on Feistel networks. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 21–40, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Berlin, Germany. 4

[DPR+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. 5, 6, 7, 51, 55, 108, 109, 110, 135, 137, 138

[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, Berlin, Germany. 18

[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. 27, 36, 129, 130, 132

[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, Berlin, Germany. 135

[Dzi06] Stefan Dziembowski. Intrusion-resilience via the bounded-storage model. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 207–224, New York, NY, USA, March 4–7, 2006. Springer, Berlin, Germany. 4

[ESC05] D. Eastlake, J. Schiller, and S. Crocker. *RFC 4086 - Randomness Requirements for Security*, June 2005. 3

[EYP10] Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*, pages 633–658. 2010. 4

[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, Berlin, Germany. 4, 44, 45, 46, 103, 105, 106, 109, 110, 137, 138

[FPZ08] Pierre-Alain Fouque, David Pointcheval, and Sébastien Zimmer. HMAC is a randomness extractor and applications to TLS. In Masayuki Abe and Virgil Gligor, editors, *ASIACCS 08: 3rd Conference on Computer and Communications Security*, pages 21–32, Tokyo, Japan, March 18–20, 2008. ACM Press. 114

[FSK10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010. 135

[GB01] Shafi Goldwasser and Mihir Bellare. Lecture notes on cryptography, 2001. 9, 25, 114

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986. 23

[GLSV12] François Goichon, Cédric Lauradoux, Guillaume Salagnac, and Thibaut Vuillemin. Entropy transfers in the Linux Random Number Generator. Rapport de recherche RR-8060, INRIA, September 2012. 119

[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, California, USA, May 21–24, 2006. IEEE Computer Society Press. 5, 115, 119

[Gut98] Peter Gutmann. Software generation of practically strong random numbers. Proceedings of the 7th USENIX Security Symposium. Full version available at `http://www.cypherpunks.to/ peter/06_random.pdf`, 1998. v, 3, 5, 22, 27, 28, 29, 30, 32, 38, 43, 47, 48, 49, 125

[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 Security Symposium*, August 2012. 5

[Hea] The Heartbleed Bug, 2014. http://heartbleed.com. 4

[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. 19

[IBM14] Recent Fixes in IBM SecureRandom, 2014. http://www.cigital.com/justice-league-blog/2014/05/06/recent-fixes-ibmsecurerandom/. 133

[Imp95] Russell Impagliazzo. A Personal View of Average-Case Complexity. In *Structure in Complexity Theory Conference*, pages 134–147, 1995. 104

[ISO11] Information technology - Security techniques - Random bit generation. ISO/IEC18031:2011, 2011. 3

[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. 5

[Kil11] Killmann, W. and Schindler, W. A proposal for: Functionality classes for random number generators. AIS 20 / AIS31, 2011. 3

[Koo02] Philip Koopman. 32-bit cyclic redundancy codes for internet applications. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 459–472, Washington, DC, USA, 2002. IEEE Computer Society. 120

[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, Berlin, Germany. v, 3, 5, 22, 27, 28, 29, 30, 32, 38, 43, 47, 48, 49, 129, 132

[LHA+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, Berlin, Germany. 5

[LRSV12] Patrick Lacharme, Andrea Rock, Vincent Strubel, and Marion Videau. The linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251, 2012. 115, 119, 124

[Luc00] Stefan Lucks. The sum of PRPs is a secure PRF. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 470–484, Bruges, Belgium, May 14–18, 2000. Springer, Berlin, Germany. 111

[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, Berlin, Germany. 5, 114, 127, 131

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards.* Springer, 2007. 93

[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, Berlin, Germany. 44

[Net96] How secure is the World Wide Web?, , 1996. http://www.cs.berkeley.edu/ daw/papers/ddj-netscape.html. 5

[NS02] Phong Q. Nguyen and Igor Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, 2002. 5

[NZ93] Noam Nisan and David Zuckerman. More deterministic simulation in logspace. In *25th Annual ACM Symposium on Theory of Computing*, pages 235–244, San Diego, California, USA, May 16–18, 1993. ACM Press. 13

[Ope13] OpenSSL PRNG Is Not (Really) Fork-safe, Aug 21st, 2013. http://emboss.github.io/blog/2013/08/21/openssl-prng-is-not-really-fork-safe/. 126

[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, Berlin, Germany. 4, 44

[Pol] PolarSSL is an open source and commercial SSL library licensed by Offspark B.V. https://polarssl.org. 73, 109, 135

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159, Athens, Greece, May 26–30, 2013. Springer, Berlin, Germany. 4, 44

[Pul] The emergence of complexity in mathematics, physics, chemistry and biology. Proceedings of the plenary session of the Pontifical Academy of Sciences, Vatican City, Italy, October 27-31 1992. Edited by Bernard Pullman. iv, 137

[SEC13] SecurityTracker Alert ID: 1028916. SecurityTracker, 2013. 5

[Sha48] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27, 1948. 12

[SHA95] Secure hash standard. National Institute of Standards and Technology, NIST FIPS PUB 180-1, U.S. Department of Commerce, April 1995. 113, 128, 129

[Sha02] Ronen Shaltiel. Recent developments in explicit constructions of extractors. *Bulletin of the EATCS*, 77:67–95, 2002. 12

[Sho06] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, 2006. 9, 10, 16

[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, Berlin, Germany. 4

[SV84] Miklos Santha and Umesh V. Vazirani. Generating quasi-random sequences from slightly-random sources (extended abstract). In *25th Annual Symposium on Foundations of Computer Science*, pages 434–440, Singer Island, Florida, October 24–26, 1984. IEEE Computer Society Press. 13, 14

[SV03] Amit Sahai and Salil P. Vadhan. A complete problem for statistical zero knowledge. *J. ACM*, 50(2):196–249, 2003. 53

[Tru15] Open Crypto Audit Project, 2015. https://opencryptoaudit.org. 135

[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, California, USA, November 12–14, 2000. IEEE Computer Society Press. 18

[Vad12] Salil Vadhan. Pseudorandomness, draft survey monograph, 2012. 9, 16, 20

[vdVdSCB12] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 86–106, Berlin, Heidelberg, 2012. Springer-Verlag. 4

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 282–296, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Berlin, Germany. 109

[VN51] John Von Neumann. 13. various techniques used in connection with random digits. 1951. 13

[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, Berlin, Germany. 4, 44, 45, 46, 47, 103, 104, 105, 107, 109, 137, 138

[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. 4, 44, 45, 46, 137

Abstract

In cryptography, randomness plays an important role in multiple applications. It is required in fundamental tasks such as key generation and initialization vectors generation or in key exchange. The security of these cryptographic algorithms and protocols relies on a source of unbiased and uniform distributed random bits. Cryptography practitioners usually assume that parties have access to perfect randomness. However, quite often this assumption is not realizable in practice and random bits are generated by a Pseudo-Random Number Generator. When this is done, the security of the scheme depends of course in a crucial way on the quality of the (pseudo-)randomness generated. However, only few generators used in practice have been analyzed and therefore practitioners and end users cannot easily assess their real security level.
We provide in this thesis security models for the assessment of pseudo-random number generators and we propose secure constructions. In particular, we propose a new definition of robustness and we extend it to capture memory attacks and side-channel attacks. On a practical side, we provide a security assessment of generators used in practice, embedded in system kernel (Linux `/dev/random`) and cryptographic libraries (OpenSSL and Java `SecureRandom`), and we prove that these generators contain potential vulnerabilities.

Keywords: pseudo-random number generators, security models, robustness, memory attacks, side-channel attacks, Linux `/dev/random`, OpenSSL, Java `SecureRandom`.

Résumé

La génération d'aléa joue un rôle fondamental en cryptographie et en sécurité. Des nombres aléatoires sont nécessaires pour la production de clés cryptographiques ou de vecteurs d'initialisation et permettent également d'assurer que des protocoles d'échange de clé atteignent un niveau de sécurité satisfaisant. Dans la pratique, les bits aléatoires sont générés par un processus de génération de nombre dit pseudo-aléatoire, et dans ce cas, la sécurité finale du système dépend de manière cruciale de la qualité des bits produits par le générateur. Malgré cela, les générateurs utilisés en pratique ne disposent pas ou peu d'analyse de sécurité permettant aux utilisateurs de connaître exactement leur niveau de fiabilité.
Nous fournissons dans cette thèse des modèles de sécurité pour cette analyse et nous proposons des constructions prouvées sûres et efficaces qui répondront à des besoins de sécurité forts. Nous proposons notamment une nouvelle notion de robustesse et nous étendons cette propriété afin d'adresser les attaques sur la mémoire et les attaques par canaux cachés. Sur le plan pratique, nous effectuons une analyse de sécurité des générateurs utilisés dans la pratique, fournis de manière native dans les systèmes d'exploitation (/dev/random sur Linux) et dans les librairies cryptographiques (OpenSSL ou Java `SecureRandom`) et nous montrons que ces générateurs contiennent des vulnérabilités potentielles.

Mots clés : générateurs de nombres pseudo-aléatoires, modèles de sécurité, robustesse, attaques contre la mémoire, attaques par canaux cachés, Linux `/dev/random`, OpenSSL, Java `SecureRandom`.