Chapter 2
Random Number Generators for Cryptographic Applications

Werner Schindler

2.1 Introduction

A large number of cryptographic applications require random numbers, e.g., as session keys, signature parameters, ephemeral keys (DSA, ECDSA), challenges or in zero-knowledge protocols. For this reason, random number generators (RNGs) are part of many IT-security products. Inappropriate RNGs may totally weaken IT systems that are principally strong, e.g., if an adversary is able to determine session keys.

It is intuitively clear that random numbers should remain unpredictable, even if an adversary knows a large number of other random numbers (predecessors or successors of the random numbers of interest) that have been generated with the same RNG, e.g., from openly transmitted challenges or session keys from messages that the adversary has received legitimately. Ideally, random numbers should be uniformly distributed on their range and independent. However, this characterizes an *ideal RNG*, which is a mathematical construction.

In Section 2.2 we formulate the general requirements RNGs should have, and in Section 2.3 we divide the entity of 'real-world' RNGs into several classes. The main classes are deterministic RNGs and true RNGs, the latter falling into two subclasses (physical and non-physical true RNGs).

The designer of an RNG is faced with two challenges. First he has to develop an appropriate design and implement it suitably. Especially for true RNGs the second task is usually even more difficult, namely to prove or at least to give strong evidence that the chosen design and the concrete implementation are indeed secure.

The main part of this chapter is devoted to deterministic RNGs (Section 2.4). The basic aspects of true RNGs are addressed in Sections 2.5 and 2.6. Evaluation criteria for physical RNGs are treated intensively in the following chapter. Section 2.7

Bundesamt für Sicherheit in der Informationstechnik
e-mail: Werner.Schindler@bsi.bund.de

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_2,

addresses important standards and evaluation guidances for RNGs. Sections 2.8 and 2.9 contain exercises and possible implementation projects.

2.2 General Requirements

Many cryptographic applications require random numbers. The protocol usually only demands 'generate a 64-bit challenge', 'generate a random prime', 'generate a random session key' etc., but does not specify any requirements these random values should have. Intuitively, the matter seems to be clear: Random numbers should assume all possible values with equal probability and should be independent from predecessors and successors. However, these (usually unspoken) requirements are very restrictive and characterize an ideal RNG. Note that even if a real-world RNG was ideal it is hardly possible to give evidence in a strict sense (cf. the next chapter).

A closer look at typical applications allows a positive formulation of necessary requirements. Absolutely inevitable is

- (R1) The random numbers should have good statistical properties.

Requirement (R1) is usually checked with a particular statistical test suite, ideally adjusted to the concrete RNG. For specific applications, as for many challenge–response protocols or openly transmitted IVs for block ciphers in CBC mode, (R1) should be fully sufficient. In particular, (R1) shall exclude replay attacks or correlation based attacks.

Unfortunately, (R1) is insufficient for sensitive applications. In Section 2.4.3 we will treat RNGs that have good statistical properties but allow an adversary to predict the whole sequence of random numbers from a small, known subsequence. The assumption that an adversary knows some random numbers is realistic for many applications. Consider, for instance, the generation of session keys if the same RNG is also used for challenges that are transmitted openly. Another example is a classical hybrid protocol where Alice encrypts a confidential message with a randomly selected session key k_{rnd} and sends k_{rnd} to the legitimate receiver, using a suitable key exchange protocol. Of course, the legitimate receiver of particular messages shall not be able to decrypt other messages. In this context, a legitimate receiver of a message is principally a privileged attacker since he knows at least one session key. If Alice represents a public server an adversary may learn millions of random numbers. This suggests the next requirement, namely.

- (R2) The knowledge of subsequences of random numbers shall not allow one to *practically* compute predecessors or successors or to guess these numbers with non-negligibly larger probability than without knowledge of these subsequences.

In Section 2.4 we will introduce two further requirements that are characteristic for DRNGs.

2.3 Classification

Following [1] (which narrows the focus to random bit generators) 'real-world' RNGs fall into two main classes. The first class consists of the *deterministic RNGs* (DRNGs, aka *pseudorandom number generators*). Starting with a *seed*, DRNGs generate pseudorandom numbers algorithmically. The *true RNGs* (TRNGs) form the second class, which falls into two subclasses: *physical TRNGs* (PTRNGs) and *non-physical TRNGs* (NPTRNGs). Physical TRNGs use non-deterministic effects of electronic circuits (e.g., shot noise from Zener diode, inherent semiconductor thermal noise, free-running oscillators) or physical experiments (e.g., time between emissions of radioactive decay, quantum random processes). NPTRNGs exploit non-deterministic events (e.g., system time, hard disk seek time, RAM content, user interaction). So-called *hybrid RNGs* have design elements from both DRNGs and TRNGs. Roughly speaking, the security of a DRNG essentially depends on the computational complexity of possible attacks (→ practical security), while TRNGs rely on the unpredictability of their output (→ theoretical security). We will illuminate this aspect later. Depending on their main 'security anchor' we distinguish between *hybrid DRNGs* and *hybrid* TRNGs (Figure 2.1).

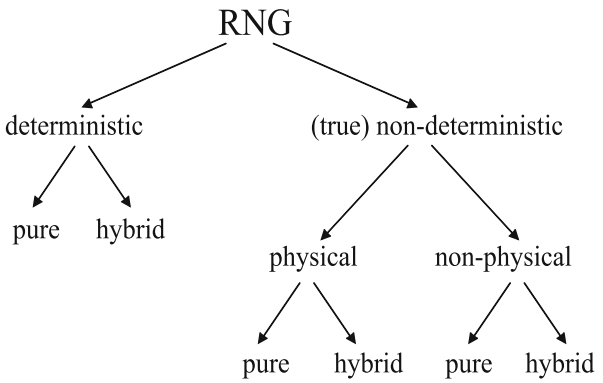

Fig. 2.1 RNG classification.

2.4 Deterministic Random Number Generators (DRNGs)

In this section we consider deterministic random number generators. The main part of this section deals with pure DRNGs but we also consider hybrid DRNGs. We formulate and justify two additional DRNG-specific requirements (R3) and (R4). We illustrate the general principles by many examples, and we also address stochastic simulations and Monte Carlo integration.

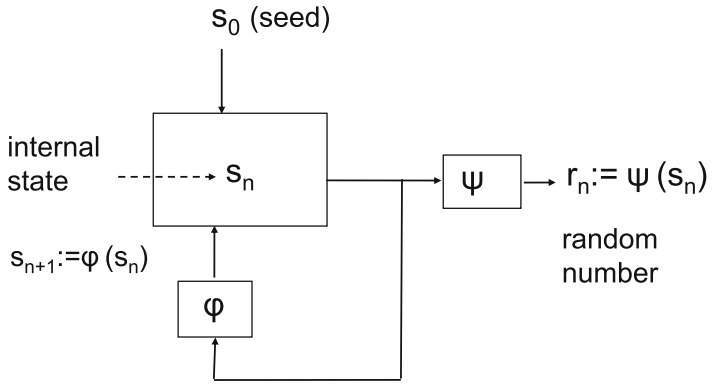

Fig. 2.2 Pure DRNG: Generic design.

2.4.1 Pure DRNGs

This subsection considers the generic design and basic properties of pure DRNGs, and we analyze the security properties of several pure DRNG designs. Figure 2.2 illustrates the generic design of a *pure* DRNG. After $n-1$ random numbers

$$r_1, r_2, \ldots, r_{n-1} \in R$$

have been generated, the internal state of the DRNG attains the value $s_n \in S$. The finite sets S and R are called the *state space* and the *output space* of the DRNG. The *output transition function* $\psi\colon S \to R$ computes the next random number r_n from the current internal state s_n. Then s_n is updated to s_{n+1} with the *state transition function* ϕ, i.e., $s_{n+1} := \phi(s_n)$. The first internal state s_1 is derived from the *seed* s_0, e.g., simply $s_1 = \phi(s_0)$, or a more complicated mechanism may be used. Clearly, the seed s_0 determines all internal states $s_1, s_2, \ldots$ and all random numbers $r_1, r_2, \ldots$. In order to fulfil requirement (R2) the seed must be selected randomly. A pure DRNG can be described by a 5-tuple

$$(S, R, \phi, \psi, p_S) \tag{2.1}$$

where p_S defines the probability distribution of the random seed. Note, however, that the seed generation is performed outside the DRNG boundaries. Usually the seed is generated by a TRNG.

A drawback of DRNGs (compared to TRNGs) is that the output is completely determined by the seed, and the future random numbers depend only on the current internal state. Thus the internal state must be protected even if the device is not active. In particular, implementing a pure DRNG on a PC and using its current internal state in the next session may be dangerous. Typically, DRNGs are implemented on smart cards. Of course, pseudorandom numbers cannot be truly random. On the positive side implementing a DRNG is relatively cheap, and unlike for physical RNGs, no dedicated hardware is needed.

We point out that (R2) demands that the seed entropy must be 'large' and that the state transition function and the output function are sufficiently complex. Pure

DRNGs can at most provide *practical security* (computational security). In an information theoretical sense already a few random numbers fully determine the seed and all the generated random numbers completely. In this regard the situation is similar to that of cryptographic primitives (e.g., to block ciphers). In fact, DRNGs typically apply cryptographic primitives.

Example 2.1. Consider a linear feedback shift register (LFSR) over GF(2) with t cells and recursion formula $a_{n+t+1} \equiv c_1 a_{n+t} + \ldots c_n a_{n+1} \pmod 2$ with $c_1, \ldots, c_n \in \{0,1\}$. In this example, $S = \{0,1\}^t$, $s_n := (a_n, \ldots, a_{n+t-1})$,

$$s_{n+1} = \phi(s_n) = (a_{n+1}, \ldots, a_{n+t}) \ ,$$

$R = \{0,1\}$, $r_n = \psi(s_n) = a_n$.

For primitive feedback polynomials the output sequence $r_1, r_2, \ldots$ is known to have good statistical properties unless t is too small. Hence this DRNG should fulfil (R1). Moreover, LFSRs can be implemented efficiently, and they are very fast. On the other hand, the random numbers $r_1, r_2, \ldots$ depend GF(2)-linearly on the initial state of the LFSR. If an adversary knows about t output bits he can easily recover s_1 and hence the whole sequence $r_1, r_2, \ldots$. Consequently, LFSRs do not fulfil requirement (R2), and they are absolutely inappropriate for sensitive cryptographic applications.

Example 2.2. Assume that Enc: $\{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$ defines a block cipher where $\{0,1\}^n$ and $\{0,1\}^m$ denote the plaintext space (respectively, the ciphertext space and the key space). Here $S = \{0,1\}^n \times \{0,1\}^m$ and $R = \{0,1\}^n$. Further, $s_n = (r_n, k)$ where the key k has to be kept secret. Finally, $\psi(r_n, k) = r_n$ and $s_{n+1} = (\mathrm{Enc}(r_n, k), k)$ for $n \geq 0$. For commonly used block ciphers no statistical weaknesses are known, and hence (R1) should be fulfilled.

Now assume that the adversary knows random numbers $r_i, \ldots, r_{i+j}$. Since $r_{t+1} = \mathrm{Enc}(r_t, k)$ the adversary knows $j-1$ (specific) (plaintext ciphertext) pairs. Hence finding r_{i-1} is at least as difficult as a chosen-input attack on Enc. (Actually, the situation is even close to a known plaintext attack.) Analogously, the computation of r_{i+j+1} cannot be easier than a chosen plaintext attack on the decryption function Enc^{-1}. Against strong block ciphers chosen-plaintext attacks on Enc and Enc^{-1} are not practically feasible. (Otherwise these algorithms would not be viewed as secure.) Provided that the seeding process guarantees that k cannot be guessed with non-negligible probability, we may assume that the DRNG fulfils (R2) if Enc=AES or Enc =Triple-DES, for instance.

Example 2.2 demonstrates a typical security proof for DRNGs where security properties are traced back to properties of well-studied primitives. Note that requirement (R2) is not fulfilled for Enc =DES, although in the eighties one would presumably have confirmed this property. This underlines another important property of DRNGs, namely that their assessment may change in the course of time.

Assume that an attacker gets knowledge of the current internal state s_n, e.g., because he has mounted a successful hardware attack on a smart card or has had interim access to a computer where this DRNG is implemented. Of course, then the

random numbers $r_n, r_{n+1}, \ldots$ follow immediately from s_n. For many applications it is desirable that the DRNG additionally meets

- (R3) The knowledge of the internal state shall not allow one to *practically* compute 'old' random numbers or even a previous internal state or to guess these values with non-negligibly larger probability than without knowledge of the internal state.

Requirement (R3) demands one-way state transition functions $\phi : S \to S$. We note that Example 2.2 does not fulfil (R3). Once an adversary knows k he simply decrypts $r_{n-1} = \mathrm{Enc}^{-1}(r_n, k)$, $r_{n-2} = \mathrm{Enc}^{-1}(r_{n-1}, k), \ldots$.

Example 2.3. Let $S = R = \{0,1\}^{160}$ while ϕ and ψ are given by the hash functions SHA-1 and RIPEMD-160. At this time both SHA-1 and RIPEMD-160 are assumed to meet the one-way property. As a consequence, this RNG meets (R3) (as well as (R1) and (R2)).

The next example underlines that not only ϕ and ψ are relevant but also their interaction.

Example 2.4. [weak RNG] Let $S = R = \{0,1\}^{256}$ and $\phi = \psi =$ SHA-256. Obviously, $s_{n+1} = \phi(s_n) = \psi(s_n) = r_n$, and $r_{n+1}, r_{n+2}, \ldots$ follow from r_n. In other words, this RNG does not meet (R2).

Example 2.5. Appendix 3.2 in [2] specifies the generation of pseudorandom ephemeral keys. The 'core' of this algorithm defines a DRNG: $w_0 := f(s_n)$,

$$s' := (1 + s_n + w_0) (\mathrm{mod}\ 2^v) \ ,$$

$w_1 := f(s')$, $s_{n+1} := (1 + s' + w_1) (\mathrm{mod}\ 2^v)$, $r_n := (w_1, w_0)$ with a one-way function f which is defined in [2] (cf. Exercise 2).

We mention that occasionally even DRNGs proposed by adopted standards may contain security flaws. Bleichenbacher detected a weakness in the random number generation specified in the preceding version of [2, 3] which led to a change notice [33]. The problem was the following: Uniformly distributed random numbers r_n on $Z_{2^{160}} := \{0, 1, \ldots, 2^{160} - 1\}$ were transformed to random numbers on Z_p by computing $r_n (\mathrm{mod}\ p)$ where p denotes a 160-bit prime. Obviously, the small values in Z_p occur twice as often as the large ones. Although the weakness itself is obvious, the attack is not. Interestingly, a full paper that describes the attack in detail has never been published. Reference [4] shows that the DRNG that is used by Windows 2000 does not meet requirement (R3) (see also Section 2.6).

Remark 2.1. Many applications provide implicit information on the generated random numbers, e.g., by known (plaintext/ciphertext) pairs that correspond to a random session key. Sometimes less complicated formulae also exist that contain information on the unknown random numbers. For DSA- and ECDSA signatures, for instance, the adversary knows an underdetermined system of linear equations in the

signature key and the ephemeral keys. We point out that this aspect may be more relevant for PTRNGs that aim at security in an information theoretical sense. We will come back to this issue in the next chapter.

A class of DRNGs which is very interesting from a theoretical point of view are *cryptographically secure* RNGs. Their security relies upon intractability assumptions (e.g., that factoring large integers is hard). On the basis of this intractability assumption(s), security properties of the DRNGs, in respect of the random numbers can be proved.

Unfortunately, the security assertions concern the whole family of DRNGs, and in a strict sense, it is usually not clear what this means for a concrete member of this family, i.e., for a fixed DRNG. In this regard, the situation reminds us of DRNGs that rely on the security of (concrete) block ciphers or hash functions (where, not even an asymptotic security proof exists). A drawback of cryptographically secure DRNGs is their low output rate.

Example 2.6. (Blum-Blum-Shub DRNG) Let $n = p_1 p_2$ for two m-bit primes p_1 and p_2 with $p_i \equiv 3 (\mathrm{mod}\, 4)$. Starting with a quadratic residue $x_0 \in Z_n^* := \{0 \leq j < n \mid gcd(j,n) = 1\}$ (seed) we compute $x_{n+1} \equiv x_n^d (\mathrm{mod}\, n)$ and $r_n := x_n (\mathrm{mod}\, 2^{t(m)})$.

The generation of $t(m)$ random bits requires the modular exponentiation of a $2m$-bit integer. It is known that a Blum-Blum-Shub DRNG – or more precisely, a family of Blum-Blum-Shub DRNGs – is asymptotically secure if $t(m) = O(\log \log m)$. Roughly speaking, a non-negligible advantage in guessing the next bit (compared to 'blind guessing') enabled an efficient factoring algorithm, contradicting the factoring intractability.

We mention also that RSA- and Rabin RNGs belong to the class of cryptographically secure RNGs (see [5], Section 5.5, and [6], for instance). We leave this field and refer the interested reader to the relevant literature.

2.4.2 Hybrid DRNGs

Pure DRNGs compute $r_n := \psi(s_n)$ and update their internal state by $s_n \to \phi(s_n)$. Hybrid DRNGs allow additional input from a finite set E_0. The state transition function then reads $\phi_H : S \times E \to S$ with $E = E_0 \cup \{\infty\}$ where ∞ means 'no additional input'. Formally, any pure DRNG can be viewed as a hybrid DRNG with additional input ∞ in each step, i.e., $E = \{\infty\}$ (Figure 2.3).

Example 2.7. Consider Example 2.2 with $E_0 = \{0,1\}^n$ and

$$\phi_H((r_n,k),e_{n+1}) = (\mathrm{Enc}(r_n \oplus e_{n+1}),k)$$

for $e_{n+1} \in E_0$ and $\phi_H((r_n,k),\infty) = (\mathrm{Enc}(r_n),k) = \phi(s_n)$, and $\psi_H = \psi$. (As usual $\oplus$ stands for the bitwise addition modulo 2.)

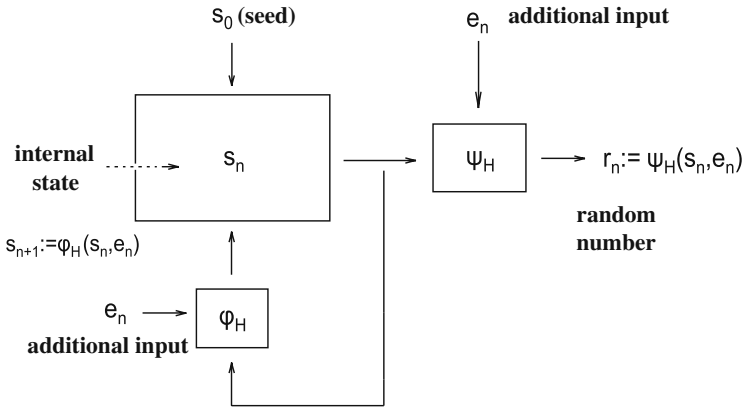

Fig. 2.3 Hybrid DRNG: Generic design.

Of course, a hybrid DRNG cannot be described by a 5-tuple (S,R,ϕ,ψ,p_S) (cf. (2.1)). Instead, we use a 7-tuple

$$(S,R,E,\phi_H,\psi_H,p_S,(q_n)_{n\in N}). \tag{2.2}$$

The set E and the sequence $(q_n)_{n\in N}$ denote the set of additional input data and the probability distributions of the additional data. Note that if $E=\{\infty\}$ and $q_n=\varepsilon_\infty$ (Dirac measure, which has its total mass concentrated on ∞) for all $n\in N$ ('never any additional input') the 7-tuple describes a pure DRNG.

Clearly, even if the input sequence $e_1,e_2,\ldots$ is constant or completely known by a potential attacker this does not reduce the security of the hybrid DRNG below the security of the respective pure DRNG from Example 2.2. Whether the additional input actually increases the security depends on its randomness and unpredictability properties. If the additional input is derived from the current time, for instance, the security gain may be small, depending on the knowledge of the attacker.

For certain applications the following property is desirable, namely when an attacker gets knowledge of the current internal state of the DRNG (e.g., of a software DRNG implementation on a PC) without being noticed by the user of this DRNG, which generates further random numbers.

- (R4) Even the knowledge of the internal state shall not allow one to *practically* compute the next random numbers or to guess these values with non-negligibly larger probability than without the knowledge of the internal state.

Of course, pure DRNGs cannot fulfil (R4). Whether (R4) is met depends on the randomness of the additional input. Regular additional input from a strong TRNG clearly implies (R4). We will learn more about TRNGs in Sections 2.5 and 2.6 and in the next chapter.

In Example 2.7 we *updated* the internal state before applying the seed transition function ϕ. In fact, ϕ_H may be viewed as a two-step procedure. We note that in the

first step the mapping $r_n \to r_n \oplus e_{n+1}$ is injective for any fixed e_{n+1}. This has the pleasant consequence that the security of the hybrid DRNG cannot drop below the level of the respective pure DRNG, regardless of the nature of the additional input and the adversary's knowledge of this input. Much more critical was *reseeding*, realized, for example, by $\phi'_H((r_n,k),e'_{n+1}) = \mathrm{Enc}(r_n,e'_{n+1})$ with $e'_{n+1} \in E' = \{0,1\}^m$. For reseeding, the unpredictability of the additional input is absolutely inevitable.

Remark 2.2. (i) Requirements (R3) and (R4) are specific DRNG requirements. For TRNGs, (R3) and (R4) are usually 'automatically' fulfilled if (R2) is valid.
(ii) In some scenarios the designer may not be able to specify the distributions of the additional input data. Then no security value can be assigned to the additional input. For a seed-update in a strict sense only the security level of the respective DRNG can be assured (provided that $\psi_H(\cdot,e)$ is injective for each $e \in E$). In case of reseeding, no security assertions are yet possible.

Example 2.8. ANSI X9.17 DRNG (hybrid DRNG)
Let $s_n = (r_n,k)$ where k denotes a Triple-DES key while the additional input t_n is a 64-bit representation of the current time. Compute $e_n := \text{Triple-DES}(t_n;k)$, $r_n := \text{Triple-DES}(s_n \oplus e_n;k)$, $s_{n+1} := \text{Triple-DES}(r_n;k)$. This DRNG does not fulfil (R3) if the adversary knows the exact times when random numbers are generated, i.e., if an adversary knows all values of t_n (Exercise 4).

We point out that [1], Annex C, provides several more complex examples of strong DRNGs which use block ciphers, hash functions, or elliptic curves. Reference [1] defines several security levels, demanding different parameter sets.

2.4.3 A Word of Warning

Pseudorandom numbers are used in several branches of applied mathematics, e.g., for stochastic simulations or Monte Carlo integrations. For these applications only statistical properties are significant (cf. [7], for instance) while the unpredictability of pseudorandom numbers is irrelevant. Consequently, pseudorandom numbers that are suitable for stochastic simulations or Monte Carlo integrations usually are completely inappropriate for sensitive cryptographic applications. However, the identical terminology occasionally confuses designers of cryptosystems who have only limited experience with RNGs.

Linear congruential random number generators are widespread since they are extremely fast. Assume, for example, that

$$s_{n+1} \equiv as_n + 1 \pmod{2^m} \quad \text{with } a \equiv 1 \pmod 4. \tag{2.3}$$

Then $x_n := s_n/2^m \in [0,1)$, $n = 1,2,\ldots$ gives a sequence of so-called standard random numbers which are assumed to have similar statistical properties as values taken on by independent random variables $X_1, X_2, \ldots$ that are uniformly distributed on the

unit interval. The sequence $s_0, s_1, \ldots$ is periodic with maximum length 2^m. The parameter $m = 64$ fits perfectly to 64-bit computer architectures. Of course, from s_n a potential attacker can easily determine the whole sequence $x_1, x_2, \ldots$ of pseudo-random numbers. Moreover, the k least significant bit of s_n has period length 2^k. This does not play a role for typical applications of stochastic simulations since there only the most significant bits of the real numbers $x_1, x_2, \ldots$ are relevant. For the generation of secret data short periods of particular bits are not tolerable. Even more, in specific applications short periods might enable correlation attacks. Linear congruential generators do not even meet the basic requirement (R1).

We point out that linear congruential generators can be strengthened in the sense of unpredictablity at the cost of throughput, namely by outputting, let's say, only the m–k most significant bits of s_n, i.e., returning $x_n := (s_n >> k)/2^{m-k}$. However, Knuth found a recovery attack that requires $O(2^{2k} m^2 t^{-2})$ operations if the adversary knows t random numbers $x_1, \ldots, x_t$ ([8]). We mention also that other moduli than powers of 2 and generalizations of linear congruential generators have been studied. We do not delve into this aspect here. We strictly recommend not to use linear congruential generators or related designs for sensitive cryptographic applications.

2.5 Physical True Random Number Generators (PTRNGs)

In this section we explain the generic design of *Physical TRNGs* (PTRNGs) and their important properties. Moreover, we address the concept of entropy and workload. For a thorough treatment of evaluation aspects for PTRNGs we refer the interested reader to the next chapter.

2.5.1 The Generic Design

Just as DRNGs, PTRNGs also are typically implemented in smart cards. Figure 2.4 illustrates the generic design of a physical RNG. The 'core' is the noise source, typically realized by electronic circuits (e.g., using noisy diodes or free-running oscillators) or by physical experiments (radioactive decay, quantum effects of photons, etc.) The noise source generates time-continuous analog signals which are (at least for electronic circuits typically) periodically digitized to binary values at some stage. We call the digitized values *digitized analog signals* or briefly *das random number numbers*. If the das random numbers are binary-valued we also speak of *das bits*. The das random numbers may be algorithmically postprocessed to internal random numbers in order to reduce potential weaknesses. Note that reducing weaknesses (and not simply transforming them into others, (e.g., bias into dependencies) requires data compression which in turn lowers the output rate of the RNG. The algorithmic postprocessing may be memoryless, i.e., it may only depend on the

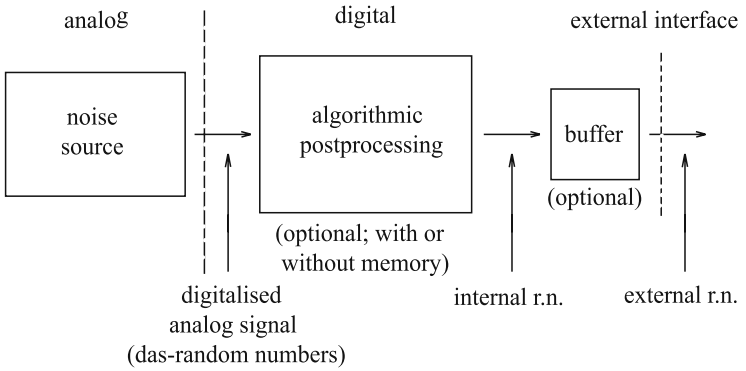

Fig. 2.4 Generic design of a physical RNG.

current das bits, or it may combine the current das random numbers with memory values that depend on the preceding das random numbers (and maybe on some other, possibly secret parameters). Note that strong noise sources do not necessarily require algorithmic postprocessing. Upon external request the internal random numbers are output.

The security of a pure DRNG essentially depends on two factors: on the expected number of guesses to find the seed or any internal state of the DRNG with non-negligible probability and on the complexity of the state transition function and the output function, which determine the workload of possible attacks. Typically, the basic components of DRNGs are cryptographic primitives, and the security of the DRNG can usually be traced back to well-known properties of these primitives (cf. Example 2.2). Clearly, DRNGs can at most be *practically secure*, and their assessment changes in the course of time when attacks on the primitives have become feasible. Consider, for instance, Enc=FEAL-8, which was introduced in the late eighties. Some years later Biham showed that only 2^{24} known-plaintexts are sufficient to recover the 64-bit key. Another prominent example is the change notice in [2] in response to Bleichenbacher's attack.

Contrary to DRNGs, TRNGs (physical and non-physical) rely on the unpredictablity of the generated random numbers. In Section 2.5.2 we will learn that this question is closely related to entropy. Provided, of course, that the entropy estimates are correct, the expected workload to guess such random numbers remains invariant over time. Consequently, it is reasonable to use TRNGs at least for the generation of random numbers that shall protect secrets in the long term. Theoretical security bounds, quantified by the expected number of guesses to find (a sequence of) random numbers, can only be achieved by TRNGs. Unlike for DRNGs, this number does not decrease in the course of time unless, of course, the evaluator made a mistake when estimating the entropy per random bit. We will treat the evaluation of PTRNGs in the next chapter. The following subsection introduces the notion of entropy and guesswork.

2.5.2 Entropy and Guesswork

For the remainder of this subsection, X denotes a random variable that assumes values in the finite set $\Omega = \{\omega_1, \ldots, \omega_m\}$. Without loss of generality, we may assume that $p(\omega_1) := \text{Prob}(X = \omega_1) \geq \cdots \geq p(\omega_m) := \text{Prob}(X = \omega_m)$. The most efficient strategy to guess the outcome of X is clearly to check $\omega_1, \omega_2, \ldots$ until the correct value has been found. The work factor

$$w_\alpha(X) = \min\left\{k : \sum_{i=1}^{m} p(\omega_i) \geq \alpha\right\} \tag{2.4}$$

equals the minimum number of guesses to find the correct value of X with probability $\geq \alpha$. The *guesswork* quantifies the expected number of guesses that are needed to find the outcome of an experiment that is interpreted as a realization of X. The guesswork of X is defined by

$$W(X) := \sum_{j=1}^{m} j p(\omega_j) \tag{2.5}$$

In [9] Shannon introduced the notion of entropy

$$H(X) = -\sum_{j=1}^{m} p(\omega_j) \log_2(p(\omega_j)) \tag{2.6}$$

with $0 \cdot \log_2(0) := 0$. The quantity $H(X)$ is called the *Shannon entropy* or in short, entropy. The most general definition of entropy is the *Rényi entropy* [10], given by

$$H_\alpha(X) = \frac{1}{1-\alpha} \log_2\left(\sum_{j=1}^{m} p(\omega_j)^\alpha\right), \quad 0 \leq \alpha \leq \infty. \tag{2.7}$$

Obviously, $H_\alpha(X)$ is well-defined for $\alpha \neq 1$. For $\alpha = 1$ we set $H_1(X) := \lim_{\alpha \to 1} H_\alpha(X)$. Using L'Hôpital's Rule it is easy to show (Exercise 6) that

$$H_1(X) = H(X). \tag{2.8}$$

Besides $\alpha = 1$, the limit $\alpha \to \infty$ is also of particular importance, which yields the so-called *min-entropy*

$$H_\infty(X) = \min_{j \leq m}\{-\log_2(p(\omega_j))\}. \tag{2.9}$$

Moreover,

$$H_2(X) = \log_2(\text{Prob}(X = Y)) \tag{2.10}$$

with independent, identically distributed random variables X and Y.

For fixed random variable X, the Rényi entropy $H_\alpha(X)$ decreases monotonically for $\alpha \in [0,\infty)$ (Exercise 7), implying that the min entropy is the most conservative entropy measure.

Example 2.9. Let X denote a binary-valued random variable with $\text{Prob}(X=1) = p \in [0,1]$. Then $H(X) = -(p\log_2(p) + (1-p)\log_2(1-p))$, $H_2(X) = -\log_2(p^2 + (1-p)^2)$, and $H_\infty(X) = \min\{-\log_2(p), -\log_2(1-p)\} = -\log_2(\max\{p, 1-p\})$. For $p = 0.5$ (uniform distribution) we have $H(X) = H_2(X) = H_\infty(X) = 1$.

Example 2.10. Assume that X and Y denote binary-valued random variables. For this example we use the abbreviation $q_{xy} := \text{Prob}((X,Y) = (x,y))$. Let $q_{00} = 0.1$, $q_{01} = 0.3$, $q_{10} = 0.3$, and $q_{11} = 0.3$. Elementary computations yield $H(X,Y) = \log_2(10) - 0.9\log_2(3) = 1.895$, $H_2(X,Y) = 1.837$, and $H_\infty(X,Y) = 1.737$.

The work factor $w_{\frac{1}{2}}(X)$ satisfies the following inequality (cf. [11])

$$\left\lfloor 2^{-H_\infty - 1} \right\rfloor \le w_{\frac{1}{2}} \le \left\lceil \left(1 - 0.5 \sum_{j=1}^{m} \left| p(\omega_j) - \frac{1}{m} \right| \right) m \right\rceil . \tag{2.11}$$

If the random variables $X_1, \ldots, X_n$ iid (e.g., describing a memoryless random source) for large n we have $\log_2(w_\alpha(X_1, \ldots, X_n)) \approx nH(X_1)$ for any α ([12], Section 2.3). Note that for the uniform distribution on Ω we have $H_1 = \log_2(k) = H_\infty$, and in the vicinity of the uniform distribution H_1 and H_∞ give similar values.

In the context of PTRNGs, we are usually faced with stationary processes that assume values in $\Omega = \{0,1\}$, for which the entropy per random bit is close to 1. Usually, these processes only have a short-range memory and/or are rapidly mixing. When guessing long sequences of random bits (e.g., session keys) we obtain the same relation between work factor and Shannon entropy as in the memoryless case (cf. Exercise 8).

This justifies the use of the Shannon entropy $H(X)$, which is easier to handle than the Rényi entropy for parameter $\alpha \neq 1$. This, in particular, concerns the conditional entropy which is relevant when evaluating PTRNGs with dependent random numbers.

Let Y be a further random variable that assumes values in a finite set Ω_Y. For any parameter α

$$H_\alpha(X \mid Y = y) = \frac{1}{1-\alpha} \log_2 \left(\sum_{j=1}^{m} \text{Prob}(X = \omega_j \mid Y = y)^\alpha \right), \tag{2.12}$$

defines the conditional entropy if $Y = y$. (If the random variables X and Y are independent, clearly $\text{Prob}(X = \omega_j \mid Y = y) = p(\omega_j)$ for any pair (ω_j, y).) In particular, the conditional Shannon entropy ($\alpha = 1$) equals

$$H(X \mid Y = y) = -\sum_{j=1}^{m} \text{Prob}(X = \omega_j \mid Y = y) \log_2 \text{Prob}(X = \omega_j \mid Y = y) \tag{2.13}$$

The functional equation of the logarithm function, $\log(ab) = \log(a)\log(b)$, implies

$$H(X \mid Y) = \sum_{y \in \Omega_Y} \text{Prob}(Y = y) H(X \mid Y = y), \tag{2.14}$$

representing the conditional Shannon entropy $H(X,Y)$ as a mixture of conditional entropies (2.13) for which $Y = y$ is fixed. There is no pendant to (2.14) for the min-entropy.

2.6 Non-physical True Random Number Generators (NPTRNGs): Basic Properties

Figure 2.5 illustrates the generic design of a non-physical true RNG. The generic design reminds one of PTRNGs, the *entropy source* being the pendant of the noise source. Unlike the noise source of a physical TRNG, the entropy source of an NPTRNG does not require dedicated hardware but exploits system data (e.g., PC time, RAM data, thread numbers, etc.) and/or human interaction (e.g., key strokes, mouse movement). The entropy of these *raw bits* is usually low, demanding a highly compressing postprocessing algorithm. Moreover, the entropy source is not under the designer's control as it depends on the configuration on the computer used and/or the user himself. This implies considerable differences in the security evaluation of PTRNGs and NPTRNGs.

The NPTRNGs are predestined for software implementation on computers. Example 2.11 addresses a typical design. The output of the NPTRNG may be used 'directly', or it may serve to seed (reseed, update the seed of) a DRNG or as additional input. At least at the beginning of a session, the internal state of the DRNG should be updated, which nullifies attacks on the internal state of the DRNG between the particular sessions. To make attacks on the current internal state (e.g., buffer-overflow attacks) inefficient, the internal state of the DRNG should be updated even during the sessions, e.g., periodically after a particular (small) number of internal random numbers have been output.

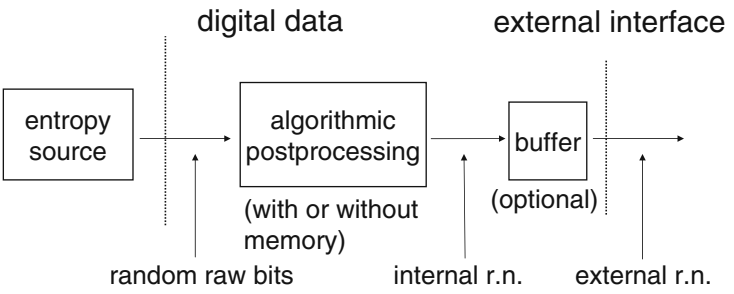

Fig. 2.5 NPTRNG: Generic design.

Example 2.11. Assume that the entropy source uses the system time, the time since system start, several thread numbers and handles, the cursor position and a hash value over a specified RAM area to generate a raw bit string of 1024 bits. This raw bit string is algorithmically postprocessed with the SHA-1 hash function which gives a 160-bit string (internal random number).

Guessing an internal number 'directly' affords 2^{159} trials on average. More interesting is the alternative approach, where the raw bit string is guessed first and then the SHA-1 function is applied to this guess. Unlike in the blind guessing approach, an attacker may exploit his insight into the stochastic properties of the particular components of the raw bit string in order to improve his success probability. We point out that only the one-way property of the SHA-1 hash function is relevant in the context (e.g., when an adversary knows some internal random numbers) whereas collision resistancy is not mandatory. We concentrate on the analysis of a single raw bit string and do not go into detail, but only give general advice.

The central goal of a security evaluation is to ensure that even for small α the work factor w_α is large enough to make guessing attacks infeasible. (The quantitative meaning of 'small' may depend on the intended applications.) To reach this goal the designer, as well as the evaluator, tries to estimate the entropy of the raw bit string. If possible, independent subsets should be identified in order to split the entropy estimation problem into several independent smaller ones. Unlike for PTRNGs, the environments where NPTRNGs run need not essentially be identical but may be very different, which may have impact on the entropy of the raw bits. Unlike for PRTRNGs, it is hardly possible to formulate reliable stochastic models which allow precise entropy estimates.

If the NPTRNG is called automatically when the the PC is booted, at least the time since system start or particular thread numbers should be better predictable (at least for an expert on operating systems) than if the NPTRNG is started on demand. Of course, the concrete implementation, the operating system, and the programs that run on the computer also play a role. From the view of security evaluation, the situation becomes even worse if parts of the raw bit string are derived from the interaction of the user (key strokes, mouse movement). Generally speaking, for NPTRNGs the knowledge of the adversary plays an important role. This may concern technical issues such as the operating system or the used configuration of the attacked system; or the time stamp of an e-mail may provide a rough estimate for the time when the random number was generated. The best the designer or the evaluator of the NPTRNG can do is to determine (and to justify!) a lower entropy bound that shall be valid for all possible implementations and environments, as also for the worst case scenario. Since the raw bits are not stationary, the min-entropy should be the appropriate type of entropy. In practice, normally the Shannon entropy is still used.

Linux operating systems use /dev/random and /dev/urandom to generate random numbers. The function /dev/random may be viewed as a pure NPTRNG while /dev/urandom 'extends' bit strings from the entropy pool, i.e., it may be viewed as a hybrid DRNG that is seeded (and its seed is updated) by an NPTRNG ([13], see also [14]).

We refer the interested reader to [4] which explains a sophisticated attack on the DRNG that is used by Windows 2000. The authors examined the binary code of a particular Windows distribution. This DRNG does not fulfil requirement (R3) since it only requires $O(2^{23})$ operations to get the preceding internal state from the current one. The internal state is periodically updated with an NPTRNG. Since these intervals are too large, and due to the way the random number generator is run by the operating system, a single compromised internal state of the DRNG may compromise up to 128 KBytes of random numbers.

2.7 Standards and Evaluation Guidances

A number of evaluation guidances and standards are effective, and many of them have already been well tried in practice [1, 2, 15–20]. These documents define and explain properties that strong RNGs should have. The evaluation guidances [19] and [20] are technically neutral. They define the criteria and explain how these criteria shall be verified. The criteria are generic in order not to exclude appropriate RNGs. On the other hand, these criteria are clear enough to ensure identical evaluation results, independent of who performs the evaluation. For DRNGs, [19] defines classes K1 to K4 with increasing requirements. Simply speaking, class K2 corresponds to Requirement (R1), while the classes K3 and K4 demand (R1) + (R2) or (R1) + (R3), respectively. Several evaluation guidances and standards [1, 2, 17, 19] give approved designs for DRNGs, or at least discuss examples.

For physical RNGs, it is hardly possible to specify approved designs since secuity-relevant properties depend on the concrete implementation. Of course, particular RNG designs may be analyzed and central steps of the evaluation process defined. Finally, the evaluation yet requires measurements on the concrete implementation. Statistical blackbox tests (as were formulated e.g., in [18, 21]) cannot ensure the security of an RNG.

2.8 Exercises

1. Assume that Alice uses the DRNG from Example 2.2 with Enc =DES to generate 512-bit RSA primes. Discuss this application. Is Alice's choice Enc =DES appropriate?
2. (a) Describe the DRNG from Example 2.5 in our notion. In particular, define the state transition function and the output function.
   (b) Which of the security requirements (R1 to R3) does this DRNG fulfil?
3. Formulate the describing 5-tuple for the Blum-Blum-Shub DRNG (Example 2.6).
4. Example 2.8 describes the ANSI X9.17 hybrid DRNG which has been used in many applications. Assume that an adversary knows all additional input data e_n. Show that this DRNG does not fulfil (R3).

5. Let X denote a random variable that assumes values in $\{0,1\}^{128}$. In particular, $\text{Prob}(X = (0,\ldots,0)) = 0.5$ while $\text{Prob}(X = \omega) = 2^{-128}$ for all strings $\omega \in \{0,1\}^{128}$ that begin with 1. Compute the work factor $w_{0.5}$, the guesswork $W(X)$, the Shannon entropy and the min entropy. Discuss the results.
6. Prove Formula (2.8).
7. Show that the Rényi entropy $H_\alpha(X)$ is monotonically decreasing for $\alpha \in [0,\infty)$.
8. Assume that $X_1, X_2, \ldots$ define a stationary (but not necessarily independent) stochastic process such that the vectors $(X_a,\ldots,X_b)$ and $(X_c,\ldots,X_d)$ are independent if $a < b < c < d$ and $c - b > 1$. Show that $\log_2(w_\alpha(X_1,\ldots,X_n)) \approx H(X_1,\ldots,X_n)$ for any fixed α if n is sufficiently large.
9. Verify Formula (2.14).
10. Consider the NPTRNG from Example 2.11. Assume that the designer adds the absolute time at system start to the raw bit string. Does this increase the security of the NPTRNG? Explain your answer.

2.9 Projects

1. Implement a non-physical true random number generator (NPTRNG) on a PC. Try to state and justify lower entropy bounds. Work out the differences between Windows and Linux operating systems.
2. Implement the DRNG from Exercise 4 in software and determine the data rate that is achievable on your computer.

References

1. ISO/IEC 18031. *Random Bit Generation*. November, 2005.
2. NIST. *Digital Signature Standard (DSS)*. FIPS PUB 186-2, 27.01.2000 with Change Notice 1, 5.10.2001. csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf
3. Lucent Technologies, Bell Labs. *Scientist discovers significant flaw that would have threatened the integrity of on-line transactions*, press article at www.lucent.com/press/0201/010205.bla.html.
4. L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the Windows Random Number Generator. In *Proc. ACM—CCS 2007*, ACM Press, pp. 476–485, New York, 2007.
5. A. J. Menezes, P. C. v. Oorschot, and S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, Boca Raton (1997).
6. J. C. Lagarias. Pseudorandom Number Generators in Cryptography and Number Theory. *Proc. Symp. Appl. Math.*, 42: 115–143, 1990.
7. G. Marsaglia. *Diehard* (Test Suite for Random Number Generators). www.stat.fsu.edu/∼geo/diehard.html

8. D. E. Knuth. Deciphering a Linear Congruential Encryption. IEEE Trans. Inform. *Theory*, 31: 49–52, 1985.
9. C. Shannon. Mathematical Theory of Communication. *Bell System Technology*, 27, 1949.
10. A. Rényi. On the Measure of Entropy and Information. In *Proc. Fourth Berkeley Symp. Math. Stat. Prob. 1 1960*, University of California Press, Berkeley, 1961.
11. J. O. Pliam. *The Disparity Between the Work and the Entropy in Cryptology*, 01.02.1999. eprint.iacr.org/complete/
12. J. O. Pliam. Incompatibility of Entropy and Marginal Guesswork in Brute-*Force Attacks*. In B. K. Roy, E. Okamoto editors, *Indocrypt 2000*, Springer, Lecture Notes in Computer Science, Vol. 2177, 67–79, Berlin, 2000.
13. T. Ts'o. random.c—Linux kernel random number generator. http://www.kernel.org.
14. Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the Linux Random Number Generator. IEEE Symp. on Security and Privacy, IEEE, pp. 371–385, 2006.
15. AIS 20. *Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators*. Version 1, 02.12.1999 (mandatory if a German IT security certificate is applied for; English translation). www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf
16. AIS 31. *Functionality Classes and Evaluation Methodology for Physical Random Number Generators*. Version 1, 25.09.2001 (mandatory if a German IT security certificate is applied for; English translation). www.bsi.bund.de/zertifiz/zert/interpr/ais31e.pdf
17. ANSI X9.82. *Random Number Generation* (Draft Version).
18. NIST. *Security Requirements for Cryptographic Modules*. FIPS PUB 140-2, 25.05.2001 and Change Notice 1, 10.10.2001. csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
19. W. Schindler. *Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators*. Version 2.0, 02.12.1999, mathematical-technical reference of [15] (English translation); www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf
20. W. Killmann and W. Schindler. *A Proposal for Functionality Classes and Evaluation Methodology for True (Physical) Random Number Generators*. Version 3.1, 25.09.2001, mathematical-technical reference of [16] (English translation); www.bsi.bund.de/zertifiz/zert/interpr/trngk31e.pdf
21. NIST. *Security Requirements for Cryptographic Modules*. FIPS PUB 140-1, 11.04.1994. www.itl.nist.gov/fipspubs/fip140-1.htm
22. M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. SIAM *Journal of computers*, 13 850–864: 1984.
23. J.-S. Coron and D. Naccache. An Accurate Evaluation of Maurer's Universal Test. In S. Tavares and H. Meijer editors. *Selected Areas in Cryptography—SAC '98*. Springer, Lecture Notes in Computer Science, Vol. 1556 pp. 57–71, Berlin, 1999.
24. L. Devroye. Non-Uniform Random Variate Generation. Springer, New York, 1986.

25. U. Maurer. A Universal Statistical Test for Random Bit Generators. *Journal of Cryptology*, 5: 89–105, 1992.
26. A. Rukhin et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800–22 with revisions dated (15.05.2001). csrc.nist.gov/rng/SP800-22b.pdf
27. W. Schindler and W. Killmann. Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications. In B. S. Kaliski Jr., Ç. K. Koç, C. Paar editors, *Cryptographic Hardware and Embedded Systems—CHES 2002*, Springer, Lecture Notes in Computer Science 2523, pp. 431-449, Berlin, 2003.