Pseudorandom Number Generators for Cryptographic Applications

Diplomarbeit zur Erlangung des Magistergrades
an der Naturwissenschaftlichen Fakultät
der Paris-Lodron-Universität Salzburg

Andrea Röck

Salzburg, März 2005

Abstract

Random numbers are a fundamental tool in many cryptographic applications like key generation, encryption, masking protocols, or for internet gambling. We require generators which are able to produce large amounts of secure random numbers. Simple mathematical generators, like linear feedback shift registers (LFSRs), or hardware generators, like those based on radio active decay, are not sufficient for these applications.

In this thesis we discuss the properties and a classification of cryptographic random number generators (RNGs) and introduce five different examples of practical generators: /dev/random, Yarrow, BBS, AES, and HAVEGE.

For a complete overview of this topic we also provide an introduction to three mathematical theories that are used in connection with random number generators. We focus our discussion on those theories that try to define the notion of randomness and address Shannon's entropy, Kolmogorov complexity and polynomial-time indistinguishability. In all three cases we study the relation between these theoretical notions and random number generators.

Acknowledgments

I would like to thank my supervisor, Peter Hellekalek, for his advice and encouragement during the development of my thesis and for finding time for me even in stressful moments. Especially, I express my gratitude towards him for always supporting my wish to visit foreign universities. I'm indebted to Nicolas Sendrier for welcoming me for one month in his research group at INRIA-Rocquencourt and for giving me the chance to to get a deeper insight into the functionality of the HAVEGE generator. The help of Stefan Wegenkittl with patiently answering my questions about Markov chains is greatly appreciated.

I'm grateful to Peter Kritzer and Cédric Lauradoux for proofreading my thesis and for all their formal and textual advices which have been a great help during my writing process.

Especially, I'm indebted to my parents Heinz and Elisabeth Röck, who always encouraged and supported me in my studies. In particular, I would like to thank my mother for many mathematical discussions which helped me to solve several tricky problems. Last but not least, I wish to express my gratitude towards my sister, Susanne, and all my friends and colleagues in Tyrol, Salzburg, and all over the world which accompanied me during my studies.

Contents

Chapter 1

Introduction

Random number and random bit generators, RNGs and RBGs, respectively, are a fundamental tool in many different areas. The two main fields of application are stochastic simulation and cryptography. In stochastic simulation, RNGs are used for mimicking the behavior of a random variable with a given probability distribution. In cryptography, these generators are employed to produce secret keys, to encrypt messages or to mask the content of certain protocols by combining the content with a random sequence. A further application of cryptographically secure random numbers is the growing area of internet gambling since these games should imitate very closely the distribution properties of their real equivalents and must not be predictable or influenceable by any adversary.

A random number generator is an algorithm that, based on an initial seed or by means of continuous input, produces a sequence of numbers or respectively bits. We demand that this sequence appears "random" to any observer.

This topic leads us to the question: *What is random?* Most people will claim that they know what randomness means, but if they are asked to give an exact definition they will have a problem doing so. In most cases terms like *unpredictable* or *uniformly distributed* will be used in the attempt to describe the necessary properties of random numbers. However, when can a particular number or output string be called unpredictable or uniformly distributed? In Part I we will introduce three different approaches to define randomness or related notions.

In the context of random numbers and RNGs the notions of "real" random numbers and true random number generators (TRNGs) appear quite frequently. By real random numbers we mean the independent realizations of a uniformly distributed random variable, by TRNGs we denote generators that output the result of a physical experiment which is considered to be random, like radioactive decay or the noise of a semiconductor diode. In certain circumstances, RNGs employ TRNGs in connection with an additional algorithm to produce a sequence that behaves almost like real random numbers.

However, why would we use RNGs instead of TRNGs? First of all, TRNGs are often biased, this means for example that on average their output might contain more ones than zeros and therefore does not correspond to a uniformly distributed random

variable. This effect can be balanced by different means, but this post-processing reduces the number of useful bits as well as the efficiency of the generator. Another problem is that some TRNGs are very expensive or need at least an extra hardware device. In addition, these generators are often too slow for the intended applications. Ordinary RNGs need no additional hardware, are much faster than TRNGs, and their output fulfills the fundamental conditions, like unbiasedness, that are expected from random numbers. These conditions are required for high quality RNGs but they cannot be generalized to the wide range of available generators. Despite the arguments above, TRNGs have their place in the arsenal. They are used to generate the seed or the continuous input for RNGs. In [Ell95] the author lists several hardware sources that can be applied for such a purpose.

Independently of whether a RNG is used for stochastic simulation or for cryptographic applications, it has to satisfy certain conditions. First of all the output should imitate the realization of a sequence of independent uniformly distributed random variables. Random variables that are not uniformly distributed can be simulated by applying specific transformations on the output of uniformly distributed generators, see [Dev96] for some examples or [HL00], which provides a program library that allows to produce non-uniform random numbers from uniform RNGs. In this paper we limit our discussion on generators that imitate uniformly distributed variables.

In a binary sequence that was produced by independent and identically (i.i.d.) uniform random variables the ones and zeros as well as all binary n-tuples for $n \geq 1$ are uniformly distributed in the n-dimensional space. Furthermore there exists no correlation between individual bits or n-tuples, respectively. From the output of a high quality RNG we expect the same behavior. For some generators those conditions can be checked by theoretical analysis, but for most RNGs they are checked by means of empirical tests.

Moreover, a good RNG should work efficiently, which means it should be able to produce a large amount of random numbers in a short period of time. For applications like stochastic simulation, stream ciphers, the masking of protocols or online gambling, huge amounts of random numbers are necessary and thus fast RNGs are required.

In addition to the conditions above RNGs for cryptographic applications must be resistant against attacks, a scenario which is not relevant in stochastic simulation. This means that an adversary should not be able to guess any current, future, or previous output of the generator, even if he or she has some information about the input, the inner state, or the current or previous output of the RNG. This aspect is discussed in detail in Chapter 8.

The topic of cryptographic RNGs concerns both mathematicians and engineers. Most of the time engineers are more interested in the design of specific RNGs or test suites, whereas mathematicians are more concerned with definitions of randomness, theoretical analysis of deterministic RNGs and the interpretation of empirical test results. In this thesis we try to address both disciplines by giving the description of five real-world cryptographic RNGs as well as the necessary mathematical background. We hope that this thesis helps to get a compact overview over the topic of cryptographic RNGs.

1.1 Outline and Summary

Chapter 1 provides the introduction and the outline of the thesis.

In Part I we study mathematical terms and definitions that are used in connection with random number generators. An introduction to this part is given in Chapter 2. Chapter 3 discusses *Shannon's entropy* for random variables and Markov chains, including two methods of estimation. In Chapter 4 we will introduce the *Kolmogorov complexity* of a binary sequence and its connection with *randomness* by means of statistical tests. Finally, in Chapter 5 we will illustrate the definition of a *pseudorandom generator* whose output is *computationally indistinguishable* from real random numbers. Chapter 6 summarizes the results from Part I.

Part II is devoted to the description of practical random number generators. Chapter 7 gives an introduction into this topic including a classification of the RNGs into three categories: pseudorandom number generators (PRNGs), entropy gathering generators, and hybrid generators, and a mathematical model to describe the RNGs. The next chapter covers the topic of possible *attacks* on a random number generator. The danger of attacks distinguishes generators for cryptographic applications from those for stochastic simulations.

In Chapter 9 - 13 we introduce five different generators which represent widely different concepts for producing random numbers. The generator /dev/random (Chapter 9) collects entropy from external events, Yarrow (Chapter 10) is a generic concept that combines the processing of external input with a simple PRNG, the BBS generator (Chapter 11) is completely deterministic and proven to be next-bit unpredictable, AES (Chapter 12) employs an internationally accepted industry standard to produce the random numbers, and finally, the recent generator HAVEGE (Chapter 13) gathers the entropy directly from the internal states of the processor. In Chapter 14 we compare and sum up all the generators previously discussed.

Part I

Theoretical Background

Chapter 2

Overview

The following chapters provide an insight into the mathematical notions which are used in connection with random number generators. We thereby set the focus on terms which define *randomness*. There exist three different mathematical branches, information theory, complexity theory and theory of computability, which we discuss in three separated chapters. In each chapter we point out the connection between the theoretical terms and random number generators.

At first, Chapter 3 deals with Shannon's definition of *entropy*. This term was introduced by Shannon in 1948 in [Sha48] and belongs to the information theory. Entropy is only defined for probability distributions but is often used in connection with random number generators. A sequence of i.i.d random variables is called random if it has the maximal per-bit entropy, which is 1.0. In addition to the illustration of the term we discuss the usage of entropy in the description of RNGs, and different methods of entropy estimation.

Subsequently, in Chapter 4 we cover the *Kolmogorov complexity* of a sequence. This term was developed independently by Kolmogorov, Solomonov, and Chaitin in the 1970's and is part of complexity theory. Kolmogorov's introduction can be found in [Kol65]. The Kolmogorov complexity is defined for individual strings and specifies the minimal length of a program that is able to compute the string. A binary sequence is denoted incompressible if its complexity is almost as large as the length of the sequence itself. This notions are described by means of Turing machines. Additionally, we introduce a definition of statistical tests and use this notion to determine randomness for binary sequences. We show that randomness and incompressibility are equivalent. After that, we discuss the correlation between the Kolmogorov complexity and Shannon's entropy. Since the Kolmogorov complexity is, in general, not computable, it is a theoretical definition. However, we show that it can be approximated by compression methods, like the LZ78 algorithm.

The final chapter discusses *pseudorandomness* and *computational indistinguishability*. Goldreich states in [Gol99] that a sequence of random variables $\{X_n\}_{n\geq 1}$, a so-called ensemble, is called pseudorandom if it is indistinguishable in polynomial-time from a sequence of real random numbers. By real random numbers we mean realizations of i.i.d.

uniform random variables, by computationally indistinguishable or indistinguishable in polynomial-time we mean that the two sequences cannot be told apart by any polynomial-time probabilistic Turing machine. We may also say that an ensemble is pseudorandom if it passes all polynomial-time statistical tests. This approach is rooted in complexity theory. In the end, we discuss if polynomial-time test are sufficient and study pseudorandomness in connection to the Kolmogorov complexity.

Chapter 3

Entropy

Besides "randomness", entropy is probably the most common term used when describing random number generators. It is a measure for the uncertainty about the output of a data source and was introduced for the first time 1948 by C. E. Shannon as a central concept of his work about communication theory. His definition of entropy was an answer to the question:

> Can we find a measure of how much "choice" is involved in the selection of the event or how uncertain we are of the outcome? [SW49, p. 49]

Entropy belongs to the area of information theory. In this chapter we deal with the definition of entropy and some of its properties. Subsequently we investigate entropy in connection with random number generators and, finally, we discuss the entropy estimation of a source.

Shannon's fundamental work on entropy can be found in [SW49]. Further information on this topic is contained in [Ö04] and Cover and Thomas's monograph [CT91]. For the issue of entropy estimation we refer to the work in [Weg98], [Weg01], [HW03], [Weg02], and [Mau92].

3.1 Definition and Characteristics

Shannon analyzed the process of communication. In his basic model, a source sends a message that was chosen out of a set Ω. This message gets encrypted into a symbol by a transmitter and is sent over a defective channel. At the other end of the channel, the received symbol is decrypted by the receiver and forwarded to the destination (see Figure 3.1).

We are only interested in the initial part of the transmission. Shannon wanted to know how much information is contained in the choice of a specific element of the source. R. V. L. Hartley claimed that if an element was chosen uniformly out of a set Ω_n of size $|\Omega_n| = n$, then the information of this choice is determined by [Ö04, p. 4]

$$I(\Omega_n) = \log_2(n). \tag{3.1}$$

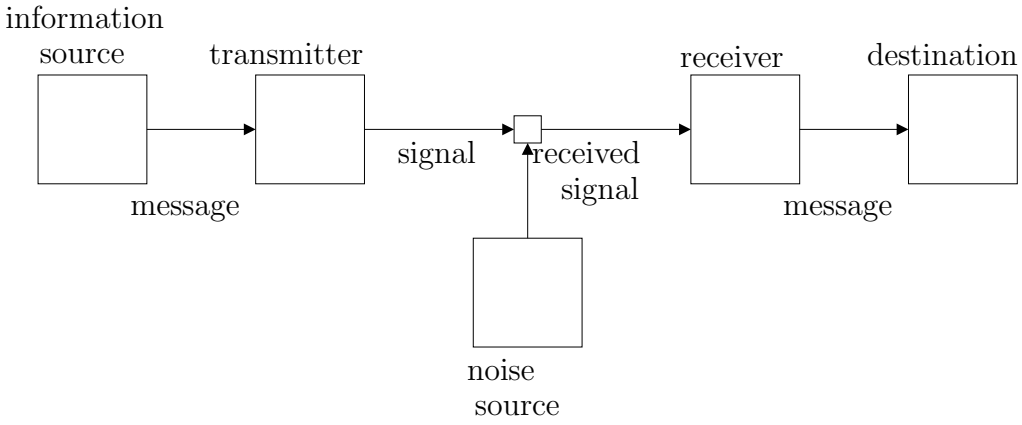

Figure 3.1: Schematic diagram of a general communication system [SW49, p. 34].

Shannon saw a need to modify this result. He was searching for a measure that takes into account the probability of choosing an element of Ω_n even if the corresponding distribution is not uniform. Let us assume that $\Omega_n = \{\omega_1, \omega_2, \ldots, \omega_n\}$ and that p_i, $1 \leq i \leq n$, is the probability of the occurrence of ω_i. Of an appropriate measure $H(p_1, p_2, \ldots, p_n)$ of uncertainty we demand the following properties:

(H 1) H must be continuous in p_i, for all $1 \leq i \leq n$.

(H 2) If $p_i = \frac{1}{n}$ for all $1 \leq i \leq n$, which means that occurrences of the elements of Ω_n are uniformly distributed, then H should monotonically increase with n. This property is based on the characteristic that with uniformly distributed elements the uncertainty of choosing a particular element increases with n.

(H 3) Let us assume that we split the choice into two successive ones, i.e. instead of choosing element x directly out of $\Omega_n = \{\omega_1, \omega_2, \ldots, \omega_n\}$, we first decide if x is ω_1, ω_2, …, ω_{n-2} or falls into the set $\{\omega_{n-1}, \omega_n\}$ and in a second choice we determine if $x = \omega_{n-1}$ or $x = \omega_n$. Without loss of generality let $p = p_{n-1} + p_n > 0$. Then, the original measure should be equal to the measure of the first choice plus the weighted measure of the second choice

$$H(p_1, p_2, \ldots, p_n) = H(p_1, p_2, \ldots, p_{n-2}, p) + pH\left(\frac{p_{n-1}}{p}, \frac{p_n}{p}\right). \tag{3.2}$$

We will illustrate this property by an example (see Figure 3.2). Let us assume that $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{3}$, and $p_3 = \frac{1}{6}$. In the left side of the figure we decide between the three elements at once. In the right side we first decide if we take ω_1 or one of the other two elements. Each event has probability $\frac{1}{2}$. If we chose the second alternative, then we additionally have to decide between ω_2 and ω_3, according to the two new

Figure 3.2: Decomposition of a choice from three possibilities [SW49, p. 49].

probabilities $p'_2 = \frac{2}{3}$ and $p'_3 = \frac{1}{3}$. (H 3) then means that we demand the property

$$H\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right) = H\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{1}{2}H\left(\frac{2}{3}, \frac{1}{3}\right).$$

The weight $\frac{1}{2}$ is necessary because the second alternative is only taken with probability $\frac{1}{2}$.

We then have the following theorem.

Theorem 3.1
Let p_i, $1 \leq i \leq n$, be the probability distribution on $\Omega_n = \{\omega_1, \omega_2, \ldots, \omega_n\}$. The only function H satisfying the three assumptions H1, H2, and H3 above is of the form:

$$H = -K \sum_{i=1}^{n} p_i \log p_i, \tag{3.3}$$

where K is a positive constant.

The conditions as well as the theorem were taken from [SW49, p. 49 f.], whereas the proof can be found in [Fei58, p. 4 ff.]. The result of Theorem 3.1 motivates the definition of entropy for discrete random variables.

Definition 3.2 (Entropy)
Let us assume that X is a discrete random variable on the sample space $\Omega_n = \{\omega_1, \omega_2, \ldots, \omega_n\}$ with probability distribution $P = \{p_i, 1 \leq i \leq n\}$. The entropy $H(X)$ *of X is defined by*

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i.$$

By convention, $0 \log_2 0 = 0$, which is supported by the asymptotic behavior of the function $f(x) = x \log_2 x$, when x tends towards zero,

$$\lim_{x \to 0} x \log_2 x = 0.$$

Therefore, elements occurring with probability 0 have no effect on entropy.

For the definition of entropy we could have chosen any base of the logarithm. Since

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)},$$

for every choice of bases $a, b \geq 2$, a different base would change the value of the entropy only by a constant factor, which conforms to (3.3). We will always use the logarithm in base 2, since this choice corresponds to the binary representation of data in a computer. In this case, the unit of entropy is one bit (binary digit). If we would have chosen the Eulerian number e as base, then the entropy would be measured in nats (natural units).

Shannon did not limit his investigations to the case of a single random variable. He also defined the entropy for sources that can be represented by an ergodic Markov chain. In Section 3.3 we will discuss this topic in greater detail. Here, we focus our discussion to the independent case. For a better understanding of the term entropy, we shall consider two different examples.

Example 3.3
In the first example we will discuss three different cases of random variables, X_1, X_2, and X_3. The three variables have the sample spaces $\Omega_1 = \{\omega_1, \omega_2, \omega_3, \omega_4\}$, $\Omega_2 = \{\omega_1, \omega_2, \ldots, \omega_8\}$, and $\Omega_3 = \{\omega_1, \omega_2, \omega_3, \omega_4\}$, respectively. X_1 and X_2 are uniformly distributed, which means that $p_i^{(1)} = \frac{1}{4}$, $1 \leq i \leq 4$ and $p_i^{(2)} = \frac{1}{8}$, $1 \leq i \leq 8$, respectively, where $p_i^{(j)}$ describes the probability of the event $X_j = \omega_i$. The probability distribution of the last variable X_3 is given by

$$\begin{aligned} p_1^{(3)} &= \frac{1}{2}, \\ p_2^{(3)} &= \frac{1}{4}, \\ p_3^{(3)} &= p_4^{(3)} = \frac{1}{8}. \end{aligned}$$

This implies the following values of entropy.

$$\begin{aligned} H(X_1) &= -\sum_{i=1}^{4} p_i^{(1)} \log_2 p_i^{(1)} \\ &= -4\frac{1}{4}\log_2 \frac{1}{4} \\ &= 2, \end{aligned}$$

$$\begin{aligned} H(X_2) &= -\sum_{i=1}^{8} p_i^{(2)} \log_2 p_i^{(2)} \\ &= -8\frac{1}{8}\log_2 \frac{1}{8} \\ &= 3. \end{aligned}$$

Generally, if X is *uniformly distributed* on a sample space Ω, then $H(X) = \log_2 |\Omega|$, which is conform to (3.1). Thus, Hartley's notion of information is a special case of Shannon's notion of entropy. Concerning X_3, we have

$$\begin{aligned} H(X_3) &= -\sum_{i=1}^{4} p_i^{(3)} \log_2 p_i^{(3)} \\ &= -\left(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{4}\log_2\frac{1}{4} + 2\frac{1}{8}\log_2\frac{1}{8}\right) \\ &= \frac{1}{2}1 + \frac{1}{4}2 + \frac{1}{4}3 \\ &= \frac{7}{4}. \end{aligned}$$

Let us first compare the entropy of the variables X_1 and X_2. $H(X_2)$ is larger than $H(X_1)$, which corresponds to condition (H 2). The more elements are available, the larger is the measure of uncertainty and thus the value of entropy.

The variables X_1 and X_3 have the same sample space, but $H(X_1)$ is larger than $H(X_3)$. X_1 is uniformly distributed, thus if someone wants to guess the outcome of X_1, all elements are equally probable. With X_3, the occurrence of ω_1 is as probable as the occurrence of the other three elements together. If we guess ω_1 as the outcome of X_3 we would be right in half of the cases. Thus, one may state that the value of X_1 is more "uncertain" than the value of X_3.

Theorem 3.4 shows that for a fixed sample space the uniform distribution always has maximum entropy.

Theorem 3.4
Let X be a random number on the sample space Ω_X and $|\Omega_X|$ denotes the number of elements in the range of X. Then,

$$H(X) \leq \log_2 |\Omega_X|,$$

with equality if and only if X has a uniform distribution over Ω_X.

The theorem as well as the proof can be found in [CT91, p. 27].

This characteristic of the uniform distribution is often used in relation with random number generators. A number is denoted "real random", if it was chosen according to a uniform distribution, which provides maximal entropy. Let X be the random variable describing the output of a RNG. From a good RNG we would expect $H(X)$ to be as high as possible.

Example 3.5 (Minimal number of binary questions)
In this example we study the number of binary questions that are, on average, necessary to find out which element was produced by our information source. We will discuss the connection of this number to Shannon's entropy.

Let us assume that the random variable X represents the source, such that Ω is the sample space and $P = \{p(x), x \in \Omega\}$ the probability distribution of X. A questioning strategy S specifies which binary questions we will ask to determine the chosen element x. E.g., if $\Omega = \{1, 2, \ldots, 8\}$, then such a question could be "Is x smaller than 4?". The value $a(x, S)$ determines the number of questions that are necessary to find x if we use the strategy S. Furthermore, S_Ω represents the set of all possible questioning strategies on the sample space Ω. Then,

$$E_P[a(X, S)] = \sum_{x \in \Omega} a(x, S)p(x)$$

is the expectation of the number of questions. Österreicher defines in [Ö04, p. 17] the exact entropy ("wirkliche Entropie") by the minimal number of necessary questions. This term was originally introduced by Topsøe in [Top74].

Definition 3.6 (Exact Entropy)
The term

$$H^*(X) = \min_{S \in S_\Omega} E_P[a(X, S)]$$

represents the exact entropy *of X with probability distribution P. The strategy S^* is called optimal if*

$$H^*(X) = E_P[a(X, S^*)].$$

Furthermore Österreicher shows in [Ö04, p. 21] that the following condition holds.

Theorem 3.7 *With the notation as above,*

$$H(X) \leq H^*(X) \leq H(X) + 1$$

This means that Shannon's entropy is approximately equal to the exact entropy and thus a measure for the expected number of necessary binary questions to determine a certain element. Consequently, if an element was produced by a information source of high entropy, then an adversary needs, on average, more effort guessing the specific element.

Information theory, to which the notion of entropy belongs, is a much broader topic. We have limited our excursion into this field to the definitions and theorems above, since they cover most of the notions needed in the description of RNGs. In the rest of this chapter we will study in which ways the term entropy occurs in connection with random number generators and how we are able to estimate the entropy of a generator.

3.2 Entropy and RNGs

The term entropy is applied in various ways in the description of random number generators. In this section we are going to consider some of them in more detail. Generally, entropy is only defined for information sources. However, we will show here in which other ways it is applied.

3.2.1 RNG as Information Source

In this case, the random number generator itself is considered to be an information source. The output is interpreted as a realization of independent, identically distributed (i.i.d.) random variables or as a Markov chain as in Section 3.3. An optimal RNG is represented by i.i.d. uniform random variables and provides a per-bit entropy of 1.0, i.e. if the generator produces n-bit output, then this distribution of the output must have the maximal entropy of n-bits. For some completely deterministic generators (see Section 7.1.1), this distribution can be analyzed for each specific initial value. However, if the computation is too complex or if unpredictable components, like external events influence the output, then most of the time we are limited to the use of entropy estimators as in Section 3.3.

3.2.2 External Input as Information Source

Descriptions of RNGs often refer to the entropy of the input. In this case, we assume that the input is produced from an external source and that we are, at least theoretically, able to calculate the entropy. In practice, the entropy of the input is only estimated, and not always by means of mathematically proven methods.

3.2.3 Shifting of Entropy

In later chapters we will use the phrase that entropy gets shifted from one pool $\mathcal{P}_1$ to another pool $\mathcal{P}_2$. A pool is represented by a data storage, e.g. an array of fixed length, which, as a whole, can take various values. If we assume that each value occurs with a specific probability, then we can interpret the pool as a realization of a random variable and thus are able to determine its entropy. However, the probability distribution can vary with the time. If we mix unknown input into the content of the pool we may increase the uncertainty of the content of the pool and thus its entropy. If we use the value of the pool to generate an output, we may gain some information about the pool by observing the output. Therefore the uncertainty of the pool and consequently, the entropy gets decreased.

To shift information from pool $\mathcal{P}_1$ to pool $\mathcal{P}_2$ we first generate output from $\mathcal{P}_1$, which decreases the entropy of $\mathcal{P}_1$, and subsequently use the generated data as input for $\mathcal{P}_2$ and, thus, increase the entropy of $\mathcal{P}_2$. If we use an entropy-preserving function (see Section 3.2.4) for mixing the data into $\mathcal{P}_2$, then the entropy of $\mathcal{P}_1$ is decreased by the same amount as the entropy of $\mathcal{P}_2$ is increased.

3.2.4 Entropy-Preserving Functions

We shall now study the notion of an *entropy-preserving function*. Let us assume that X and Y are random variables on the sample spaces Ω_X and Ω_Y with probability distributions $P_X = \{p_X(x), x \in \Omega_X\}$ and $P_Y = \{p_Y(y), y \in \Omega_Y\}$, respectively.

Definition 3.8 (Ideal Entropy-Preserving Function)
Let $f : \Omega_X \to \Omega_Y$ be a function such that

$$p_Y(y) = \begin{cases} p_X(x), & \text{if } y = f(x) \\ 0, & \text{otherwise.} \end{cases}$$

Then we call f an ideal entropy-preserving function *if*

$$H(X) = H(f(X)) = H(Y).$$

Remark 3.9 *This condition is satisfied if f is injective, since in this case*

$$\begin{aligned} H(Y) &= -\sum_{y\in\Omega_Y} p_Y(y)\log_2 p_Y(y) \\ &= -\sum_{f(x)\in f(\Omega_X)} p_Y(f(x))\log_2 p_Y(f(x)) \\ &= -\sum_{x\in\Omega_X} p_X(x)\log_2 p_X(x) \\ &= H(X) \end{aligned}$$

Remark 3.10 *However, if f is not injective and maps two values ω'_X and ω''_X with probabilities greater than 0, i.e. $p_X(\omega'_X) > 0$ and $p_X(\omega''_X) > 0$, to the same value ω'_Y, then the entropy $H(f(X))$ is smaller than $H(X)$.*

Proof. Without loss of generality we assume that for a given $n \in \mathbb{N}$
$\Omega_X = \{\omega_1, \omega_2, \ldots, \omega_n\}$, $\Omega_Y = \{\omega_1, \omega_2, \ldots, \omega_{n-1}\}$, and

$$f(x) = \begin{cases} \omega_{n-1}, & \text{if } x = \omega_{n-1} \text{ or } \omega_n \\ x, & \text{otherwise.} \end{cases}$$

$p_X(\omega_{n-1}) > 0$, $p_X(\omega_n) > 0$, and $p = p_X(\omega_{n-1}) + p_X(\omega_n) > 0$. From condition (3.2) follows

$$\begin{aligned} H(X) &= H\left(p_X(\omega_1), p_X(\omega_2), \ldots, p_X(\omega_n)\right) \\ &= H\left(p_X(\omega_1), p_X(\omega_2), \ldots, p_X(\omega_{n-2}), p\right) + pH\left(\frac{p_X(\omega_{n-1})}{p}, \frac{p_X(\omega_n)}{p}\right) \\ &= H\left(p_Y(\omega_1), p_Y(\omega_2), \ldots, p_Y(\omega_{n-2}), p_Y(\omega_{n-1})\right) + pH\left(\frac{p_X(\omega_{n-1})}{p}, \frac{p_X(\omega_n)}{p}\right) \\ &= H(Y) + pH\left(\frac{p_X(\omega_{n-1})}{p}, \frac{p_X(\omega_n)}{p}\right). \end{aligned}$$

Thus,

$$H(X) = H(Y) + pH\left(\frac{p_X(\omega_{n-1})}{p}, \frac{p_X(\omega_n)}{p}\right). \tag{3.4}$$

Since $p_X(\omega_{n-1})$, $p_X(\omega_n)$, and p are all greater than zero the right summand is larger than zero as well, and hence

$$H(X) > H(Y).$$

□

If we merge new input into a pool by means of an entropy-preserving function, we can add the entropy of the input to the entropy of the pool. Let us assume that the random variable X represents the input source, Y the pool previous to the input, and Z the pool after the input was merged. The processing of the input happens by the function $f : \Omega_X \times \Omega_Y \to \Omega_Z$. If f is an entropy-preserving function then

$$H(X, Y) = H(f(X, Y)) = H(Z),$$

where $H(X, Y)$ is the joint entropy of X and Y.

Definition 3.11 (Joint Entropy)
The joint entropy $H(X, Y)$ *of a pair* (X, Y) *of discrete random variables with joint distribution* $p(x, y)$ *is defined as*

$$H(X, Y) = -\sum_{x \in \Omega_X} \sum_{y \in \Omega_Y} p(x, y) \log_2 p(x, y).$$

From the chain rule in [CT91, p. 17] we obtain

Corollary 3.12 *If* X *and* Y *are independent random variables, then*

$$H(X, Y) = H(X) + H(Y).$$

Remark 3.13 *Thus, if* f *is an entropy-preserving function and the input* X *and the old content of the pool* Y *are independent, then the entropy of the new content is the sum of the two previous entropies, i.e.,*

$$H(Z) = H(X, Y) = H(X) + H(Y).$$

One class of functions that are claimed to be (almost) entropy-preserving is the family of cryptographic hash functions. Hash functions map a large set into a much smaller set, in our case this means that they map long binary sequences to shorter ones. The conditions for a function to be a cryptographic hash function $h : \Omega_1 \to \Omega_2$ are that

1. h is a one way function, which means it is easy to calculate $h(x)$, i.e. realizable in polynomial time, but there exists no polynomial time function that determines $h^{-1}(y)$,

2. h should be almost collision free. By a collision we mean that two input values from Ω_1 are assigned to the same value in Ω_2. Thus, a function is denoted to be collision free if m different elements of Ω_1 with $m \leq |\Omega_2|$ should be assigned to m different elements in Ω_2. The nearer m comes to $|\Omega_2|$ the more collisions will occur.

3. If one of the input bits gets modified, then about half of the output bits should change. This equals the condition that each output-bit should depend on almost every input bit.

The second condition means that we have a low probability of collision and thus, p in (3.4) is small. Consequently,

$$H(X) = H(Y) + pH\left(\frac{p_X(\omega_{n-1})}{p}, \frac{p_X(\omega_n)}{p}\right) \approx H(Y).$$

Due to this characteristic, hash functions are often referred to as entropy-preserving functions.

3.3 Estimation of Entropy

Shannon was not satisfied with analyzing information sources that correspond to i.i.d. random variables. He also studied the case in which the probability distribution of the current output depends on previous output of the source. Such a stochastic sequence $(X_n)_{n\geq 1}$ is called a Markov chain (MC). Shannon considered the special case of ergodic MCs.

First, we will define the notion of an ergodic Markov chain. Subsequently, we will introduce two methods which allows to estimate the entropy of such a chain. Since the case of i.i.d. random variables corresponds to a MC of order $\kappa = 0$, all our results also apply to this special case. Let us begin with the definition of a Markov chain.

Definition 3.14 (Markov Chain of Order κ)
The sequence $(X_n)_{n\in\mathbb{N}}$ is called a homogeneous Markov chain of oder κ *with discrete time and state space S if and only if for every $n \in \mathbb{N}$ the condition*

$$\begin{aligned} P[X_{n+1} = j | X_0 = i_0, \ldots, X_n = i_n] &= P[X_{n+1} = j | X_{n-\kappa+1} = i_{n-\kappa+1}, \ldots, X_n = i_n] \\ &= p_{i_{n-\kappa+1}\ldots i_n j} \end{aligned}$$

is satisfied for all $(i_0, \ldots, i_n, j) \in S^{n+2}$, for which $P[X_0 = i_0, \ldots, X_n = i_n] > 0$.

Thus, the probability of X_n depends only on the previous κ states and is independent of the specific n.

Remark 3.15 *In the case of Markov chains we speak of the state space S instead of the sample space Ω.*

Generally, Markov chains are also interesting for the description of RNGs, because they allow to model the behavior of a generator if there exists a correlation between different output values. In the following we will limit ourselves to the special case of a MC of order $\kappa = 1$. We observe as in [Weg98] that for each MC $(X_n)_{n\geq 1}$ of order κ, the sequence

$(\tilde{X}_l^{(d)})_{l\geq 1}$ of overlapping $d-tuples$ (see Remark 3.23) forms a MC of order $\kappa' = 1$ if $\kappa \leq d$. Thus, it it is sufficient to consider only Markov chains of order one. For more details see [Weg98, p. 21]. The advantage of a MC of order $\kappa = 1$ is that it can be easily defined by means of a transition matrix.

Definition 3.16 (Markov Chain of Order $\kappa = 1$)
Let

- $S = \{1, \ldots, m\}$ *be the state space of the random variables* X_n, $n \geq 1$,
- $\mathbb{P} = (p_{ij})_{(i,j)\in S^2}$ *be the transition matrix such that*

$$\begin{aligned} p_{ij} &\geq 0 \qquad \forall (i,j) \in S^2, \\ \sum_{j\in S} p_{ij} &= 1 \qquad \forall i \in S, \end{aligned}$$

and

- $\mathbf{P} = (p_{01}, \ldots, p_{0m})$ *be the initial distribution with*

$$\begin{aligned} p_{0i} &> 0 \qquad \forall i \in S, \\ \sum_{i\in S} p_{0i} &= 1. \end{aligned}$$

Then the triple $(S, \mathbb{P}, \mathbf{P}_0)$ *describes a* homogeneous Markov chain $(X_n)_{n\in\mathbb{N}}$ *of order* $\kappa = 1$ *if and only if for every* $n \in \mathbb{N}_0$ *the condition*

$$P[X_{n+1} = j | X_0 = i_0, \ldots, X_n = i_n] = P[X_{n+1} = j | X_n = i_n] = p_{i_n j}$$

is satisfied for all $(i_0, \ldots, i_n, j) \in S^{n+2}$, *for which* $P[X_0 = i_0, \ldots, X_n = i_n] > 0$. *The probability distribution of the specific random variable* X_n *is then given by*

$$\mathbf{P}_0 \mathbb{P}^n,$$

where $\mathbb{P}^n$ *represents the n'th power of the transition matrix.*

An important probability distribution of a MC is its so-called stationary distribution.

Definition 3.17 (Stationary Distribution)
A distribution $\mathbf{P} = (p_1, \ldots, p_m)$ *on the sample space* S *which satisfies* $\sum_{i\in S} p_i p_{ij} = p_j$ *for all states* $j \in S$ *or* $\mathbf{P}\mathbb{P} = \mathbf{P}$ *in matrix notation, is called a* stationary distribution.

Definition 3.18 (Stationary Markov Chain)
Let P *be the stationary distribution of the MC* $(S, \mathbb{P}, \mathbf{P}_0)$. *The* stationary Markov chain $(S, \mathbf{P})$ *represents the chain that is generated by setting* $\mathbf{P}_0 = \mathbf{P}$. *In such a chain all random variables* X_n *of the MC* $(X_n)_{n\in\mathbb{N}}$ *have the same probability distribution* $\mathbf{P}$.

Now we return to the term "ergodic". We call a MC ergodic if the stochastic properties of sequences produced by the MC are independent of the length and the starting time of the sequence. In more detail, this means that the relative frequencies of single elements or d-dimensional tuples converge, for n approaching infinity, toward a fixed limit for almost every sequence. Sequences for which this property does not hold have probability zero. Ergodicity is satisfied if the MC is *finite*, *irreducible* and *aperiodic*.

A Markov chain is called *finite* if its sample space S is finite. In our case we only consider finite Markov chains. Let $p_{ij}^{(n)}$ denote the elements of the n'th power matrix $\mathbb{P}^n$, then we can give following definitions.

Definition 3.19 (Irreducibility)
A MC is called irreducible *(or undecomposable) if for all pairs of states $(i,j) \in S^2$ there exists an integer n such that $p_{ij}^{(n)} > 0$.*

By the *period* of a irreducible MC we denote the smallest number p such that the set of all possible return times from i to i can be written as $p\mathbb{N} = \{p, 2p, 3p, \ldots\}$, i.e.

$$p = \gcd\{n \in \mathbb{N} : p_{ii}^{(n)} > 0\}.$$

Definition 3.20 (Aperiodicity [Weg98, p. 14])
An irreducible chain is called aperiodic *(or acyclic) if the period p equals 1.*

The following lemma holds.

Lemma 3.21 *Let $(S, \mathbb{P}, \mathbf{P}_0)$ be a finite, irreducible, aperiodic MC with stationary distribution $\mathbf{P}$. Then $\mathbf{P} = (p_1, \ldots, p_m)$ is given by*

$$p_j = \lim_{n\to\infty} p_{ij}^{(n)} \tag{3.5}$$

for all pairs $(i,j) \in S^2$ of states.

In a more general version this lemma can be found in [Weg98, p. 15 ff.].

Markov chains that satisfy Condition (3.5), are called *ergodic*. According to this, the behavior of an ergodic Markov chains approximates the behavior of a stationary Markov chain if n approaches infinity. If not mentioned otherwise we will assume in the following that all Markov chains are ergodic and of order $\kappa = 1$.

After we have studied the necessary definitions we are now able to define the entropy of a Markov chain.

Definition 3.22 (Entropy of a Markov Chain)
Denote by $H(S, \mathbb{P})$ the entropy *of the ergodic chain $(S, \mathbb{P}, \mathbf{P}_0)$,*

$$H(S, \mathbb{P}) = -\sum_{i=1}^{m} p_i \sum_{j=1}^{m} p_{ij} \log_2 p_{ij},$$

where $\mathbf{P} = (p_1, \ldots, p_m)$ is the stable distribution of the chain.

In the remaining part of this section we will deal with the entropy estimation of Markov chains. To do so, we need some notation.

Remark 3.23 (Notation)
Let $(S, \mathbb{P}, \mathbf{P}_0)$ be an ergodic MC with finite state space S, where $|S| = m$. For simplicity we assume that $S = \{1, 2, \ldots, m\}$.

- $(X_n)_{n\geq 1}$ *denotes a sequence of random variables underlying* $(S, \mathbb{P}, \mathbf{P}_0)$.
- $(x_n)_{n=1}^N$ *is a realization of length N of the MC.*
- x_i^j, $i \leq j$ *denotes the subsequence* $(x_i, x_{i+1}, \ldots, x_j)$.
- $(X_l^{(d)})_{l\geq 1}$ *is the sequence of overlapping d-tuples, where* $X_l^{(d)} = (X_l, \ldots, X_{l+d-1})$ *for all* $l \leq 1$.
- $(\tilde{X}_l^{(d)})_{l\geq 1}$ *is the sequence of overlapping d-tuples defined in a cyclic way on the sequence* $(X_l)_{l=1}^n$ *of length n, such that* $\tilde{X}_l^{(d)} = X_r^{(d)}$ *for all* $l \geq 1$ *if* $l - 1 \equiv r - 1$ *modulo n.*
- $(\overline{X}_l^{(d)})_{l\geq 1}$ *is the sequence of non-overlapping d-tuples, where* $\overline{X}_l^{(d)} = (X_{(l-1)d+1}, \ldots, X_{ld})$ *for all* $l \geq 1$.
- $\hat{P}^{(d)}(n) = (\hat{p}_{\mathbf{i}}^{(d)}(n))_{\mathbf{i}\in S^d}$ *is the vector of relative frequencies of the overlapping tuples in* x_1^n, *with*
$$\hat{p}_{\mathbf{i}}^{(d)}(n) = \frac{1}{n}\#\{1 \leq l \leq n : \tilde{X}_l^{(d)} = \mathbf{i}\}$$

We will now introduce two methods, the Overlapping Serial Test and Maurer's Universal Test, which may be used to estimate the entropy of a MC. The first algorithm employs the relative frequency of overlapping tuples, the second one the return time of non-overlapping tuples. Both methods are explained and compared by means of empirical results in [HW03]. In addition to the pure entropy estimation, the algorithm can be applied for testing whether the output of a RNG corresponds to a realization of i.i.d. uniform random variables and thus can be denoted "random".

3.3.1 Overlapping Serial Test

The first estimator $\hat{H}_f^{(d)}$ uses the relative frequency of overlapping d-tuples,

$$\hat{H}_f^{(d)} = -\sum_{\mathbf{i}\in S^d} \hat{p}_{\mathbf{i}}^{(d)}(n) \log_2 \hat{p}_{\mathbf{i}}^{(d)}(n) + \sum_{\mathbf{i}\in S^{d-1}} \hat{p}_{\mathbf{i}}^{(d-1)}(n) \log_2 \hat{p}_{\mathbf{i}}^{(d-1)}(n).$$

$\hat{H}_f^{(d)}$ is an *asymptotically unbiased* and *consistent* estimator for the entropy of a Markov chain $(S, \mathbb{P}, \mathbf{P}_0)$, if the order κ of the Markov chain is less than d. Asymptotically unbiased means that

$$\lim_{n\to\infty} E(\hat{H}_f^{(d)}) = H(S, \mathbb{P}),$$

where E represents the expectation of the random variable. Consistency stands for the condition that the estimator converges in probability toward the true value, i.e. for all $\epsilon > 0$

$$\lim_{n\to\infty} P[|\hat{H}_f^{(d)} - H(S,\mathbb{P})| \leq \epsilon] = 1.$$

Wegenkittl shows in [Weg01, p. 2484] in detail that the estimator even converges almost surely (a.s.) if $d > k$, and $n \to \infty$, which means

$$P[\lim_{n\to\infty} \hat{H}_f^{(d)} = H(S,\mathbb{P})] = 1.$$

A special characteristic of $\hat{H}_f^{(d)}$ is that for $d = 1$ it estimates the entropy $H(\mathbf{P})$ of the stable distribution of the chain. In the independent case ($k = 0$) this is equivalent to the entropy of the chain. However, we should not forget that if $d \leq \kappa$, then $\hat{H}_f^{(d)}$ only estimates the d-dimensional marginal distribution, which may differ considerably from the entropy $H(S,\mathbb{P})$ of the chain.

The estimator $\hat{H}_f^{(d)}$ can also be used as a statistical test for RNGs. It represents a special class of generalized serial tests. For the test we apply the scaled factor $\hat{I}^{(d)} = 2n(1 - \hat{H}_f^{(d)})$, which is compared by the Kolmogorov-Smirnov test to the chi-square distribution $\chi^2_{2^d-2^{d-1}}$ with $2^d - 2^{d-1}$ degrees of freedom. This is done due to the fact that if the MC represents i.i.d. uniform random variables, then $\hat{I}^{(d)}$ converges in distribution to $\chi^2_{2^d-2^{d-1}}$. This means that for all $a \in \mathbb{R}$

$$\lim_{n\to\infty} P[\hat{I}^{(d)} \leq a] = P[\chi^2_{2^d-2^{d-1}} \leq a].$$

3.3.2 Maurer's Universal Test

The second estimator $\hat{H}_r^{(d)}$ calculates the entropy using the return time of non-overlapping d-tuples. The estimator was introduced in [Mau92] and underlies Maurer's Universal Statistical Test for random bit generators. The universality corresponds to the fact that the test should find every defect in the output of a RNG that can be modeled by a finite order Markov chain, if the dimension d of the tuples and the sample size n approaches infinity.

Let, for every $0 \leq i \leq n$,

$$T^{(d)}(i) = \min\{1 \leq j \leq i+1 : \overline{x}_{i-j}^{(d)} = \overline{x}_i^{(d)}\}$$

denote the truncated return time to $\overline{x}_i^{(d)}$. If we set $\overline{x}_{-1}^{(d)} = \overline{x}_i^{(d)}$, then $T(i)$ is well defined and finite. The estimator is defined by

$$\hat{H}_r^{(d)} = \frac{1}{dn} \sum_{i=Q}^{Q=n-1} \log_2 T^{(d)}(i).$$

Q represents the "warm up" of the test and should guarantee that, with high probability, each possible d-tuple occurs at least once in the sequence prior to the test, so that $T^{(d)}(i)$

provides significant results for all $i \geq Q$. Maurer recommends in [Mau92] to choose Q greater than $10m^d$. If we want to analyze n overlapping d-tuples we need a sample size of $d(Q+n)$.

$\hat{H}_r^{(d)}$ is an *asymptotically unbiased* and *consistent* estimator for $\frac{1}{d}E(\log_2 T^{(d)})$, but not for the entropy $H(S,\mathbb{P})$ itself. $E(\log_2 T^{(d)})$ is the expectation of the return time $T^{(d)}(0)$ of the first non-overlapping d-tuple. As d approaches infinity, the scaled expectation of the return times converges toward the entropy of the Markov chain. Wegenkittl showed in [Weg01, p. 2486] that, for every Markov chain $(S,\mathbb{P},\mathbf{P}_0)$,

$$\lim_{d\to\infty} \frac{E(\log_2 T^{(d)})}{d} = H(S,\mathbb{P}).$$

Thus, if the sample size n and the dimension d approaches infinity, $\hat{H}_r^{(d)}$ estimates the entropy $H(S,\mathbb{P})$ of the MC. In [HW03] the quantity

$$\hat{N}^{(d)} = \frac{\hat{H}_r^{(d)} - E_{d\log_2 m}}{\sqrt{V_{d\log_2 m}}}$$

was used as test statistic to find defects in the output of a RNG, which can be modeled by a finite order MC. $E_{d\log_2 m}$ and $V_{d\log_2 m}$ are the numerically computed expectation and variation of $d\hat{H}_r^{(d)}$ and can be found in [Mau92, Table I, p. 14]. For i.i.d. uniform random variables, $\hat{N}^{(d)}$ converges to $N[0,1]$, the standard normal distribution. Thus, in the test this distribution was employed in the Kolmogorov-Smirnov test.

Generally, we see that for n and d large enough, both methods can be applied to estimate the entropy of a Markov chain. To achieve the same number of tuples, the second estimator needs an about d times larger sample. In [Weg01] the author compares the the empirical behavior of both estimators. For a fixed tuple size $d > 1$ and increasing number of tuples n, Maurer's estimator approximates the entropy of an independent Markov chain a little faster. However, one has to consider that, for the same n, this method needs a much larger sample size. For a fixed n and increasing d the overlapping serial test converged much faster to the entropy of the chain in the independent as well as in the dependent case.

In [HW03] the corresponding statistical tests of the estimators are compared and applied on RNGs with defects of specified order. For these tests the sample size was fixed, thus in the second case much less tuples were available. In general the overlapping serial test performed better. Maurer's test found only those defects with order $k = d$ and $k = 2d$, but not all defects with $k < d$.

3.3.3 Entropy and Cryptographic RNGs

Entropy as a measure of the uncertainty of a random variable is a useful tool to describe the quality of the output or respectively the input of a random number generator. It enables us to describe several processes within a generator and how much information an adversary actually has about the inner state of the RNG.

Since the exact probability distribution of the corresponding random variable is rarely known, we have to use entropy estimators. We have to bear in mind that an overestimation of the entropy due to wrong assumptions or inappropriate estimators can have severe effects on the security of the generator (see Chapter 8).

Chapter 4

Kolmogorov Complexity

In this section we discuss the Kolmogorov complexity of finite sequences. This approach was independently developed by Kolmogorov, Solomonov, and Chaitin in the 1970's. Kolmogorov complexity defines the complexity of a given binary sequence as the size of the minimal program that is able to reproduce the sequence.

To show some examples of different complexities let us first consider three different sequences of length $n = 64$. Which one looks most random? Which one can be most easily described?

1. 0101010101010101010101010101010101010101010101010101010101010101

2. 0110101000001001111001100110011111111001110111100110010010000 1000

3. 1101111001110101111101101111101110101101111000101110010100111011

The first sequence consists of thirty-two blocks of the form 01 and most people will not call it random. The second one looks random at first sight, but it represents the first bits of the binary expansion of $\sqrt{2}-1$ and is thus easily reproducible and has a short description. The third sequence again looks random, although, it has more ones than zeros. It turns out that if k is the number of ones then the sequence can be described approximately by $\log_2 n - n(\frac{k}{n}\log_2\frac{k}{n} + \frac{n-k}{n}\log_2\frac{n-k}{n})$ bits. This is done by a program of the form:

> Generate all sequences with k ones in lexicographical order and output the ith sequence. [CT91, p. 152]

In this program only the variables i and n depend on the specific sequence and are thus crucial for the necessary length.

If we increase the length n, the description of the first two sequences stays the same, if n is known, whereas the length of the description of the third sequences growths, although it remains clearly smaller than n if $k < \frac{n}{2}$. Intuitively, we could say neither of the sequences has a complexity as high as a "real" random number, although the third one is more complex than the first two.

In this chapter we explain all terms needed in the definition of Kolmogorov complexity. Subsequently, we introduce the notation of an incompressible sequence and show that it corresponds to a definition of randomness introduced by statistical tests. Furthermore, we compare Kolmogorov complexity with Shannon's entropy and discuss its computability. Finally, we question if the concept of Kolmogorov complexity is applicable in practice.

A basic description of this topic as well as the three sequences above can be found in [CT91, Chapter 7]. However, most of the definitions and theorems of this chapter were taken from Li and Vitanyi's book [LV93] where the authors cover the Kolmogorov complexity in details. In the final part of this chapter we discuss a possible method for estimating the Kolmogorov complexity. For this topic we refer to [LZ76] and [EHS02] for details.

4.1 Definition and Characteristics

Before we start with further explanations, we will introduce some notation we are going to use during this chapter. Let us assume that there exists a bijective function from the set of finite binary sequences onto the natural numbers. The ordinary binary representation of integers is inappropriate for this assignment because it is ambiguous, e.g. both 0010 and 10 represent the integer 2. A simple unambiguous function could be realized by ordering the finite binary strings lexicographically by their length and mapping each binary sequence to its position in the list. If ϵ represents the empty binary string, then the ordered list would look like $(\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots)$ and the assignments like $(\epsilon \to 0, 0 \to 1, 1 \to 2, 00 \to 3, \ldots)$. For the remaining part of this chapter we assume that we have agreed on such an unambiguous mapping. We will consider the binary string and its assigned integer as the same object and switch between both representations, like $x = 00 = 3$. By $l(x)$ for $x \in \mathbb{N}$ we mean the length of the binary equivalent of x, e.g. if $x = 3$, then $l(x) = 2$.

4.1.1 Turing Machine (TM)

For defining the Kolmogorov complexity we use Turing machines. Allan Turing described this simple computer in the 1930s in the attempt to determine the class of all computable functions. It can be shown that until now, every other attempt to define this class of function is equivalent to Turing's definition. For the Kolmogorov complexity we need the special class of universal Turing machines (UTM)

A *Turing machine* has an infinitely long *tape*, which is partitioned into cells. Each cell contains a character from the finite alphabet $A = \{0, 1, B\}$ where B represents the *blank* symbol. At the beginning all cells contain a B except of a finite consecutive sequence right beside the starting cell. This sequence is the input of the TM.

Furthermore, there is a *read/write head*, which points at each time to a particular cell of the tape. It scans the character of the cell and is able to execute one of three commands:

- *Move* one position to the *right*.
- *Move* one position to the *left*.
- *Write* the symbol $a \in A$.

The TM also contains a *finite control*, which is always in one of the states q_i out of the finite state space Q.

The behavior of the TM is determined by a finite list of *rules*. A rule indicates in which way the TM reacts on a specific state of the control and a specific character read by the head, e.i., it indicates which command the head is going to perform next and which new state the control receives. A rule can be be described by a tuple (p, s, a, q), where $p \in Q$ is the current state of the control and $s \in A$ is the symbol scanned from the current cell. $a \in S$ is the command the read/write head has to perform, with $S = \{L, R, 0, 1, B\}$ and q is the state the finite control receives after the step. For each pair (p, s) there exist at most one rule. If there exists no rule for a specific pair, then the machine *holds* as soon as it reaches the given state. The TM is therefore defined by a mapping of finite subsets of $Q \times A$ into $S \times Q$.

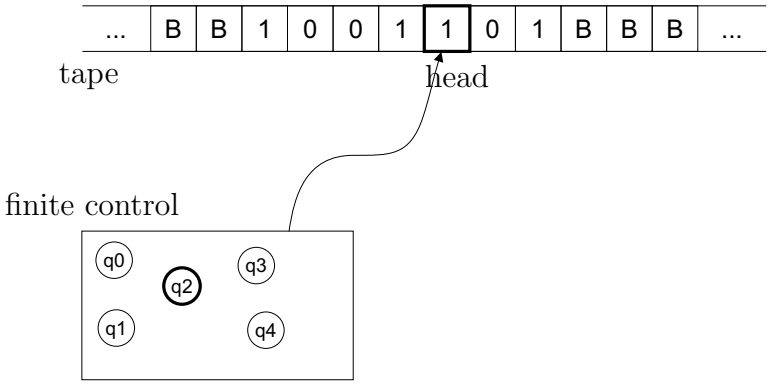

Figure 4.1: Turing Machine [LV93, p. 27]

The computation by means of a TM starts by convention in state q_0, with the head pointing at the starting cell. The input of the TM is represented by a single binary string. An input-tuple of n integers $(x_1, x_2, \ldots, x_n)$ is mapped to a single string by the recursive bijective pairing function. The basic function $\langle ., . \rangle : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ assigns the integer $\langle x, y \rangle$ to the pair (x, y). A simple realization of such a function would be

$$\langle x, y \rangle = \overline{x}\, y, \tag{4.1}$$

where $\overline{x} = 1^{l(x)}0x$. In this case we use the binary representation of the variables x and y and, thus, $\overline{x}\, y$ and $1^{l(x)}0x$ mean that the corresponding strings are concatenated together. To extend this function to n-tuples of integers we use the iterative definition

$$\langle x_1, ..., x_n \rangle = \langle x_1, \langle x_2, ..., x_n \rangle \rangle$$

for all $x_i \in \mathbb{N}$, $1 \leq i \leq n$.

The calculation stops as soon as the TM holds, but it may also occur that this situation is never achieved. After the machine held, the output of the computation is determined by the integer representation of the maximal binary string on the tape framed by blanc characters.

Definition 4.1 (Computability)
Under this convention for inputs and outputs, each Turing machine defines a partial function from n-tuples of integers onto the integers, $n \geq 1$. We call such a function partial recursive *or* computable. *If the function computed is defined for all arguments and the Turing machine halts for all inputs, then we call it* total recursive *or simply* recursive.

Remark 4.2 *In the theory of computability, a function is called* total *if it is defined for all elements of a particular set.*

This means that the class of partial recursive functions represents all those that are computable by a TM.

We write $T(x) = z$ to indicate that the Turing machine T produces the output z on the given input x. Likewise, if the partial recursive function ν represents T, we write $\nu(x) = z$. For convenience we will sometimes use T and ν equivalently, like when we talk about the Turing machine ν.

4.1.2 Universal Turing Machine

A special case of a TM is the universal Turing machines (UTM), which can simulate any Turing machine. A UTM can be realized by effectively enumerating the set of Turing machines. To simulate the n'th Turing machine T_n we only have to pass the index n to the universal Turing machine U. By using the index n, the UTM determines the specific Turing machine T_n, including its corresponding list of rules, and calculates

$$U(x, n) = T_n(x)$$

for every finite binary sequence x. By an effective enumeration we mean a computable bijective function from the set of TMs onto the natural numbers. A simple enumeration would be to uniquely encode the finite list of rules for each TM into a binary sequence, and then order them lexicographically. We then map each TM to the index of its binary representation. In the further section we assume that there exists such an enumeration of the Turing machines and that we are able to identify a specific TM by its index.

With the help of UTMs we can define the class of universal partial recursive functions.

Definition 4.3 (Universal Partial Recursive Function)
The partial recursive function $\nu(i, x)$ computed by the universal Turing machine U is called a universal partial recursive function.

4.1.3 Complexity

Now we have all the necessary ingredients to describe the complexity of a binary sequence. At first, we define the complexity corresponding to an arbitrary Turing machine. After that we specify the definition to the class of UTMs and will show that the choice of the UTM is irrelevant.

The conditional complexity $C(x|y)$ of a sequence is the complexity of x if another sequence y is already known. In this case, y stands for some additional information about x, like the length of the sequence.

Definition 4.4 (Conditional Complexity C_ϕ)
Let x, y, p be natural numbers. Any partial recursive function ϕ, together with p and y, such that $\phi(\langle p, y\rangle) = x$ is a description of x. The complexity C_ϕ of x conditional *to y is defined by*

$$C_\phi(x|y) = \min\{l(p) : \phi(\langle p, y\rangle) = x\},$$

and $C_\phi(x|y) = \infty$ if there are no such p. We call p a program *to compute x by ϕ, given y.*

Let T be the corresponding Turing machine to ϕ. Then the conditional complexity of the sequence x, if y is known, is defined by the length of the shortest program, which together with y calculates x on the Turing machine T. The length of the shortest program can vary from machine to machine. Some machines may favor specific sequences and assigns them much smaller complexities. For example for a TM that always outputs the constant sequence x, the corresponding complexity of x will be 0 independent of the structure of x. Intuitively the complexity of a sequence should be a property of the sequence itself and, therefore, invariant to the choice of the TM. The next theorem shows that by using a universal Turing machine we can guarantee that the complexity is additively optimal, which means that the complexity calculated by a UTM is shortest, except of an additive constant. This condition is also called *universality*.

Theorem 4.5 (Universality of Conditional Complexity)
Let ϕ_0 be a universal partial recursive function, thus it is computable by a universal Turing machine U, then for any other partial recursive function ϕ

$$C_{\phi_0}(x|y) \leq C_\phi(x|y) + c_\phi$$

for all finite binary sequences x and y.

Proof. Intuitively this theorem is true, since the UTM ϕ_0 needs only a constant number of bits to simulate the Turing machine ϕ.

Let ϕ_0 be the UTM that on input $\langle n, p, y\rangle$ simulates the n'th Turing machine with input $\langle p, y\rangle$, which means that $\phi_0(\langle n, p, y\rangle) = \phi_n(\langle p, y\rangle)$ for all pairs (p, y). Let ϕ be the n'th Turing machine. For every program p it is satisfied that if $\phi(\langle p, y\rangle)$ generates x, then

$\phi_0(\langle n, p, y\rangle)$ does this as well. Let us denote the constant difference $l(\langle n, p, y\rangle) - l(\langle p, y\rangle)$ by c_ϕ. Thus,

$$C_{\phi_0}(x|y) \leq C_\phi(x|y) + c_\phi$$

for all pairs (x, y). If we realized $\langle x, y\rangle$ by $\overline{x}\, y$ like in (4.1) then $c_\phi = 2l(n) + 1$. □

From Theorem 4.5 it follows that the complexity of a finite binary sequence x calculated by a universal Turing machine ϕ_0 is always finite. There always exist a Turing machine ϕ that calculates x by a finite program, like the machine that passes the first input variable directly to the output. Therefore,

$$\begin{aligned} C_{\phi_0}(x|y) &\leq C_\phi(x|y) + c_1 \\ &\leq l(x) + c_2 \\ &< \infty. \end{aligned}$$

The theorem also implies that the complexity of two different UTMs differs by at most a constant factor, which depends only on the two machines. Thus, if ψ and ψ' are universal Turing machines, then for all x and y holds

$$|C_\psi(x|y) - C_{\psi'}(x, y)| \leq c_{\psi,\psi'}. \tag{4.2}$$

We can therefore determine any arbitrary UTM ϕ for the final definition of the Kolmogorov complexity. The complexity according to ϕ will then be minimal for all sequences compared to other Turing machines except of an additive constant.

Definition 4.6 ((Conditional) Kolmogorov Complexity)
Fix a universal partial recursive function ϕ_0 and dispense with the subscript by defining the conditional Kolmogorov complexity $C(.|.)$ *by*

$$C(x|y) = C_{\phi_0}(x|y).$$

This particular ϕ_0 is called the reference function for C. We also fix a particular Turing machine U that computes ϕ_0 and call U the reference machine. *The* unconditional Kolmogorov complexity $C(.)$ *is defined by*

$$C(x) = C(x|\epsilon).$$

4.1.4 Incompressible Sequences

By means of the Kolmogorov complexity we are now able to define incompressible sequences.

Definition 4.7 (c-Incompressible)
For each constant c we say a string x is c-incompressible *if*

$$C(x) \geq l(x) - c.$$

This condition is often used for defining randomness. We will show that incompressibility is equivalent to a notion of randomness that is defined by statistical tests. Thus, incompressibility is equivalent to randomness and high Kolmogorov complexity.

Remark 4.8 *In contrast to Chapter 5, this time we do not limit ourselves to tests running in polynomial time.*

4.1.5 Martin-Löf Test of Randomness

In this section we introduce a definition for statistical tests, which we are going to use for determining the randomness of a binary sequence.

Let S be a sample space with probability distribution P, and let $\epsilon > 0$ denote the *level of significance* of the test. A *majority* $M \subseteq S$ is an arbitrary set such that $1 - P(M) \leq \epsilon$. We denote an element $x \in S$ as a *typical outcome* of S if x belongs to any majority M. If we choose x at random out of S, then with high probability x will lie in the intersection of all majorities. Later (see Definition 4.14), we will associate this property with randomness. To prove this condition we apply tests that check the hypothesis "x belongs to majority M in S".

A *test* is nothing else but a rule that determines, for any level of significance ϵ, for which elements we reject the hypothesis "x belongs to a majority M in S". Let us assume that $\epsilon = 2^{-m}$ for $m \in \mathbb{N}$. Then we can define a test by a set $V \subseteq \mathbb{N} \times S$, such that $(m, x) \in V$ if and only if the hypothesis is rejected for x with level of significance $\epsilon = 2^{-m}$. By means of V we can define a nested sequence $(V_m)_{m \geq 1}$, $V_m \supseteq V_{m+1}$, $\quad m = 1, 2, \ldots$, of sets by

$$V_m = \{x : (m, x) \in V\}.$$

The condition that x is rejected with level of significance ϵ is expressed by

$$\sum_x \{P(x) : l(x) = n, x \in V_m\} \leq \epsilon. \tag{4.3}$$

This characteristic means that the probability of the set of elements of length n which are rejected is less than ϵ for any arbitrary $n \in \mathbb{N}$. V_m is called the *critical region* on the level of significance ϵ. If $x \in V_m$, then the hypotheses "x belongs to majority M in S" is rejected for x with level of significance ϵ.

Statistical tests try to avoid the case that elements get falsely rejected. Such a case is called *type I error*. Condition (4.3) guarantees that the probability of rejection and consequently the probability of false rejection is less than the level of significance ϵ.

Subsequently, we assume that all statistical tests used in practice are computable in terms of Definition 4.1 and are, therefore, partial recursive. Other types of tests do not appear to be useful.

For Definition 4.10 we need the notion of recursive enumerable sets.

Definition 4.9 (Recursive Enumerable)
A set A is recursively enumerable *if it is empty or the range of some total recursive function f. In this case we say that f* enumerates *A.*

We are now able to give a formal definition of a statistical test.

Definition 4.10 (P-Test)
*Let P be a recursive probability distribution, i.e. computable by a recursive function, on the sample space $\mathbb{N}$. A total function $\delta : \mathbb{N} \to \mathbb{N}$ is a P-*test *(Martin-Löf test for randomness) if*

1. *δ is enumerable (the set $V = \{(m, x) : \delta(x) \geq m\}$ is recursively enumerable); and*
2. *$\sum\{P(x) : \delta(x) \geq m, l(x) = n\} \leq 2^{-m}$, for all n.*

The critical regions are given by

$$V_m = \{x : \delta(x) \geq m\}.$$

Each test corresponds to a specific probability distribution P. In the same way as for Turing machines we define a universal P-test, which is able to simulate all other P-tests.

Definition 4.11 (Universal P-Test)
A universal Martin-Löf test for randomness with respect to the probability distribution P, a universal P-test *for short, is a test $\delta_0(.|P)$ such that for each P-test δ, there is a constant c, such that for all x, we have $\delta_0(x|P) \geq \delta(x) - c$.*

Thus, if x passes the universal test δ_0, which means the hypothesis "x belongs to a majority M in S" is not rejected for x, then it also passes every other P-test, except for a change of the level of significance. That is, if

$$\begin{aligned} V &= \{(m, x) : \delta(x) \geq m\}; \text{ and} \\ U &= \{(m, x) : \delta_0(x|P) \geq m\} \end{aligned}$$

then,

$$V_{m+c} \subseteq U_m \qquad m = 1, 2, ...$$

where c only depends on V and U and not on m.

Theorem 4.12 *For every distribution P there exists a universal P-test $\delta_0(.|P)$.*

The proof of this theorem can be found in [LV93, p. 109 ff.].

In the following definition we consider the special class of universal P-tests for the uniform distribution L. This distribution is given by $L(x) = 2^{-2l(x)}$ for all $x \in \mathbb{N}$. If we restrict the length of the elements to $l(x) = n$ then we use $L_n = 2^{-n}$.

Theorem 4.13 (Universal L-test)
The function $f(x) = l(x) - C(x|l(x)) - 1$ is a universal L-test where L is the uniform distribution.

This theorem as well as its proof can be found at [LV93, p. 110 ff.].

Due to the universality of the test, every x that appears random by this test, i.e., which was not rejected, will be considered random by any other L-test, except for a change of the level of significance. Thus, we are able to define randomness for a specific element $x \in \mathbb{N}$.

Definition 4.14 (c-Random)
Let us fix $\delta_0(x) = l(x) - C(x|l(x)) - 1$ as the reference universal test with respect to the uniform distribution. (In our previous notation, $\delta_0(.) \equiv \delta_0(.|L)$, where L is the uniform distribution.) A string x is called c-random*, if*

$$\delta_0(x) \leq c.$$

Let us assume that the element x is c-random and that we use the recursive bijective pairing function $\langle ., . \rangle$ in (4.1). Then

$$C(x|l(x)) \leq C(x) \leq C(x|l(x)) + 2C\left(l(x) - C(x|l(x))\right),$$

except for an constant factor, because $l(x)$ can be reconstructed from the shortest program p to produce x and the value $l(p) - l(x) = C(x|l(x)) - l(x)$. Since x is c-random, $l(x) - C(x|l(x)) \leq c$, and thus

$$C(x|l(x)) \leq C(x) \leq C(x|l(x)) + \log_2(c).$$

Remark 4.15 *If the element x is c-random, then*

$$C(x|l(x)) = C(x),$$

except of an additive constant.

However, with compressible elements, like $x = 1^n$, the two quantities may differ up to $\log_2(l(x))$ bits.

Consequently, the definition of incompressibility and randomness are comparable, if x is c-random. Let us assume that x is c_1-random, then there exists a constant c_2, such that x is c_2-incompressible and conversely if x is c_1-incompressible, then there exist a constant c_2, such that x is c_2-random.

In Chapter 5 we give another definition of randomness which is related to statistical tests. In contrast to the definition in Chapter 5, in this chapter we allow all statistical tests without considering their efficiency. Chapter 5 only allows those tests that run in polynomial time, which limits the number of tolerated tests and therefore increases the number of strings that are considered "random". Consequently, it is possible that in Chapter 5 we consider a string as *pseudorandom*, even if it was generated from a shorter sequence and, thus, only has the low Kolmogorov complexity of the short sequence.

4.2 Kolmogorov Complexity and Shannon's Entropy

Kolmogorov complexity as well as Shannon's entropy are measures of the number of bits that are necessary to identify a specific string. Kolmogorov complexity is a property of the string itself, whereas entropy is a property of the source from which the string was chosen. Thus, Kolmogorov complexity has the advantage that it is also defined even if we do not know the probability distribution of the source. However, in practice it has the drawback that it is not computable (see Section 4.3).

Both definitions are related even if they may vary widely in some specific cases. On the one hand, let us assume that a source produces only a small number of different output strings. Then the entropy of the source will only be a few bits, independently of the (probably high) Kolmogorov complexity of each individual string. On the other hand, let us assume that a source produces all binary string of length n with uniform probability 2^{-n}. In this case, the source has the maximal entropy of n bits. Still, it will produce the string consisting of n zeros, which has a very low Kolmogorov complexity.

There exists a direct connection between the two quantities. Cover and Thomas show in [CT91, p. 154] that the expected conditional Kolmogorov complexity of n consecutive elements taken from a source with entropy $H(X)$ converges towards n times the entropy of the source, for n approaching infinity.

4.3 Computability of Kolmogorov Complexity

Kolmogorov complexity is an important concept to define the randomness of a individual string, but is it actually computable? Unfortunately, the answer to this question is negative as we shall see in the next theorem.

Theorem 4.16 *The function $C(x)$ is not partial recursive. Moreover, no partial recursive function $\phi(x)$, defined on an infinite set of points, can coincide with $C(x)$ over the whole of its domain of definition.*

However, although $C(x)$ is not computable, it can be approximated by a computable function.

Theorem 4.17 *There is a total recursive function $\phi(t, x)$, monotonically decreasing in t, such that*

$$\lim_{t \to \infty} \phi(t, x) = C(x).$$

Both theorems as well as their proofs can be found in [LV93, p. 103].

For an upper bound of Kolmogorov complexity, we may apply any lossless compression algorithm. If a string x can be compressed to a string x' such that x is reconstructible from x', then the Kolmogorov complexity is certainly less or equal to $l(x')$. A famous lossless

compression algorithm that can be used for approximating the Kolmogorov complexity is the Lempel-Ziv algorithm.

The basic idea of the this algorithm is that a binary sequence $S = s_1 s_2 ... s_n$ of length n is partitioned into subsequences, such that the subsequence $s_{i+1}...s_j$, $1 \leq i < j \leq n$, can be generated from its prefix $s_1...s_i$ plus an additional character a. To produce the subsequence $s_{i+1}...s_j$ from its prefix, we need a position k, $1 \leq k \leq i$ and a symbol a, such that

$$\begin{aligned} s_{i+1+r} &= s_{k+r}, \text{ for } 0 \leq r \leq j-i-2, \text{ and} \\ s_j &= a. \end{aligned}$$

Consequently, it is sufficient to know k and a to generate the subsequence $s_{i+1}...s_j$ from its prefix. The whole sequence S is then reconstructed by producing the single subsequences and concatenating them together. Let S_i be the i'th subsequence $s_{h_{i-1}-1}...s_{h_i}$, determined by the indices h_i and h_{i-1}, $0 \leq i \leq m$, then $S = S_1...S_m$ with $1 = h_1 < h_2... < h_m = n$ and $h_0 = 0$. The compressed file only contains the values k and a for every subsequence. Thus, the minimal number of needed subsequences is a measure of the length of the compressed file and, therefore, an upper bound for the Kolmogorov complexity. Lempel and Ziv state in [LZ76] that the minimal number of subsequences can be used as measure for the complexity of a sequence.

However, we have to consider that, like for every compression algorithm, there exist worst case strings that are not compressible by the algorithm, but have a Kolmogorov complexity that is clearly less than the length of the sequence. Let us consider the 218 bit string that was constructed by concatenating the first 52 strings of the sequence $0, 1, 00, 01, 10, 11, 000, ...$ The authors of [EHS02] state that if this string gets "compressed" by LZ78, an implementation of the Lempel-Ziv algorithm, then the result of the compression needs 318 bits which is much more than the Kolmogorov complexity of the string.

4.4 Kolmogorov Complexity and Cryptographic RNGs

The Kolmogorov complexity is defined for individual binary strings and is therefore independent of the probability distribution of their source. Thus, the definition is even meaningful for random number generators for which the distribution of the output is unknown because, for example, they process unpredictable external input.

However, the Kolmogorov complexity has the weakness that, in general, it is not computable. Consequently, we cannot use it directly but must employ approximations. The Lempel-Ziv algorithm is a possible way of approximating the Kolmogorov complexity, although, like all compression algorithms, it provides only an upper bound for the complexity.

Additionally, there are situations in which it is an drawback that the Kolmogorov complexity does not take the distribution of the elements into account. Let as assume that a random number generator produces only a small variety of different binary sequences of high Kolmogorov complexity. If an adversary knows about the limited number of outputs he or she can easily guess the correct value independently of the Kolmogorov complexity. Thus, if the adversary has an idea of the distribution of the output it is more reasonable to apply the entropy of the generator, which uses this information, to judge the randomness of a sequence.

Generally, if we have no idea about the distribution of the information source and are able to provide an adequate estimator, then the Kolmogorov complexity is a good indicators of the randomness of an individual sequence.

Chapter 5

Polynomial-Time Indistinguishability

In the previous chapter we have defined the randomness of a binary sequence by means of statistical tests. We only considered the result of the tests but not how long it takes to compute them. In this chapter we limit our discussion to the class of "efficient" algorithms, which means that they work in polynomial-time of the length of the input.

A main problem in the evaluation of cryptographic random number generators is that we are not able to make any statements about the methods that will be applied during an attack. However, we may assume that the adversary possesses a limited amount of computational resources and is, thus, only able to apply efficient algorithms. Therefore, we denote a sequence as "pseudorandom" if its not feasible for such an algorithm to distinguish the sequence from "true random numbers". This means we are satisfied with sequences that "look" random for any efficient attack. By *true random numbers* we mean sequences that are chosen according to a uniform distribution.

This approach is described in detail by Goldreich in [Gol99]. A more mathematical description, to which we refer in this chapter, can be found in [Gol95].

5.1 Definitions and Characteristics

In order to describe Goldreich's definition of pseudorandomness we have to specify some terms we are going to use in the remaining chapter. The output of a generator is said to be *pseudorandom* if it is indistinguishable from true random numbers by any efficient algorithm.

We call an algorithm *efficient* if it corresponds to a probabilistic polynomial-time Turing machine (PTM). A *probabilistic Turing machine* M is a TM that, in addition to the basic rules, can "flip coins" to decide where to go next. Referring to the definition of a TM in Chapter 4, this means that for a pair (p, s) of current state and scanned symbol, there may exist two rules with different pairs (a, q) and (a', q') of the symbol to write and the command to perform. Which rule is applied during a calculation is determined by an internal coin flip. $M(x)$ represents the probability space that is produced by the constant input x and all possible internal coin flips. $M(x, y)$ denotes the output of the TM on input

x and the coin flips y. *Polynomial-time* means that the TM needs at most $p(|x|)$ steps for the calculation on input x, where $p(.)$ is a positive polynomial and $|x|$ the length of the input.

Remark 5.1 (Notation)
Here, we specify some more phrases that we will employ in the following definitions.

- *An* ensemble $X = \{X_n\}_{n\in\mathbb{N}}$ *is a sequence of random variables* X_n.
- *Let us assume that* A *is a probabilistic polynomial-time TM. Then* $Pr(A(X_n, 1^n) = 1)$ *is the probability that the TM outputs* 1 *on input* $(x, 1^n)$ *if* x *was chosen according to the probability distribution of* X_n.
- U_m *represents the uniform distribution over* $\{0,1\}^m$.
- $\{U_n\}_{n\in\mathbb{N}}$ *is called the standard uniform ensemble. However, it is convenient, with regard to Definition 5.4, to also call ensembles of the form* $\{U_{l(n)}\}_{n\in\mathbb{N}}$ *uniform, where* $l : \mathbb{N} \to \mathbb{N}$ *is a function on natural numbers.*

We are now able to describe the situation that two ensembles cannot be told apart by any efficient algorithm. We then claim that the two ensembles are computationally indistinguishable or indistinguishable in polynomial-time.

Definition 5.2 (Polynomial-Time Indistinguishability)
Two probability ensembles, $X := \{X_n\}_{n\in\mathbb{N}}$ *and* $Y := \{Y_n\}_{n\in\mathbb{N}}$, *are* indistinguishable in polynomial time *if for every probabilistic polynomial-time algorithm* A, *for any polynomial* p *and for all sufficiently large* n

$$|Pr(A(X_n, 1^n) = 1) - Pr(A(Y_n, 1^n) = 1)| < \frac{1}{p(n)}. \tag{5.1}$$

The second input to A *specifies the length of the elements in* X_n *and* Y_n, *respectively. In this case both random variables are defined over elements of length* n. *The probability is taken over* $\{X_n\}_{n\in\mathbb{N}}$ *(respectively* $\{Y_n\}_{n\in\mathbb{N}}$*) as well as over the internal coin tosses of the algorithm* A.

An ensemble is said to be pseudorandom if it is computationally indistinguishable from true random numbers, which are represented by an ensemble of uniformly distributed variables.

Definition 5.3 (Pseudorandom Ensemble)
Let $U := \{U_{l(n)}\}_{n\in\mathbb{N}}$ *be a uniform ensemble, where* l *is a function on natural numbers. The ensemble* $X := \{X_n\}_{n\in\mathbb{N}}$ *is called* pseudorandom *if* X *and* U *are indistinguishable in polynomial-time.*

In the next step we present a definition of a pseudorandom generator, which differs from the one we will use in Chapter 7. The definition in Chapter 7 describes the structure of a

generator independently of the quality of the output. Goldreich defines a pseudorandom generator as an algorithm G that extends an input x into an output $G(x)$ with $|x| < |G(x)|$, and, furthermore, the output must be pseudorandom according to Definition 5.3 if the input is uniformly distributed.

Definition 5.4 (Pseudorandom Generators)
A pseudorandom generator *is a deterministic polynomial-time algorithm, G, satisfying the following two conditions:*

1. expansion*: there exists a function $l : \mathbb{N} \to \mathbb{N}$, such that $l(n) > n$ for all $n \in \mathbb{N}$ and $|G(s)| = l(|s|)$ for all $s \in \{0,1\}^*$. This means that for all $s \in \{0,1\}^*$ of length $|s|$, the generator produces a binary string $G(s)$ of length $l(|s|)$.*

2. pseudorandomness (as above)*: the ensemble $\{G(U_n)\}_{n\in\mathbb{N}}$ is pseudorandom, which means that for any probabilistic polynomial-time algorithm A, for any positive polynomial p, and for all sufficiently large n,*

$$|Pr(A(G(U_n), 1^{l(n)})) = 1) - Pr(A(U_{l(n)}, 1^{l(n)}) = 1)| < \frac{1}{p(n)}. \tag{5.2}$$

The probabilistic polynomial-time algorithm A can be seen as a polynomial-time *statistical test*. If there exists an algorithm A such that condition (5.2) does not hold, we say the generator G failed the test A. Therefore, Property 2 in Definition 5.4 is sometimes described as the fact that G *passes all polynomial-time statistical tests*.

From a pseudorandom generator we expect that an observer who knows i bits of the output is not able to predict the $(i+1)$-th bit effectively with a probability significantly greater than $\frac{1}{2}$. This property is called *(next-bit) unpredictability*. We will see that our definition of a pseudorandom generator meets this requirement.

Definition 5.5 ((Next-Bit) Unpredictable)
An ensemble $\{X_n\}_{n\in\mathbb{N}}$ is called unpredictable in polynomial-time *if for every probabilistic polynomial-time algorithm A and every positive p and for all sufficiently large n,*

$$Pr(A(X_n, 1^n) = next_A(X_n, 1^n)) < \frac{1}{2} + \frac{1}{p(n)}, \tag{5.3}$$

where $next_A(x)$ returns the $(i+1)$-th bit of x if A, on input $(x, 1^n)$, reads only $i < |x|$ of the bits of x, and returns a uniformly chosen bit otherwise (i.e., in case A reads the entire string x).

Remark 5.6 *Goldreich states in [Gol95, p. 90] that a probability ensemble $\{X_n\}_{n\in\mathbb{N}}$ is pseudorandom if and only if it is (next-bit) unpredictable.*

As a consequence, the output of a pseudorandom generator is (next-bit) unpredictable.

5.1.1 Existence of Pseudorandom Generators

After having defined pseudorandom generators, the question remains if generators with such properties even exist. The existence cannot be shown directly. However, under unproven but widely believed assumptions like the intractability of factoring large integers we can claim the existence. The Blum-Blum-Shub generator in Chapter 11 is one of the few generators that are proven to be pseudorandom under such assumptions. A further theorem shows the connection between pseudorandom generators and one way functions.

Theorem 5.7 (On the existence of pseudorandom generators)
Pseudorandom generators exist if and only if one-way functions exists.

The proof of this theorem can be found in [Gol95, p. 90 and p. 106].

One-way functions are a class of important cryptographic primitives. They are easy to compute but hard to invert.

Definition 5.8 (One-Way Functions)
A function $f : \{0,1\}^* \to \{0,1\}^*$ *is called* one-way *if the following two conditions hold*

1. "easy to compute"*: There exists a (deterministic) polynomial-time algorithm, A, such that on input x algorithm A outputs* $f(x)$ *(i.e.,* $A(x) = f(x)$*).*

2. "hard to invert"*: For every probabilistic polynomial-time algorithm,* A'*, every polynomial* $p(.)$*, and all sufficiently large* n*,*

$$Pr(A'(f(U_n), 1^n) \in f^{-1}f(U_n)) < \frac{1}{p(n)}.$$

Ideal hash function as described in Section 3.2.4 are examples of one-way functions.

5.2 Polynomial-time Statistical Tests and Randomness

For the definition of pseudorandomness we limited our discussion to class of polynomial-time statistical test with the justification that this family includes all efficient methods which are realizable for larger input sizes n. Many conventional tests of randomness fall into this category (see [Knu98, Chapter 3.3] for examples).

However, there also exist important tests for verifying the randomness of a sequence of binary strings, which cannot be implemented in polynomial-time. The probably most famous one is the *spectral test*, which measures the correlation between overlapping n-tuples in the output of a RNG. This test is designed for deterministic generators (see Section 7.1.1) with periodic output and a lattice structure. Let us assume that the RNG outputs the sequence $x_1, x_2, \ldots, x_N$ with period length N. Without loss of generality we

suppose that $x_i \in [0,1]$, $1 \leq i \leq N$. For this test, we consider the overlapping n-tuples $(x_i, x_{i+1}, \ldots, x_{i+n})$, $1 \leq i \leq N$. Each tuple represents a point in the n-dimensional unit cube. The test determines the maximal distance between adjacent hyperplanes, out of the family of all parallel hyperplanes that cover every n-tuple. The smaller the distance is, the more regular the points are distributed in the n-dimensional space and the less correlated are particular points. The test requires the shortest vector algorithm, which cannot be implemented in polynomial time. Fincke and Phost introduced in [FP85] an algorithm to calculate the shortest vector and showed that the problem is NP-hard. However, an approximation of the shortest vector can be obtained by the LLL-algorithm which runs in polynomial-time [LLL82].

5.3 Polynomial-time and Efficiency

In computational theory an algorithm is called efficient if it runs in polynomial-time. However, is such an algorithm always the fastest choice? The condition guarantees that even in worst case, the number of calculation steps does not grow too fast with increasing input length. Still it may happen that for small input sizes a non-polynomial-time algorithm runs faster than a polynomial one, especially if the latter is bounded by a polynomial n^c with large constant c. Furthermore, in practice average-case time complexity is more important than the worst case one. An algorithm that has a non-polynomial-time worst-case complexity but runs in polynomial-time in the average-case may still provide an adequate performance. Consequently, polynomial-time algorithms are not always the fastest choice, especially if the input size is limited.

5.4 Kolmogorov Complexity versus Computational Indistinguishability

The notion of randomness corresponding to Kolmogorov complexity, as well as the notion of pseudorandomness in this chapter use statistical tests in their definitions. The first concept is defined for a single sequence, whereas the latter one is defined for probability ensembles. Consequently, the second one judges the whole behavior of a random source and not just a single output.

Chapter 5 considers only those statistical tests that run in polynomial-time, thus the number of possible tests is smaller than in the previous chapter and, therefore, the condition to pass all tests is weaker. For this reason, it is possible that sequences that are generated from shorter seeds are pseudorandom although they are not random with respect to Kolmogorov complexity.

We should not forget that the whole output of a pseudorandom generator, according to the definition in this chapter, depends on a relatively short seed. Thus, the security of the generator is limited to the length of the seed and the output is vulnerable to theoretical

brute-force attacks (trying out all possibilities) on the seed. Since the number of attempts during such an attack grows exponentially with the length of the seed, this case does not play an important role in practice if the size of seed is chosen large enough.

5.5 Pseudorandomness and Cryptographic PRNGs

The definition of pseudorandomness is based on the requirement that the output of a good RNG should be indistinguishable from uniformly distributed random variables. The problem of this definition is that polynomial-time statistical tests are not sufficient, since there exist desirable properties for random numbers that can only be checked by non-polynomial-time tests, like the spectral test. For linear generators, the results of (polynomial-time) statistical tests regarding a standard battery of tests like DIEHARD [Mar95], may be satisfactory, but still the generator may fail the spectral test Thus, the concept of polynomial-time indistinguishability is indeed of theoretical importance, but not very usable in practice.

Furthermore pseudorandomness is exclusively defined for completely deterministic RNGs. A generator that processes unpredictable external input is not covered by this definition, since normally the probability distribution of the output is unknown and does not depend entirely on the initial seed.

Still, the output of generators, which are proven to be pseudorandom at least guarantee a minimum of “randomness”. However, since these generators are quite slow, they are seldom used in practice.

Chapter 6

Summary of Part I

In the previous chapters we studied three different concepts of randomness. Each of them has its advantages and its drawbacks. Let us assume that we apply those methods to evaluate the output of a random number generator.

Entropy as well as pseudorandomness are defined for sequences of random variables, whereas Kolmogorov complexity is defined for individual strings. In the case that we or an adversary has any information about the probability distribution of the output, we would prefer Shannon's entropy, since it, in contrast to Kolmogorov complexity, entropy employs the whole information. However, if the distribution of the output is unknown we have to apply entropy estimators. Generally, entropy is a very practical term for describing the "mixing" operations within a RNG.

In return, the Kolmogorov complexity is defined for individual strings and is thus applicable if we do not know the distribution law of the source. However, since the complexity is not computable in general we are forced in practice to employ approximations.

The definition of pseudorandomness is based on the fact that a sequence of numbers is not random if and only if this statement follows from a statistical test. In our case we limit our considerations on polynomial-time statistical test due to the fact that those tests may still be done efficiently, even for larger input sizes n. Since not all important tests of randomness can be realized in polynomial-time this definition is only partly meaningful.

Generally speaking, in all three cases, we are often unable to employ the definitions directly but have to rely on approximations or estimators, especially if some basic conditions, like the distribution of external input, are unknown.

Part II

Practical Generators

Chapter 7

A Selection of Cryptographic Randomnumber Generators

This part deals with real-world random number generators for cryptographic applications. The first chapter gives a short introduction to the structure and the most common problems encountered with such generators. Additionally, we shall introduce a general scheme to describe and compare the different RNGs. In the following chapter we discuss several possible attacks on random number generators. The aspect of attacks is the main difference between RNGs for stochastic simulations and for cryptographic applications. In the last chapters we will study five different generators, their structure, their resistance against attacks, and their behavior in practice.

7.1 Three Categories

The random number generators which we are going to discuss in the following may be divided into three different categories, namely pseudorandom number generators, entropy gathering generators, and hybrid generators.

7.1.1 Pseudorandom Number Generators

This type of RNGs is referred to as *pseudorandom number generators* (PRNG), since their output satisfies many conditions that we expect from random numbers, although the generator operates in a deterministic way. Linear Congruential Generators (LCG) are a simple example of a PRNGs:

$$X_{n+1} = (a \cdot X_n + b) \pmod{m} \tag{7.1}$$

Once the parameters a, b and m are chosen, the sequence of generated pseudorandom numbers $(X_n)_{n\geq 1}$ will depend only on the initial value X_0. For example, the Unix *rand* generator is implemented by a LCG with parameters $a = 1103515245$, $b = 12345$ and $m = 2^{31}$. The characteristic of PRNGs is that they produce a long sequence of

random numbers from a short initial input, the so-called *seed*, by means of a completely deterministic algorithm. A mathematical discussion of this class of generators can be found in [L'E94] and a slightly modified, more recent version in [L'E04]. In this category we also find BBS and AES, which will be presented in Chapters 11 and 12.

The security analysis of such an algorithms in a cryptographic context is not restricted to a statistical analysis of the output of the generator, but we may also study the algorithm itself as well as the reseeding procedure. This analysis allows us to make statements about properties like periodicity, next-bit predictability (Definition 5.5), or correlations within the generated sequence of pseudorandom numbers. Many of these statements assume that the seed of the generator has been chosen at random, i.e. it is a realization of a sequence of i.i.d uniform random variables. In practice, this assumption implies that we are able to provide high quality random numbers for seeding. PRNGs may be used when a *slow* source of high quality random numbers (like true random number generators, see Section 7.1.2) is available and a large amount of random numbers is required.

However, the main problem of such generators is that as long as they use the same seed, they are completely vulnerable to *state compromise attacks* (see Section 8.3). If an adversary is able to guess the current state of the generator, he or she will be able to calculate all future random numbers and often even the previous ones.

Another dangerous situation may occur, if we `fork` a process that holds the PRNG. A `fork` creates a copy of the process including the values of all internal variables. We will illustrate the problem by a simple server application:

Let us consider a process $\mathbf{P}$ within a server that answers customer requests. In order to provide secure communication, the process uses a PRNG to create session keys for encryption. The seed of the generator is symbolized by V_1 and all subsequent states by V_n, $n > 1$. The answer mechanism of the process causes a `fork` of $\mathbf{P}$, which creates an identical copy $\mathbf{P}'$ of the main process $\mathbf{P}$. $\mathbf{P}'$ contains a copy of all variables in $\mathbf{P}$ including their content at the time of the `fork`. Now let us assume that two customer try to access the server at the same time. The answer mechanism will produce two identical copies $\mathbf{P}'$ and $\mathbf{P}''$ of the main process (Figure 7.1), which each holds exact copies of the seed $V_1' = V_1'' = V_1$. Thus, both processes will create exactly the same session keys. To solve

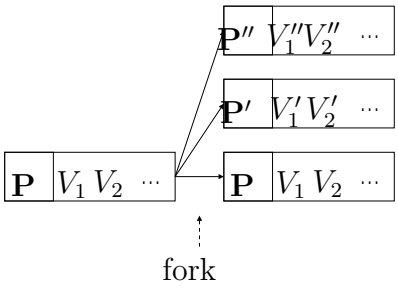

Figure 7.1: The fork command

this problem, we need to reseed the generator after each `fork` of the process. A common

but wrong solution would be to reseed the generator depending on the system time. The main problem with this method is that if the two processes $\mathbf{P}'$ and $\mathbf{P}''$ are generated shortly after each other, then due to a probably too low resolution of the clock the two process may be reseeded with the same time.

The `fork` feature is part of the POSIX (Portable Operating System Interface) standard [POS05] and is therefore implemented by any operating system that supports this standard, like Unix, Linux, or Microsoft Windows. Thus, this feature is a common pitfall in the use of PRNGs for cryptographic applications on all computing platforms. If we want to employ this kind of generators we have to be able to provide good entropy sources for the reseeding.

7.1.2 Entropy Gathering Generators

The second kind of random number generators processes input from different noise sources. At first, it extracts and stores data from the source, then it tries to estimate the entropy of the input, and limits the number of output bits to the amount of entropy that has been gathered. We may refer to this type as *entropy gathering generators*. The generator /dev/random (Chapter 9) is one example of this kind, but also every true random number generator (TRNG) falls into this category. By a TRNG we mean a generator which uses physical effects, like noise of semiconductor elements or radioactive decay, to collect entropy. The quality of entropy gathering generators depends highly on the selection of the

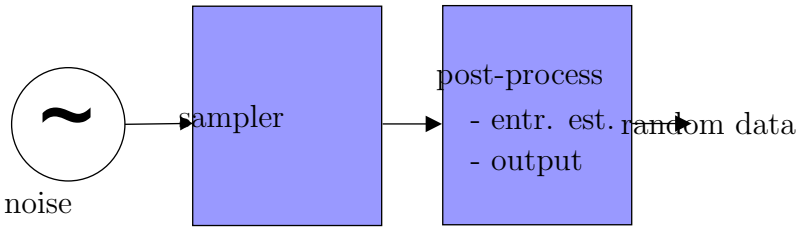

Figure 7.2: Structure of a TRNG

noise sources and on the adequateness of the entropy estimation. If an adversary is able to influence or to observe one of the sources used, he or she may gain information about the output of the generator. Of course we would like to prevent any attack that allows an adversary to obtain information about the inner state of the RNG. Let us consider two possible kinds of attacks:

- *passive attacks*: The adversary observes the noise source and part of the output of the generator. If he or she is able to discover any correlation between those two streams it may be possible to use this knowledge to predict future output.

- *active attack*: The adversary is able to influence the noise source and is thus able to introduce a bias into the generated sequence. This kind of attack is particularly

effective, since the adversary may try to adapt the input noise to the specific properties of the generator and forces the RNG to produce highly predictable output.

A more detailed description of possible attacks can be found in Chapter 8. To minimize the power of such attacks, generators often use cryptographic primitives like hash functions or block ciphers for mixing the input or for masking the inner state against the output. The security of the generator thus depends on the security assumptions of the primitives used. Block ciphers are also used to stretch the amount of generated random numbers. Therefore, generators which employ block ciphers often fall into the third category (see below). However, it is common practice to use cryptographic hash functions like SHA-1 (Secure Hash Algorithm [Nat02]) or MD5 (Message-Digest Algorithm [Riv92]) in entropy gathering generators.

Let us shortly discuss the usage of hash functions. In general, a hash function maps an arbitrary binary sequence to a (shorter) binary sequence of a fixed length. The fixed-length sequence is called the hash value or message digest. For *cryptographic hash functions* we assume that

- in practice, with limited computational resources, it is not possible to calculate the original message, given the resulting hash value (thus, hash functions are should be called *one-way functions*),
- by changing one bit of the input, about half of the output bits are changed (this property is called *avalanche effect*),
- every output bit is a function of all input bits, and
- It is practically impossible to find two input sequences that are mapped to the same output string (there is a low probability of *collisions*).

Under the condition that the generator uses a cryptographic hash function that fulfills the above criteria to mix the input of noise sources into the inner pool, the following assumptions are justified.

First let us assume that the inner state has n bits of entropy with respect to an adversary, which means on average the adversary would have to ask about n questions to guess the content of the state. Even if the adversary completely knows the content of the next input that gets mixed into the state, the state will still contains n bits of entropy after the mixing has been done. Likewise, if the adversary knows the current state of the generator, but the next input contains n bit of entropy, then after the mixing procedure the entropy of the state is again n bits. Consequently, the entropy of the unknown data was conserved. (see also Section 3.2.4 for this topic)

If the hash function is employed to produce the random numbers from the inner state, then it is practically impossible, to recover the inner state from the observed random numbers.

We will encounter several examples for the use of such cryptographic primitives like cryptographic hash functions in the subsequent generators.

In addition to the choice of adequate noise sources and reliable entropy estimators, speed is the main problem of entropy gathering generators. Since most of the time the noise sources produce entropy only at a low rate, these generators are considerably slower than generators of the first or the third category.

7.1.3 Hybrid Generators

The last kind of RNGs is a hybrid of the first two categories. Not only does it collect entropy, but it also uses some additional methods to enhance the amount of generated random numbers. Two examples are the Yarrow generator and the HAVEGE generator in Chapters 10 and 13, respectively.

Yarrow continuously gathers entropy from external sources. If the user demands more random numbers than the estimated amount of collected entropy, an additional PRNG is used to stretch the gathered information.

HAVEGE follows another strategy. It does not only collect input from a noise source but amplifies the collected entropy by directly influencing the source. Two random walks through a table filled with random numbers are employed to enhance the "chaotic" behavior of the processor.

7.2 General Problems and Properties

Apart from statistical properties, the two main requirements for cryptographic RNGs are *speed* and *robustness against attacks*. Speed may not play such an important role if random numbers are used to generate a single cryptographic key, but for all applications that require a large amount of random numbers like stream ciphers, simulations, or masking of protocols, speed is a basic prerequisite. Robustness against attacks clearly plays an important role for cryptographic RNGs. Several possible attacks are explained in Chapter 8. The RNG of the SSL (Secure Socket Layer) encryption in the Netscape browser Version 1.1 is a famous example of a generator which did not satisfy the criterion of robustness. The main reason for the weakness of this generator was the inappropriate choice of random sources, like the process ID and the current time. A 128-bit session key produced by this generator only contained at most 47 bits of entropy and, thus, could be broken within minutes [GW96].

Especially with PRNGs, robustness and speed are often contradictory properties. Fast and simple generators like linear feedback shift registers (LFSR) are most of the time not sufficiently secure for cryptographic applications. In the case of an LFSR, an adversary is able to determine the seed of a generator of length n by observing only $2n$ consecutive output bits, due to the low linear complexity of the output stream [Sch93, p. 353]. In return, PRNGs that are more robust against attacks, like the Blum-Blum-Shub generator,

described in Chapter 11, are usually much slower. However, there exist notable exceptions like AES (described in Chapter 12), which combines speed with high robustness against attacks.

The basic problem of random number generators is where to find randomness in such a deterministic device as a computer. Every RNG, even one of the PRNG category, needs at least once a source of randomness to produce random numbers. Often mentioned sources of randomness are user input, like mouse movements or time between keystrokes, audio devices, network statistics, or current system time [Ell95, p. 6 ff.].

System time is probably the worst choice of the examples mentioned above. If the execution time of the random number generator is approximately known, the time will only contain a few bits of entropy. This was the main problem of the Netscape browser v1.1 [GW96] and the Kerberos V4 [DLS97] implementations. Nevertheless, the time difference between two high resolution clocks, like a hardware clock and a software clock, may be used to gather a reasonable amount of uncertainty.

However, other sources also have their drawbacks. Network activities always contain the risk of observation or are nonexistent if the computer is not part of a network. User input from a mouse or a keyboard is not available for all computers and remote user input may again be observable. The randomness of an audio device gets highly reduced if the device gets fed with a high-amplitude periodic sound.

In addition, the operating system in use may influence the uncertainty of the sources as well. For example, some OS offer a "snap to" functionality of the mouse, which snaps the mouse pointer to the center of an icon, as soon as the pointer gets near to it. The entropy of the mouse device thus becomes considerably reduced by such a functionality. Generally, the treatment of interrupts by an OS highly influences the amount of entropy that can be gathered from external events. Consequently, the quality of random sources also depends on the OS used.

Some RNG developers may be tempted to pass the choice of applied random sources to the user. This allows to optimally adjust the generator to the available system. But most of the time such intentions will result in worse random numbers, since the average user has even less knowledge about appropriate entropy sources or just does not want to think about such problems. A similar situation occurred in connection with OpenSSL. In Version 0.9.5 of OpenSSL the program checked for the first time if any entropy was added to the PRNG used. This step produced lots of error messages, since most of the users never applied any random source. In many mailing lists a popular recommendation for this problem was to feed the random number generator with a fixed string like the sentence [Hes00]

```
"string to make the random number generator think it has entropy".
```

There exist no general studies concerning the amount of entropy the sources, mentioned above, are able to provide. However, such studies would be difficult, since the amount highly depends on the specific configuration of the system used. Generally, practitioners

suggest to apply several separated sources and to mix them by means of an appropriate function (e.g., a hash function) [Ell95, p. 11]. Thus, defects of single sources should be compensated. But if heavy defects occur in all sources used, the quality of the generated random numbers will still suffer from this fact. For this reason, it would be advisable to continuously check by means of statistical tests that no total breakdown of the entropy sources happens.

In general, the quality of produced random numbers can only be judged by statistical tests. Consequently, for any statement about the quality of a RNG, it is important to know on which tests this statement is based.

7.3 Description scheme

We will now introduce a scheme for describing RNGs, which is derived from L'Ecuyer's [L'E94] definition for pseudorandom number generators. Whereas L'Ecuyer's definition is limited to the first kind of generators, our enhanced version allows to cover all three categories by adding the notion of an input-dependent transition function.

L'Ecuyer defines a random number generator by a tuple $(\mathcal{S}, T, \mathcal{O}, g, s_0)$, where $\mathcal{S}$ represents the finite state space, $\mathcal{O}$ the finite output space, $T : \mathcal{S} \to \mathcal{S}$ the transition function, $g : \mathcal{S} \to \mathcal{O}$ the output function and $s_0 \in \mathcal{S}$ the seed of the generator. The generator starts in the initial state s_0. In each iteration, the state $s_n \in \mathcal{S}$ is changed by the transition function T to the next state s_{n+1},

$$s_{n+1} = T(s_n).$$

The output $o_n \in \mathcal{O}$ is produced by means of the output function g,

$$o_n = g(s_n).$$

The output sequence $(o_n)_{n\geq 1}$ is periodic, since only a finite number of different states are available and the next state depends uniquely on the current state. The size $|\mathcal{S}|$ of the state space is an upper bound to the period length of the output.

L'Ecuyer's scheme is limited to generators which, after seeding, never process any input. Thus, their transition function T is constant. We extend his definition to a more complex transition function $T : \mathcal{I} \times \mathcal{S} \to \mathcal{S}$ that calculates the next state depending on the current state s_n as well as on an external input $i_n \in \mathcal{I}$, such that $s_{n+1} = T(i_n, s_n)$. This extension allows us to describe a more general family of generators.

Definition 7.1 (Random Number Generator (RNG))
A RNG is a tuple $G = (\mathcal{S}, T, \mathcal{O}, \mathcal{I}, g, s_0)$, *where*

- $\mathcal{S}$ *is the finite state space,*
- $\mathcal{O}$ *is the finite output space,*

- $\mathcal{I}$ *is the input space,*
- $s_0 \in \mathcal{S}$ *is the seed,*
- $g : \mathcal{S} \to \mathcal{O}$ *is the output function and*
- $T : \mathcal{I} \times \mathcal{S} \to \mathcal{S}$ *is the transition function.*

The time steps $n \in \mathbb{N}$ are chosen such that n is increased every time new output is generated. Thus, for any $n \in \mathbb{N}$, the output is computed by

$$o_n = g(s_n),$$

and the new state is generated by

$$s_{n+1} = T(i_n, s_n),$$

depending on the current state s_n and the input $i_n \in \mathcal{I}$. Most of the time, generators are able to process arbitrarily many input strings between the generation of two consecutive outputs. All input strings that are processed between the generation of o_n and o_{n+1} are combined together to the input i_n. In such a case the input space may, at least theoretically, be assumed to be infinite, since i_n may be derived from arbitrarily many input strings. The

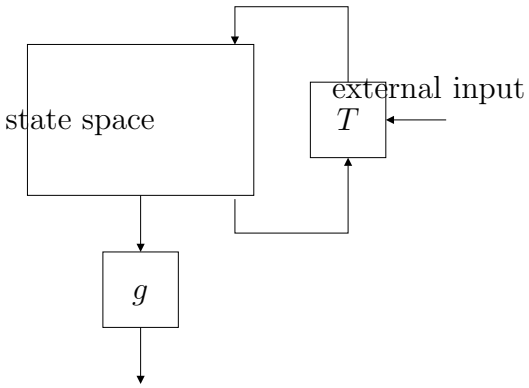

Figure 7.3: The general structure of a RNG

layout of this structure can be found in Figure 7.3. Our definition allows us to describe all three different categories of random number generators. We distinguish two special cases.

7.3.1 Case 1: Constant transition function

The first category is represented by the special case of a constant transition function. It reflects L'Ecuyer's original definition and, thus, describes our PRNG category (Section 7.1.1). In this case, T does not depend on any input and the class of functions $\{T(i, \cdot) : i \in \mathcal{I}\}$ reduces to the function $T(\cdot)$. The input space is empty ($\mathcal{I} = \emptyset$).

Remark 7.2

1. *Since $\mathcal{S}$ is finite, the output of the generator will be periodic with a period less than or equal to the number $|\mathcal{S}|$. Well-designed RNGs of this kind have a period length that is almost as large as the state space itself.*

2. *A sequence generated from this kind of generator is completely reproducible once the seed s_0 is known. Thus, only the short seed has to be transmitted or stored to obtain the original sequence, which is useful in the case of stream ciphers.*

3. *Concerning security, the disadvantage of this kind of generators is that if the state is compromised once, all future outputs are predictable.*

7.3.2 Case 2: Input-dependent transition function.

This case covers the second and the third category of random number generators (Sections 7.1.2 and 7.1.3). The transition function additionally depends on external input $i_n \in \mathcal{I}$. On the assumption of random input, the output of this kind of generators generally loses its periodicity.

Remark 7.3

1. *The seed is less important and most of these generators recover from a compromised state.*

2. *If we need the property that the output is reproducible from a short sequence, this family of generators is not appropriate.*

3. *Theoretical analysis of these generators like a priori analysis of the period length of output streams or correlation analysis by number-theoretical figures of merit like discrepancy (see [Nie92]) or the spectral test (see [Hel98] and [L'E04]) is impossible*

7.3.3 Cryptographic Strength

The *strength* of a generator is an additional figure of merit, which we will use to compare different RNGs. It represents the number of bits of the inner state that must be guessed by an adversary to predict the next output.

In the next chapter we discuss more closely possible attacks on cryptographic random number generators. The remaining chapters are then used to introduce five examples of RNGs. In addition to the description of each generator we will consider security aspects as well as their practical usability.

Chapter 8

Attacks

The main difference between random number generators for stochastic simulations and cryptographic applications is that in cryptographic systems the RNGs additionally have to provide robustness against attacks. [KSWH98, p. 3] defines an attack as "... any method of distinguishing between PRNG outputs and random output". For us an attacks means the attempt

- to learn about the unknown output,
- to gain information about the inner state (and, thus, about future output), or
- to manipulate the output of the generator.

We split the possible attacks into three different classes, *cryptanalytic attacks*, *input based attacks*, and *state compromise extension attacks*. The first type tries to gain information about the inner state or future output of the generator by observing parts of the current output. Input based attacks achieve their goal by observing or manipulating the input of the RNG. The aim of the last kind of attack is to extend the knowledge of the current generator state to the future or the past.

In this chapter we introduce several different attacks. We will show how they work and which effects they have on real RNGs. The classification and some examples are taken from [KSWH98]. Additional examples can be found in [DLS97] and [Gut98]. This chapter should not be seen as a complete reference of all possible attacks but as an introduction into this topic. It should encourage the reader to examine a generator more closely before using it. Furthermore, it should enhance the understanding of the security aspects in the subsequent discussion of RNGs.

8.1 Direct Cryptanalytic Attacks

During a cryptanalytic attack the adversary observes the output and tries to gain any information about the inner state or future output of the generator. Many RNGs use

cryptographic primitives like hash functions (e.g. SHA-1 or MD5) or block ciphers (DES, Triple-DES, AES) to prevent this kind of attacks. The underlying assumption is that the cryptographic security of the primitives transfers to the generators which employ them. Generally, the confidence into the security of this primitives is based only partially on mathematical analysis but mainly on empirical results and statistical tests. Since most of the applications that apply cryptographic RNGs rely on those primitives, we may have confidence in their security as well.

Nevertheless, it is not advisable to blindly trust generators that are built on cryptographic primitives as we will see by the example of the Kerberos 4 session key generator [DLS97]. The specific method of employing the primitives has a main impact on the security of the generator as well. The Kerberos 4 generator produces a 56-bit key for a DES block cipher by two successive calls of the UNIX `random` function which uses only a 32 bit key. The `random` function is seeded every time a key is requested. Consequently, the strength of the encryption and, thus, the resistance against cryptanalytic attacks is reduced from 56 to 32 bits. It still takes about 6 hours on a DEC Alpha [DLS97, p. 5] to gain the proper key of a plaintext-ciphertext pair by brute force, but we see that the 56 bit strength of the encryption is only an illusion. It is the weakest link in the chain that counts.

By the example of the RSAREF 2.0 generator [DLS97, p. 11 ff.], we shall study two additional cryptanalytic attacks which are especially dangerous for counter-based RNGs. The inner state of the RSAREF 2.0 generator consists of a 128-bit counter C_i, $i \geq 1$. Every time external input X_i occurs, it is used to additionally change the content of the counter by means of a MD5 hash function

$$C_{i+1} \equiv C_i + MD5(X_i) \pmod{2^{128}}.$$

The output of the generator is produced by

$$\begin{aligned} output[i] &\equiv MD5(C_i) \pmod{2^{128}} \text{ and} \\ C_{i+1} &\equiv C_i + 1 \pmod{2^{128}}. \end{aligned}$$

We are now going to explain the concept of a *partial precomputation attack* and a *timing attack* by the example of this generator.

8.1.1 Partial Precomputation Attack

A partial precomputation attack may be launched on any generator which uses a counter. Suppose no input is processed and an adversary is able to observe t successive outputs. In a next step, he or she has to compute the output (in our case, the hash-function) of every t'th value of the counter and to store it in a list. One of the t observed values must be included in the list. Having found the entry in the list, the inner state of the generator is revealed. All further outputs are known as long as no new input is processed. For a

128-bit counter this attack is not very practical. Even if the adversary is able to observe 2^{32} consecutive outputs, 2^{96} possible precomputations have to be done and stored. There are many attacks which appear not to be useful in practice, but they give an idea of possible flaws, especially when some methods can be combined.

8.1.2 Timing Attack

This attack uses the fact that the incrementation of a counter requires different amounts of time, depending on how many byte additions have to be made. If an adversary has the possibility to measure the time needed for incrementing the counter, he or she may draw conclusions on the number of zeros in the current state of the counter. It may be possible to guess the point of time, when all the low order bytes of the counter are zero, since then, the previous increment has needed particularly many byte additions. Such a situation is considered as a "weak state" in a timing attack. The information we have gained due to this attack can be combined with a precomputation attack. This means the adversary knows when it is more profitable to compare the output to the precomputation list.

8.2 Input Based Attacks

In an input based attack the adversary is able to observe or to manipulate the input of the generator. The goal of this action is to reduce the number of possible outputs, so that it becomes easier to guess them, or even to force the generator to produce designated output. There are three different kinds of input based attacks, *chosen input attacks*, *replayed input attacks*, and *known input attacks*. We are going to present the different attacks by the means of several examples.

8.2.1 Chosen Input Attacks

Here, the adversary has the power to directly manipulate the input of the generator. Such a situation appears relatively seldom, but allows very effective attacks.

At first, we shall study this attack by the example of the DSA generator [KSWH98, p. 8 ff.]. The generator is based on the SHA hash function and was designed to produce DES keys. All additions are done modulo 2^N, where $160 \leq N \leq 512$. In our example we use $N = 160$, because this value represents the weakest version of the generator. The generator contains an inner state X_i, $i \geq 1$. New input W_i is processed every time an output is generated. If no input is available, W_i is set to zero. The output is produced by

$$\begin{aligned} output[i] &\equiv SHA(W_i + X_i \pmod{2^{160}}) \text{ and} \\ X_{i+1} &\equiv X_i + output[i] + 1 \pmod{2^{160}}. \end{aligned}$$

An efficient attack would be to set the input of the generator to

$$W_i \equiv W_{i-1} - output[i-1] - 1 \pmod{2^{160}}.$$

This new input forces the generator to cycle immediately, but does not give any information about the actual value of the output.

Another chosen input attack is shown in [KSWH98], this time on the RSAFEF 2.0 generator (see Section 8.1). At first, the adversary has to find an input $input_n$, such that $MD5(input_n)$ has all ones in the n low order bytes. If the generator is fed with this input every time an output is generated, then the RNG is forced to cycle after 2^{128-8n} outputs, since the low order n bytes never change. It is quite expensive to find a suitable $input_n$ for greater values of n. We may use the birthday paradox [MvOV01, p. 53 and p. 369 f.] to reduce the precomputation effort. This paradox states that if we choose elements out of a set of size 2^N according to a uniform distribution, we may expect to have chosen two identical elements after $2^{\frac{N}{2}}$ attempts. We use this effect to find two values x_1 and x_2 such that $MD5(x_1) + MD5(x_2)$ modulo 2^{128} has all ones in the n low order bytes. The generator is fed with the series $x_1x_2x_1x_2\ldots$, which has the same effect as feeding the generator with $input_n$. However, due to the birthday paradox it is much easier to find suitable x_1 and x_2. With $n = 16$ we only need approximately $2^{64} = 2^{\frac{16 \cdot 8}{2}}$ searches, but force the generator to cycle immediately.

There also exists another chosen input attack on the RSAREF 2.0 generator. In this case, we are not forcing the generator to cycle but to repeat a specific previous output. If no input was processed for j consecutive outputs, then the generator can be brought into the previous state C_{i-j} by feeding an input X_i such that $MD5(X_i) = -j \pmod{2^{128}}$.

Replayed Input Attack

An attack which is similar, but not as effective as the chosen input attack, is the *replayed input attack*. In this case the adversary may replay existing input, but is not able to manipulate it arbitrarily.

8.2.2 Known Input Attack

During this kind of attack, the adversary is able to observe part of the input but he or she may not manipulate it. The knowledge of the input may be used to limit the number of possible output values and, thus, to reduce the strength of the generator, or to support other attacks, like for example brute force. Known input attacks can occur if the entropy estimation of the input is incorrect, or if observable input devices are applied. The first case means that the collected entropy contains less entropy regarding an adversary than the user may assume. The case of observable input can occur if remote user input, which was sent over a network, is employed as an input source. Generally, any information that is transferred over a network is a dangerous source of randomness.

One example of a known input attack on the Kerberos session key generator can be found in [DLS97]. As we have already shown in Section 8.1, the strength of the generator is at most 32 bits. The `random` function is seeded with a 32 bit key every time a DES key

is requested. The seeding is done by a XOR combination of

- the time of day seconds since UTC 0:00 Jan. 1, 1970,
- the process ID of the Kerberos server process,
- the cumulative count of session keys generated,
- the fractional part of time of day seconds since UTC 0:00 Jan. 1, 1970 in microseconds, and
- the hostid of the machine on which the Kerberos server is running [DLS97, p. 4].

Almost all entropy of the key is contained in the least significant 12 bits of the seed. The remaining first 20 bits stay constant for a period of about 12 days (2^{20} seconds). If the seed is determined once by a 32 bit brute force attack, we can use this information to reduce the strength of the generator down to 12 bits. 12 bits can be cracked easily by brute force. The 12 day period of the validity of the first 20 bits makes the session keys also vulnerable for a precomputation attack. The adversary has to calculate the cipher text for a given plaintext and for all 2^{12} possible keys if the first 20 bits are known. Subsequently, all the cipher text and key pairs have to be stored in a sorted table. To find the right key, only the matching entry in the table must be found. With precomputation, the session key can be found within a few hundreds of milliseconds, without it takes a few seconds. Thus, we see, that the knowledge of part of the input helps to gain the DES-key much easier.

8.3 State Compromise Extension Attacks

In this case we assume that at a given moment the adversary already knows part of the inner state of the generator. The attack tries to extend this knowledge to further points in time and to previous or future output, respectively. Such a situation may occur if the generating process was forked or if the RNG was started in an insecure state. The second case happens when the generator uses a fixed initial value (e.g. all zeros) and completely trusts in the processing of input, or if the generator was seeded from a file which was accessible to the adversary. During the generation process, the inner state may be revealed if it gets swapped onto an insecure part of the hard disk. A swapping happens for example if the memory in the processor becomes too small and old data has to be removed to make space for new data. [Gut98] suggests a few guidelines to protect the inner state of the generator. One of them would be to choose the strictest access possible for any file that is applied during the generation of random numbers. Generally, all generators of the PRNG category are jeopardized by state compromise extension attacks, since they never process any input. However, also other generators are vulnerable to these attacks as we will see by the example of the ANSI X9.1 random number generator [KSWH98, p. 5 ff.].

This RNG employs a secret and fixed Triple-DES key $\mathcal{K}$. The output is generated with the help of the current time, the internal state X_i, $i \geq 1$, and the DES encryption function $E_{\mathcal{K}}$ by

$$\begin{aligned} T_i &= E_{\mathcal{K}}(currenttimestamp), \\ output[i] &= E_{\mathcal{K}}(T_i \oplus X_i), \text{ and} \\ X_{i+1} &= E_{\mathcal{K}}(T_i \oplus output[i]). \end{aligned}$$

The operation $\oplus$ represents the bit-wise XOR operator. In all of the following attacks we assume that the adversary was able to learn the secret key $\mathcal{K}$.

8.3.1 Permanent Compromise Attack

This attacks means that a generator never fully recovers from a compromised state. The adversary is able to determine future and even previous output values.

Let us assume we could expose the key $\mathcal{K}$ of the ANSI X9.17 generator. Since the key is never changed by any new input, we process this information for the future at least until the whole generator including the key is reseeded. Some times later we observe two successive outputs ($output[i]$, $output[i+1]$). Assuming the current timestamp contains only 10 unknown bits, which is a realistic value, there are 2^{10} guesses for each T_i and T_{i+1}. X_{i+1} can be calculated by two different methods

$$\begin{aligned} X_{i+1} &= D_{\mathcal{K}}(T_{i+1} \oplus output[i+1]) \text{ and} \\ X_{i+1} &= E_{\mathcal{K}}(T_i \oplus output[i]), \end{aligned}$$

where $D_{\mathcal{K}}$ stands for the DES decryption. For each guess of T_i, X_{i+1} is calculated and stored in a sorted table. Subsequently, the calculation is done for each T_{i+1}. The correct value of X_{i+1} appears as a result of both computations. Thus, we only need about 2^{11} calculations to reveal the current state X_{i+1} of the generator (2^{10} calculations for determining X_{i+1} from T_i and T_{i+1}, respectively).

8.3.2 Backtracking Attack

A backtracking attack uses the compromised state to gain information about previous outputs. With the ANSI X9.17 RNG it is as easy to learn about future outputs as previous once if we use the same method as in the permanent compromised attack.

8.3.3 Integrative Guessing Attack

In this attack the adversary knows the current state S of the generator at time t and observes subsequent output. In contrast to a permanent compromise attack it is sufficient to know only a function of the output, not the output itself. Such a function could be an

encryption with the help of a generated key. An iterative guessing attack uses guessable but unknown input to determine the state S at time $t + \epsilon$.

In the case of the ANSI 9.17 generator it is quite easy to apply this attack for $\epsilon = 1$. Let us suppose, that we know the current state of the generator at time i including $\mathcal{K}$, X_i and $output[i]$ and that we see a function of $output[i + 1]$. We use the previous assumption that the time contains only 10 bits of entropy. Then, we need to test at most 2^{10} possible input values and compare the results with the function of the $output[i + 1]$ to predict the inner state X_{i+1}.

8.3.4 Meet-In-The-Middle Attack

The Meet-In-The-Middle attack combines the methods of the iterative guessing attack and the backtracking attack. The knowledge of the state S at time t and $t + 2\epsilon$ is used to determine the state at time $t + \epsilon$.

Let us assume that the ANSI X9.17 generator produces a sequence of eight consecutive cipher keys for encrypting a plaintext. The output of the generator is unknown but we were able to learn the states X_i and X_{i+1} and the encrypted ciphertext, which used the key produced at time $t + 4$. We assume that each timestamp contains 10 bit of entropy. The Meet-In-The-Middle attack allows us to find the key with much less costs than 2^{80} attempts. In the same way as was described for the permanent compromise attacks, we calculate X_{i+4} from the front and from the back by guessing $T_{i+1,i+2,i+2,i+4}$ and $T_{i+5,i+6,i+7,i+8}$. About 2^{41} calculations must be done. The values of the computations from both sides are stored in two lists and are compared to each other. 2^{16} matches will be found. A final 2^{16} key search reveals the proper key to the observed ciphertext.

Time Entropy Issues

The entropy of a time stamp must be estimated very carefully. In particular, when the generator is used to generate successive outputs in a very short period of time. If two keys are generated by consecutive calls of a RNG, the corresponding timestamps will differ only by a few bits. This property allows different kind of attacks but it makes especially Meet-In-The-Middle attack even more effective.

Some of these attacks may be easily prevented by simple countermeasures. State compromise attacks can be avoided by frequently changing the complete state of the generator. The power of input based attacks can be reduced by using different input sources and by combining all the collected data with the help of a cryptographic hash function.

A detailed description of attacks and possible counter measures can be found in [KSWH98].

Chapter 9

/dev/random

The device /dev/random is probably the most common random number generator within the Linux kernel. It was designed by Theodore T'so and is part of the Linux kernel since Linux 1.3.30 (1995). In our discussion we refer to Version 1.89 [Ts'99] of the generator.

/dev/random gathers entropy from external events like user input or unpredictable interrupts. The generator estimates the collected entropy and produces at most that many random bytes.

A variant of this generator is /dev/urandom. This device produces as many random bytes as are requested by the user without checking if enough entropy was gathered. Consequently, the output of /dev/urandom may contain only little entropy and would thus be vulnerable to attacks. This fact makes this variant inappropriate for cryptographic applications. Therefore, for security-sensitive tasks /dev/random is applied.

Compared with our three RNG categories, /dev/random belongs to the entropy gathering category, whereas /dev/urandom belongs to the class of hybrid generators.

9.1 General Structure

The main feature of the random device consists of two pools. The primary pool $\mathcal{P}_1$ is used to gather entropy from external events $\mathcal{E}$, whereas the secondary pool $\mathcal{P}_2$ is used to produce random bytes. On demand, bytes are shifted from $\mathcal{P}_1$ to $\mathcal{P}_2$.

There exist two functions that interfere with the pools, the mixing function m and the generation function *gen*. The mixing function merges input into the pool, whereas the generation function produces random numbers from the pool.

The gathering of entropy from external events happens in two steps. At first, the extracting function $e : \mathcal{E} \to \mathcal{D}$ converts the events from $\mathcal{E}$ into data from the set $\mathcal{D}$, which satisfy a given format. Those data bytes are then merged into $\mathcal{P}_1$ by means of the mixing function m.

To produce random bytes from $\mathcal{P}_2$ we apply the generation function *gen*. If $\mathcal{P}_2$ contains enough entropy, the bytes are generated exclusively from $\mathcal{P}_2$. Otherwise, bytes are shifted from $\mathcal{P}_1$ to $\mathcal{P}_2$ to achieve a maximal entropy in $\mathcal{P}_2$, prior to the production of the random

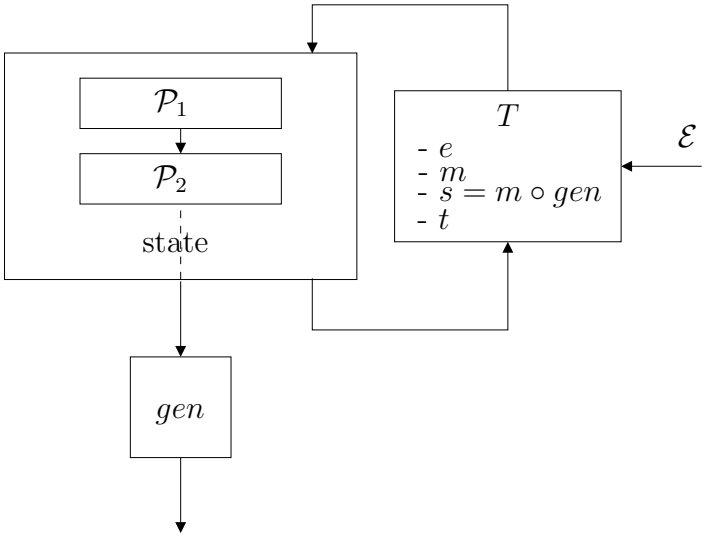

Figure 9.1: General structure of /dev/random

bytes. Thus, /dev/random checks only the amount of entropy in P_1 and not in P_2 to guarantee that the generator contains enough entropy for a secure production of random bytes.

The shifting of the bytes from $\mathcal{P}_1$ to $\mathcal{P}_2$ uses the function $s = gen \circ m$. For this purpose the random bytes are, at first, produced from $\mathcal{P}_1$ by means of the generation function gen. Subsequently, those bytes are merged into $\mathcal{P}_2$ using the mixing function m.

The transition function T is responsible for processing new input, for shifting information from the primary to the secondary pool and for mixing the secondary pool after random numbers have been produced. The last feature is done by the function $t : \mathcal{P}_2 \to \mathcal{P}_2$.

Thus, T consists of

- $e : \mathcal{E} \to \mathcal{D}$,
- $m : \mathcal{D} \to \mathcal{P}_1$,
- $s : \mathcal{P}_1 \to \mathcal{P}_2$,
- $t : \mathcal{P}_2 \to \mathcal{P}_2$.

To estimate the amount of entropy contained in $\mathcal{P}_1$ and $\mathcal{P}_2$, an entropy counter is associated with each pool. If input is mixed into one pool, its counter is increased by the estimated entropy of the input. Conversely, if random numbers are generated from one pool, its counter is decreased by the number of generated random bytes. The estimation of the input will be given in Section 9.3.1.

Finally, one should not picture a pool as a box, which contains variable amounts of bytes where each byte has an entropy of 1.0. Actually a pool is an array of fixed length

that is always completely filled with bytes. Only the entropy count provides information about the entropy contained in the pool, where the entropy is estimated over the whole content of the array. Every time input is merged into the pool the content of the array is changed and every time random bytes are generated from the pool, the whole content is used as input for the generation function.

9.2 State Space

The primary pool $\mathcal{P}_1$ is responsible for gathering entropy, the secondary pool $\mathcal{P}_2$ is employed to generate the random bytes. Generally, the size of the buffer can be set to any arbitrary multiple of 64 byte, but we will use the default sizes of 512 byte for $\mathcal{P}_1$ and 128 byte for $\mathcal{P}_2$.

The generator produces the random numbers exclusively from $\mathcal{P}_2$ as long as no information is shifted from $\mathcal{P}_1$. Information is shifted if more random numbers are requested than entropy is contained in $\mathcal{P}_2$ and every time after 128 random bytes have been produced. The second case is called catastrophic reseeding and renews the whole content of $\mathcal{P}_2$, independently from its current entropy count. Thus, in the worst case, 128 bytes of random numbers are generated from the 128 bytes of $\mathcal{P}_2$ without any new input from $\mathcal{P}_1$. If the entropy estimation of the input, and consequently the entropy count of the two pools is correct, then we obtain a strength of 128 bytes for /dev/random.

At the very beginning the pools are seeded with all zeros. As soon as input arrives from external events, the pools change into a more and more unpredictable state. The start-up of a computer does not contain many unpredictable events. Most of the time these events will appear in almost the same order. Hence, the designer of /dev/random recommends to store the state of the two pools during shut-down and restore it during start-up. However, one has to ensure that the stored data are really refreshed at every shut-down and that they are not observable from any other person.

9.3 Transition Function

The transition function is responsible for processing the input, shifting the bits from the primary pool to the secondary pool and mixing the secondary pool after an output has been produced.

9.3.1 Processing the Input

/dev/random gathers entropy from unpredictable external events. At first, the function e extracts the time and specific additional data from the event $\mathcal{E}$. Both pieces of information are combined in the extracted data $\mathcal{D}$. Subsequently, the entropy gained of the new event is estimated, $\mathcal{D}$ is fed into one pool and the corresponding entropy counter is increased

by the estimated amount. Normally the input is mixed into the primary pool, but if the entropy count of the primary pool has reached the pool size, then the input is also mixed directly into the secondary pool.

In the design of the input processing functions the main focus was set to efficiency, since the input routine is carried out during the processing of interrupts.

Entropy Estimation

The entropy of an input is estimated by the following equation

$$\begin{aligned} \Delta_n^1 &= time_n - time_{n-1}, \\ \Delta_n^2 &= \Delta_n^1 - \Delta_{n-1}^1, \\ \Delta_n^3 &= \Delta_n^2 - \Delta_{n-1}^2, \\ \Delta_n &= \min(|\Delta_n^1|, |\Delta_n^2|, |\Delta_n^3|), \\ entropy_n &= \log_2 \left(\left\lfloor \frac{\Delta_n}{2} \right\rfloor \pmod{2^{12}} \right). \end{aligned}$$

The variable $time_n$ represents the timestamp of the external event from a specific source. Each source has its own sequence $\{time_n\}_{n \geq 0}$. The reduction modulo 2^{12} limits the entropy to at most 12 bits.

This approach of entropy estimation acts on the assumption that the uncertainty of a new input-time is reflected in the differences to the previous times. We have found no theoretical proof that this algorithm gives a consistent estimator of the entropy of the source. Such a proof would be of high interest, since entropy estimation plays an important role for the security of /dev/random.

Finally, the entropy count of the pool, into which the new data was mixed, is increased by the estimated entropy.

Mixing Function

The mixing function is realized by a simple hash function. It processes arbitrary large input and inserts the results into the pool. The main purpose of the hash function is to avoid statistically significant coalitions. Thus, the entropy of the input is preserved. (see Section 3.2.4)

The hash function combines a cyclic redundancy check polynomial (CRC-32) [Wil93] and a twisted generalized feedback shift register (twisted GFSR) [L'E04]. It always processes one 32-bit word at once and puts the resulting 32-bit word at a specific position in the pool. This position is determined by a pointer and rotates to the left after each processed word.

The mixing function of /dev/random uses a CRC-32 polynomial. Such polynomials are usually employed to generate the checksum of a plaintext. For that purpose, the binary representation of the plaintext is interpreted as a polynomial $p(x)$ with coefficients 0 or 1

(e.g., 11001001 is seen as $1 + x + x^4 + x^7$). The CRC-32 checksum is the product of x^{32} and $p(x)$, reduced modulo the CRC-32 polynomial $c(x)$. Most CRC-32 implementations process the input byte-wise. For this purpose they employ a table, which assigns each byte its corresponding CRC-32 value. These 32 bits are combined with the previous results to determine the final checksum. Our mixing function uses this table only for processing one byte, not for longer strings. The result of this operation is a 32-bit word.

The processing of a word in the mixing function happens in different steps. At first, the word is rotated. The amount of rotation changes after every word by a fixed quantity.

In a second step the word is combined with values from the pool, as it is usual for feedback shift registers. Which words are chosen from the pool, starting from the pointer, is determined by a so-called polynomial. The generator defines polynomials for different pool sizes, such that the words are chosen evenly from all over the pool. Theodore T'so states in the code that the content of the pool is "(essentially) the input modulo the generator polynomial" and that for "random primitive polynomials, this is a universal class of hash functions" [Ts'99, line 359ff]. He does not claim that the provided polynomials are primitive, but argues that their irreducible factors are large enough such that periodicity is not a problem. We give this information without examining it in more detail.

Finally, the resulting word is shifted by 3 bits to the right. The former three least significant bits $b_2b_1b_0$ are set at the beginning of a byte $b = b_2b_1b_000000$. Then, by means of the CRC-Table, the CRC-32 value of the byte b is calculated and combined by XOR with the shifted word.

The resulting word is now put into the pool at the position of the pointer.

9.3.2 Shifting the information

Prior to the generation of random numbers from the secondary pool $\mathcal{P}_2$, the number of requested bits is compared to the estimated entropy of the pool. If the entropy count of $\mathcal{P}_2$ is too low, then the generator shifts bytes from the primary pool $\mathcal{P}_1$ until the entropy count of $\mathcal{P}_2$ reaches its maximum, which is the size of the pool.

In addition, $\mathcal{P}_2$ is totally rebuilt after each generation of 128 random bytes. This process is called a *catastrophic reseeding*. An output counter keeps track of the number of generated output bytes. If the counter exceeds 128 bytes, it is reset to zero and a catastrophic reseeding is initiated. For this purpose, 340 bytes (the size of a temporary buffer) are transfered from $\mathcal{P}_1$ to the $\mathcal{P}_2$. After the catastrophic reseeding only the entropy count of $\mathcal{P}_1$ is decreased by the number of shifted bits, the count of $\mathcal{P}_2$ remains unchanged. Due to the catastrophic reseeding, at most 128 random bytes are generated from $\mathcal{P}_2$ without any new input from the $\mathcal{P}_1$ primary pool. Thus, even when an adversary is capable of inverting the output function and compromises the content of $\mathcal{P}_2$ after 128 observed output bytes, he or she should not be able to take any profit from this knowledge.

We will give a short example, to illustrate in which situations bytes are shifted from $\mathcal{P}_1$ to $\mathcal{P}_2$.

1. Present state of the entropy counter $ec_{\mathcal{P}_1}$ and $ec_{\mathcal{P}_2}$ and the output counter oc.

$$\begin{aligned} ec_{\mathcal{P}_1} &= 482, \\ ec_{\mathcal{P}_2} &= 40, \\ oc &= 117. \end{aligned}$$

2. Required output: 60 bytes.

   (a) Shift bytes from $\mathcal{P}_1$ to $\mathcal{P}_2$.

$$\begin{aligned} ec_{\mathcal{P}_1} &= 394, \\ ec_{\mathcal{P}_2} &= 128, \\ oc &= 117. \end{aligned}$$

   (b) Produce 60 bytes from $\mathcal{P}_2$.

$$\begin{aligned} ec_{\mathcal{P}_1} &= 394, \\ ec_{\mathcal{P}_2} &= 68, \\ oc &= 177. \end{aligned}$$

3. Required output: 45 bytes.

   (a) Catastrophic reseeding. Shift 340 bytes from $\mathcal{P}_1$ to $\mathcal{P}_2$.

$$\begin{aligned} ec_{\mathcal{P}_1} &= 54, \\ ec_{\mathcal{P}_2} &= 68, \\ oc &= 0. \end{aligned}$$

   (b) Produce 45 bytes from $\mathcal{P}_2$.

$$\begin{aligned} ec_{\mathcal{P}_1} &= 54, \\ ec_{\mathcal{P}_2} &= 23, \\ oc &= 45. \end{aligned}$$

The counter of $\mathcal{P}_1$ may change during the calculation if any input is processed in the mean time. If $ec_{\mathcal{P}_1}$ is less than 340 prior to a catastrophic reseeding, then still 340 bytes are shifted and $ec_{\mathcal{P}_1}$ is set to 0.

For shifting the bytes, /dev/random first uses the generation function *gen* to produce random numbers from $\mathcal{P}_1$. Then it merges those number into $\mathcal{P}_2$ by means of the mixing function m. In the end, the entropy counters of both pools have to be refreshed. The counter $ec_{\mathcal{P}_1}$ is decreased by the amount of shifted data and $ec_{\mathcal{P}_2}$ is increased by the same value.

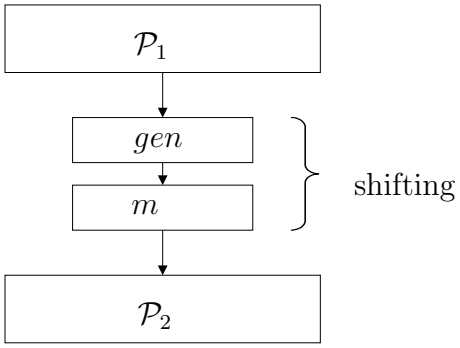

Figure 9.2: Shifting of bytes

9.3.3 Mixing the Secondary Pool

The generation function basically calculates the hash value of the secondary pool to produce the random numbers. In addition, it merges parts of the intermediate hash result back into the pool. This procedure is called $t : \mathcal{P}_2 \rightarrow \mathcal{P}_2$ and helps to prevent backtracking attacks, which means that an adversary should not be able to gain any information about previous outputs even if he or she knows the current state of the generator.

The intermediate hash result is merged into the pool by means of the mixing function, but without changing the entropy count. The secondary pool is therefore changed at each output, independent of any new external input.

9.3.4 Input Space

The input data is collected from the time between certain unpredictable events. The four main sources of entropy that have been implemented are:

- mouse-interrupt timing including the position of the mouse,
- keystroke timing including the ASCII-code of the pressed key,
- interrupt timing from unpredictable interrupts (no timer interrupts), and
- the finishing time of block requests.

Each new input data consists of the timestamp and an additional number which is determined by the specific source, like the code of the key pressed or the ID of the interrupt. Since arbitrarily many events may appear between two successive random number generations, the input space is infinite.

9.4 Output Function

The output function first checks if enough entropy is contained in the secondary pool $\mathcal{P}_2$ and then applies the generation function *gen* to produce the random numbers. The

essential part of the generation function consists of a cryptographic hash function like SHA-1 (Secure Hash Algorithm, [Nat02]) or MD5 (Message-Digest Algorithm, [Riv92]).

At a first state, the output function uses the current time as an additional input for the generator, but without increasing the entropy count of the primary pool $\mathcal{P}_1$. In the next step, the number of requested output bits is compared to the current value of the entropy count of $\mathcal{P}_2$. If the entropy count is too low, information is shifted from $\mathcal{P}_1$. In several iterations an output buffer is filled with the required random bytes. When the generation is finished the whole buffer is returned to the user at once.

Each iteration calculates a 160 bit hash value. For this purpose the hash function is reset to its initial values at the beginning of each iteration. Then the whole secondary buffer is fed into the hash function, always 64 bytes at once. After every 64 bytes a part of the current hash value is put back into the secondary pool using the mixing function (see Section 9.3.3). In the end the 160 bit hash value is cut into two halves and the two halves are combined again by a bitwise XOR. This folding is done to mask the hash value. Thus, even an adversary who is able to revert the hash function, cannot reconstruct the secondary pool. The resulting 80 bits are used to fill the output buffer.

In the end of each iteration the current time is again employed as an input for the generator.

9.5 Security

/dev/random applies several methods to frustrate attacks on the generator.

Two separated pools are used, one for processing the input and one for generating the output. This partition prevents iterative guessing attacks, because the input does not directly influence the output.

Furthermore, /dev/random is part of the kernel. Thus, chosen input attacks are hardly possible, since the generator directly uses the data from system events.

The generator employs a cryptographic hash function for producing the output. Therefore, by the state-of-the-art, the output function is not vulnerable against direct cryptographic attacks. In addition, the folding of the output masks the hash value. Thus, even if an adversary would be able to invert the hash function, he or she could not determine the state of the secondary pool from the generated random numbers. The folding has no big effect on the security of the algorithm, since the hash function can be assumed to be cryptographic secure, but it also does not weaken it.

Every time we generate an output, a part of the result is mixed back into pool. This helps to prevent backtracking attacks. If an assailant is able to compromise the current state of the generator, he or she cannot reconstruct previous output.

/dev/random extracts entropy from external events and uses it to produce random numbers. If we assume that during the processing of the input no entropy is lost and that the estimation of the entropy was correct, then the output of the generator reaches a per-bit

entropy of 1.0. Normally not much entropy is lost during the processing of the input, but the estimation of the entropy is the weakness of the generator. The security of /dev/random highly depends on the quality of the entropy estimation. The first problem is the estimation function itself. There exists no theoretical proof which guarantees the consistency of the estimator. A constant overestimation of the entropy would have fatal consequences on the security of the output. Another problem may result from inappropriate random sources. If, for example, the user input is carried out over a network, then the corresponding events may be observable. Consequently, the input loses its entropy with respect to an adversary. However, the entropy estimation function will produce an estimation of the entropy independently of the fact that the input might be known.

Altogether, if the input sources are unpredictable, not observable and not manipulable, then /dev/random is a trustable source of randomness. Since /dev/urandom never checks the quality of its generated random numbers, it should not be applied for security-sensitive applications.

9.6 Empirical Results

Statistical Tests

Graffam used Version 1.04 of /dev/random to produce an 11Mbyte file of random numbers on a heavily used Intel P5 based machine. This output passed all tests in the DIEHARD battery of Marsaglia [Mar95]. The results of the test can be found in [Gra99]. However, the empirical findings only show that the output of the hash function and thus the output of the generator passes all the tests. This property is required for all cryptographic hash functions and certainly applies to SHA-1 and MD-5. For judging the quality of the output it would be more meaningful to directly analyze the entropy of the input sources.

Throughput

The throughput of /dev/random depends on the frequency of the input. It may differ between 1500 bytes/s on a loaded machine [Gra99] and a few bytes per seconds if the machine is inactive [SS02].

9.7 Portability

The generators /dev/random and /dev/urandom are part of the Linux kernel. Linux-like environments for Windows such as cygwin (`http://www.cygwin.com`) make these generators also available for Windows operation systems.

9.8 Conclusion

The devices /dev/random and /dev/urandom are part of the Linux kernel and are therefore available to all Linux users. If /dev/random runs on a loaded machine with many unpredictable events, it produces good random numbers. But if only a few unpredictable events are available, then the generator gets slow and the output becomes more vulnerable to attacks. The main problem of this generator is speed. As most of the entropy-gathering generators, it is much slower than PRNGs or hybrid generators. /dev/random is not suitable if a large amount of random numbers is required, but it may be used to generate short random sequences, like random keys for encryption algorithms. /dev/urandom can never be recommended, neither for cryptographic applications nor for large-scale simulations.

Factsheet for /dev/random	
Cryptographic Primitives used	SHA-1 or MD5
Strength	128 byte
Statistical Test Results available	DIEHARD battery
Speed	8 - 12K bits/s [1]
Portability	part of Linux Kernel

Table 9.1: Summary of /dev/random

[1] By convention, K corresponds to 1024 and M to 1024^2 bits.

Chapter 10

Yarrow

Yarrow is a generic concept for building RNGs. It was developed at Counterpane Systems by N. Fergueson, J. Kelsey, and B. Schneier [KSF99]. Additional information and source code can be found in [Sch03].

Yarrow was designed to prevent attacks as they are described in [KSWH98] and to assure high efficiency and portability. Additionally, the generic structure allows to adjust each component of the generator to the individual needs of the user and to the current state-of-the-art.

In this chapter we will discuss the concept of Yarrow and introduce Yarrow-160 as a specific implementation.

10.1 General Structure

Yarrow employs a cryptographic hash function to feed input into two separated pools, the fast pool $\mathcal{P}_f$ and the slow pool $\mathcal{P}_s$. If enough entropy is gathered into the two pools, then a secret key $\mathcal{K}$ for a block cipher is generated from their content. The random numbers are produced by encrypting the value of a counter $\mathcal{C}$ by means of the encryption function and the secret key.

Yarrow uses established cryptographic primitives like hash functions and block ciphers and can therefore benefit from their resistance against cryptographic attacks. Both primitives have to satisfy certain conditions. The hash function employed in Yarrow must fulfill the requirements for a cryptographic hash function, i.e., it must be collision-free, one-way and the results of any arbitrary set of input samples must be distributed uniformly over the whole output space of the function. The chosen block cipher must resist known-plaintext and chosen-plaintext attacks and its output must meet the statistical standards for random numbers, even if the plaintext or the key are highly patterned. Every cryptographic primitive that satisfies this requirements can be used within the generic concept. Yarrow-160 choses SHA-1 (Secure Hash Algorithm) as hash function and Triple-DES (Triple Data Encryption Standard) as block cipher.

The concept of Yarrow consists of four main components:

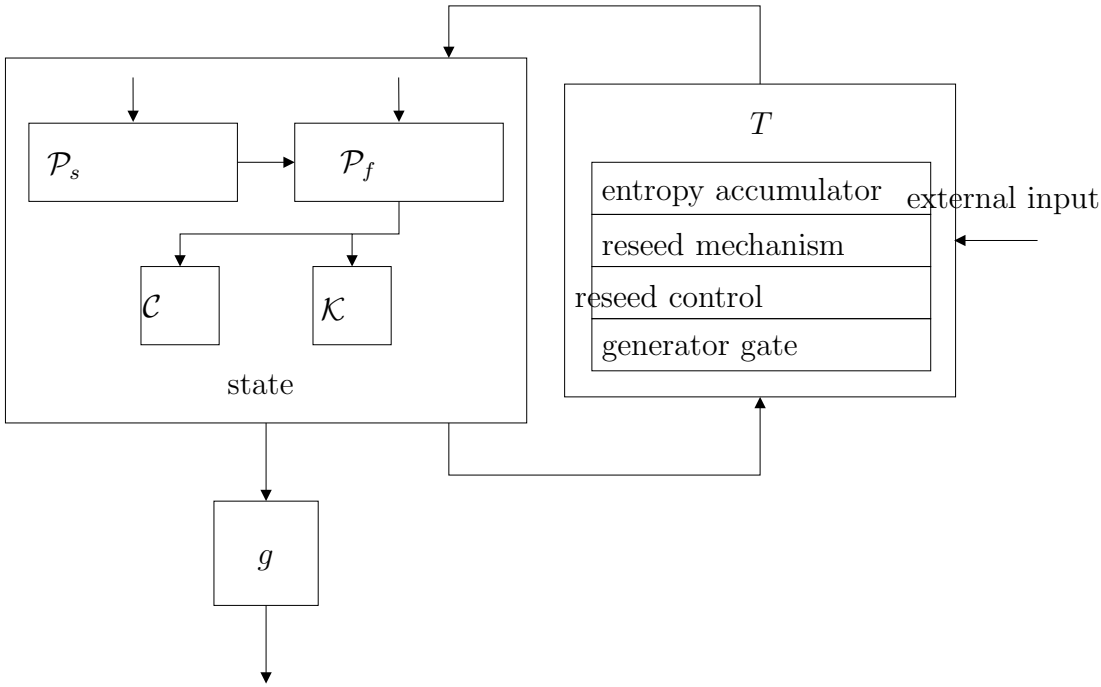

Figure 10.1: General structure of Yarrow

1. The *Entropy Accumulator* processes the input from external sources. It is responsible for mixing the input into the two pools and for estimating the collected entropy.

2. The *Reseed Mechanism* generates a new key from the pools if they are in an unpredictable state.

3. The *Reseed Control* initiates a reseeding of the key if enough entropy is collected in the pools.

4. The *Generation Mechanism* applies a block cipher and the secret key to produce random numbers from the value of a counter. It also changes the key periodically without using the content of the two pools. This replacement of the key is called *generator gate*.

The input is fed continuously into the two pools using the *entropy accumulator*. If the two pools contain enough entropy the *reseed control* initiates a reseeding of the secret key, whereas the reseeding procedure is accomplished by the *reseed mechanism*. The *generation mechanism* produces the random numbers and renews the key periodically without using the content of the two pools.

The transition function involves the entropy accumulator, the reseed mechanism, the reseed control and the generator gate of the generator mechanism. The output function only involves the generation of the random numbers within the generation mechanism.

10.2 State Space

The state of Yarrow consists of two pools, the fast pool $\mathcal{P}_f$ and the slow pool $\mathcal{P}_s$, a secret key $\mathcal{K}$ and a counter $\mathcal{C}$. Input from external sources is fed into the two pools. If the estimated entropy of the pools exceeds a given threshold, then $\mathcal{K}$ and $\mathcal{C}$ are refreshed using the pools. At this time, the content of the pool is assumed to be unguessable for any observer. The fast pool is responsible for frequent reseeding of the key. The slow pool is used to guarantee a less frequent but high quality reseeding. Finally, a block cipher employs the secret key to encrypt the value of the counter for generating the random numbers. The size of the counter corresponds to the block size of the block cipher, thus, in Yarrow-160 we use a 64-bit counter.

Each pool has its own entropy counter for each separate input source. Let us assume that there are three different entropy sources. Then the fast pool contains the counter fec_1, fec_2 and fec_3, and the slow pool contains sec_1, sec_2 and sec_3 (see Figure 10.2). If

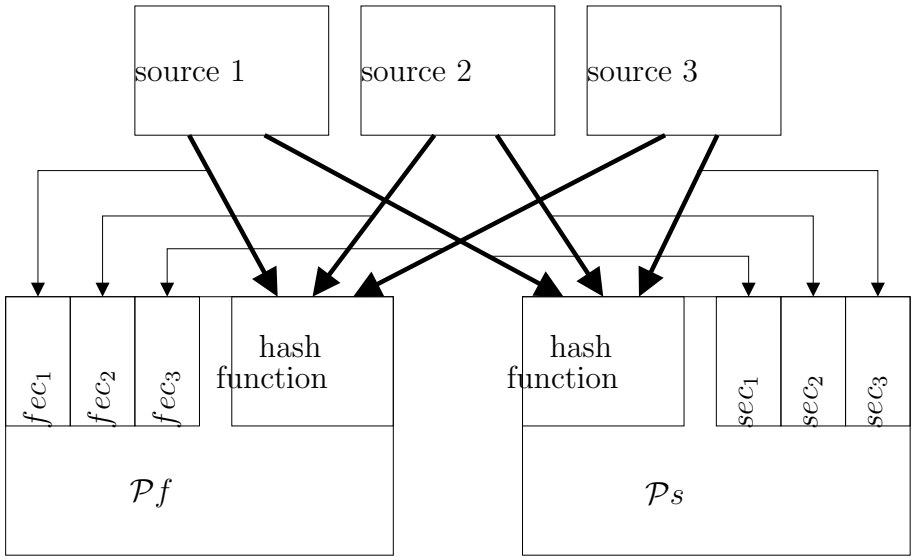

Figure 10.2: Entropy counter of the two pools

the entropy gathering component processes any input, then, at first, the entropy of the input is estimated. Subsequently, the input is mixed into one of the two pools and the corresponding entropy counter is increased. For example, if input from source 1 is mixed into the fast pool then fec_1 is increased. Likewise, if input from source 3 is mixed into the slow pool then sec_3 is increased. As soon as a pool has been used for reseeding, all its entropy counts are reset to zero.

Input is always fed into the two pools alternatingly. The content of a pool represents the hash value over the whole previous input of the pool, concatenated together. Yarrow employs a hash function as well as a block cipher. Let us assume that the hash function has an m-bit output and the block cipher uses a k-bit key. The strength of the generator is then limited to $\min(m, k)$ bits. The k bits of the key are responsible for generating the

random number, but as the key is created from the pools, its entropy cannot exceed the size of a pool (m bits). Yarrow-160 uses SHA-1 as its hash function and Triple-DES as its block cipher. Thus, $m = 160$, $k = 192$, and the strength of Yarrow-160 is limited to 160 bits.

10.3 Transition Function

The transition function consists of the entropy gathering, the reseed control, the reseed mechanism, and the generator gate.

10.3.1 Entropy Gathering

Input from different sources is fed into the two pools alternatingly. At first, the entropy of the input is estimated. Then the input is mixed into the pools and the corresponding entropy counter is increased. Each pool represents the hash value of its whole input concatenated together. This can be realized by continuously feeding the hash function with the new input and storing the current hash digest in the pool. To collect all the uncertainty from the input, the hash function should satisfy the following conditions:

1. Almost all entropy of the input should be preserved in the hash value independent of the construction of the input string.

2. It should not be possible to find any input strings which are able to reduce the current entropy of the hash value.

3. It must be impossible for an adversary to force the pool into a weak state, such that the pool is unable to collect new entropy.

4. If an adversary can choose which bits in which input string he knows but does not know n bits, there must still be 2^n possible states of the pool.

We need a cryptographic hash function to satisfy all this requirements. As stated before, in Yarrow-160 SHA-1 is used.

The designers of Yarrow claim that entropy overestimation is one big flaw of other RNGs. Yarrow therefore follows a quite conservative estimation policy. The amount of entropy of each source and each pool is estimated and recorded separately and is reset after each reseeding. Because of the separated entropy counters, the overestimation of one source cannot jeopardize the whole entropy estimation. The reseeding control uses the entropy counter of the pools to determine the next reseeding of the key.

The entropy of each input string is estimated in three steps:

1. Each program that provides input from a specific source like inter-keystroke-timing or the noise of an unplugged microphone, has to provide an entropy estimation for every generated input.

2. In addition, Yarrow uses a statistical estimator to calculate the entropy of the string. No information about the details of this estimator are available in the documentation of the generator.

3. The third reference value represents a global maximum entropy density. The authors of [KSF99] assume that a string does not contain more than p percent of entropy independent of the input source. Therefore, they limit the entropy estimation to $p * l(i)$ bits, where $l(i)$ is the length of the input i. In Yarrow-160, p is set to 50%.

As final entropy estimator, the minimum of all three values is applied.

10.3.2 Reseed Control

The reseed control initiates a reseeding of the secret key as soon as the pools are assumed to be in an unguessable state. Each pool counts the estimated entropy separately for each input source and resets the counter to zero after a pool has been used for reseeding. The entropy counts are applied to indicate when the pool reaches an unguessable state.

The fast pool $\mathcal{P}_f$ is responsible for frequent reseeding. As soon as the entropy count of one source exceeds a given threshold t_f, the key is reseeded from the fast pool (e.g.: $fec_2 > t_f$). This reseeding guarantees that the generator swiftly recovers from a compromised key. In Yarrow-160 is $t_f = 100$ bits.

The slow pool enables high quality reseeding. Only if the entropy count of at least r sources exceeds the threshold t_s the key is reseeded (e.g.: if $r = 2$, $sec_1 > t_s$, and $sec_3 > t_s$). This behavior avoids that the overestimation of one input source endangers the entropy of the key. A reseeding of the slow pool always uses the fast pool as well. In Yarrow-160 $r = 2$ and $t_s = 160$ bits. r may be adjusted for different environments, e.g., if there are three good and fast sources of entropy, r may be set to 3.

10.3.3 Reseed Mechanism

The reseed mechanism generates a new k-bit key from the content of the pools and the old key and refreshes the content of the counter. We will discuss the reseeding from the fast pool. If both pools are applied for the reseeding, then the content of the slow pool is first fed into the fast pool before the key is generated from the fast pool. After the content of a pool was used for reseeding, all its entropy counters are reset to zero. The generation of the new key is done is several steps.

Initialization: At first, $v_0 := \mathcal{P}_f$ is set to the content of the fast pool, where $\mathcal{P}_f = h(i_f)$ is the result of the the hash function h on the concatenated input i_f of the fast pool.

Iteration: In the second step, the hash function is iterated in the following way,

$$v_i = h(v_{i-1}|v_0|i) \text{ for } i = 1, ..., N_t.$$

Here, the symbol | represents the concatenation operator and $N_t \geq 0$ determines the number of iterations. The goal of this iterative calculation is to make the reseeding computationally more complex. Thus, attacks based on guessing input are made much harder, but this procedure also reduces the efficiency of the generator.

Key Generation: This step uses the old key $\mathcal{K}$ and the hash function h to create the new key $\mathcal{K}'$. The function h' is responsible for creating a key of length k independently of the length m of the hash digest (e.g. in Yarrow-160 $m = 160$ and $k = 192$). The new key is calculated by

$$\mathcal{K}' = h'(h(v_{N_t}|\mathcal{K}), k)$$

where $h'(M, k)$ is defined by

$$\begin{aligned} s_0 &:= M \\ s_i &:= h(s_0|...|s_{i-1}) \\ h'(m,k) &= \text{ first } k \text{ bits of } (s_0|s_1|...). \end{aligned}$$

Reset Counter: As a last step, the counter is initialized with $\mathcal{C} = E_{\mathcal{K}}(0)$, where $E_{\mathcal{K}}$ represents the encryption function of the block cipher with key $\mathcal{K}$.

In the end all variables are cleared from any storage, the entropy count of the used pools is reset and the seed file is filled with $2k$ bits. If no seed file is used, then the last step can be ignored.

10.3.4 Generator Gate

To prevent backtracking attacks, the key $\mathcal{K}$ of the block cipher is renewed from time to time, without reseeding it from any pool. Kelsey et al. [KSF99] call this process a *generator gate*. After every N_g output blocks the output function is employed to create k bits, which are not used as an output but to reset the key. A generator gate is not a reseeding because no external entropy is added. If an adversary can compromise the secret key and the current value of the counter at a given time, he or she is able to reconstruct the previous outputs of the RNG since the last generator gate or the last reseeding of the key. Let $\mathcal{C}$ be the n-bit counter. If more than $2^{\frac{n}{2}}$ bits are generated from one key, collisions get more and more likely by the birthday paradox [MvOV01, p. 53]. The authors in [KSF99] thus recommend to choose N_g in the range $1 \leq N_g \leq 2^{\frac{n}{3}}$. A smaller N_g reduces the number of previous output bits an adversary can learn from a compromised key but also slows down the RNG. In Yarrow-160 the parameter N_g is set to 10.

10.3.5 Input Space

The size of the input strings is unlimited. The SHA-1 hash function only accepts 512 bit blocks at once, larger strings are processed block by block. In addition, the number of

input strings between two output generations is not restricted. Thus, the input space for the transition function is theoretically infinite.

10.4 Output Function

The output function encrypts the content of an n-bit counter, by means of a block cipher and the secret key $\mathcal{K}$, to generate the random number. The n-bit counter $\mathcal{C}$ is incremented every time a block of random numbers is produced,

$$\mathcal{C}_{m+1} = \mathcal{C}_m + 1 \pmod{2^n}.$$

The output of the generator is produced by simply encrypting the content of $\mathcal{C}$,

$$o_m = E_{\mathcal{K}}(\mathcal{C}_m).$$

The number of generated output blocks between two reseedings is limited to $\min(2^n, 2^{\frac{k}{3}} N_g)$. The first value prevents cycling of the counter. The second one should prevent that two identical keys are produced by the generator gate. Due to the birthday paradox, the chances of two equal keys exceeds 50% if more than $2^{\frac{k}{3}} N_g$ output blocks would be generated.
The Yarrow-160 implementation uses a three key Triple-DES as the block cipher.

10.5 Security

Yarrow was especially designed to prevent all the attacks described in [KSWH98]. By the separation of the pools and the key, input samples do not have an intermediate influence on the output, which avoids iterative guessing attacks. The fast pool allows the generator to quickly recover from a compromised key, the slow pool prevents damage caused by overestimation of the input entropy. Due to the generator gate the number of output bits that can be leaned through a backtracking attack is limited. Yarrow further profits from the resistance of the block cipher against cryptographic attacks and the collision immunity of the hash function.

Possible weaknesses of the generator are the strength and bad input samples. The strength of the Yarrow-160 generator is only 160 bits. This can be improved by using a hash function with a larger hash digest and a block cipher with a larger key size. Although Yarrow implements many features to compensate an overestimation of the input entropy, very inappropriate samples can still weaken the generator. Therefore, one should carefully select the input sources.

10.6 Empirical Results

No empirical test results for Yarrow could be found in published literature. The results from such a test would depend on the used cryptographic primitives like the hash function

or the block cipher on the one hand and on the quality of the entropy sources on the other hand. To obtain such statements about the quality of the generated random numbers, it would be advisable to not only use statistical tests on the output of the generator, which would be highly influenced by the block cipher, but also to test the input sources used.

10.7 Portability

Yarrow is a design concept for RNGs. Thus, specific implementations can be adjusted to any platform and to any specific requirements of the user. [Sch03] provides a C-implementation of Yarrow-160 for WinNT and Win95 operating systems. A further implementation of Yarrow is applied as random device in the Mac OS X operating system.

10.8 Conclusion

The yarrow generator is an attack oriented concept for implementing RNGs. Single components can easily be replaced and adjusted to individual needs. [KSF99] analyzes the particular elements of the generator and their behavior during attacks. Unfortunately, there exist no results of empirical tests or efficiency of specific implementations of a yarrow based generator. Furthermore, Counterpane ended their support of Yarrow. In any case, Yarrow is a simple and robust possibility to combine entropy gathering with a PRNG (The counter and the block cipher together are nothing else but a PRNG). A drawback of Yarrow is the small strength, but one may weaken this problem by choosing an appropriate hash function and block cipher.

Factsheet for Yarrow	
Cryptographic Primitives used	SHA-1 3DES
Strength	160 bits
Statistical Test Results available	None
Speed	No results
Portability	Yes

Table 10.1: Summary of Yarrow

Chapter 11

Blum-Blum-Shub Generator

The Blum-Blum-Shub generator (BBS), or x^2 mod N generator as it is called sometimes, is an example of a nonlinear congruential generator. It was first introduced by L. Blum, M. Blum, and M. Shub in [BBS86]. The generator does not process any input after it was seeded once and, thus, falls into the PRNG category.

The main property of the BBS generator is that it is proven to be next-bit unpredictable to the left and to the right, if the prime factors of a large integer N are unknown. Consequently, the security of the generator is based on the intractability of factoring large integers, which is, for example, also used in the RSA encryption scheme. Next-bit unpredictability to the right (to the left) means that for every positive polynomial p and for sufficiently large m the following condition holds. If $m-1$ consecutive output bits are known, then for every polynomial-time algorithm it is impossible to guess the next (previous) output bit with a probability larger than $\frac{1}{2} + \frac{1}{p(n)}$ (see also Definition 5.5).

The BBS generator is quite slow, but due to the characteristic explained above, it is often recommended for cryptographic tasks which only need a small amount of random numbers.

11.1 General Structure

The BBS generator is part of the PRNG category. Consequently, the production of random numbers is totally determined after a seed was chosen. At first, we need two prime numbers P, Q which satisfy the condition

$$P \equiv Q \equiv 3 \pmod 4.$$

The *Blum integer* N, as it is called in [Sch93], is set to

$$N = PQ.$$

As a last preparation we need an integer $x \in [1, N-1]$ which is relatively prime to N.

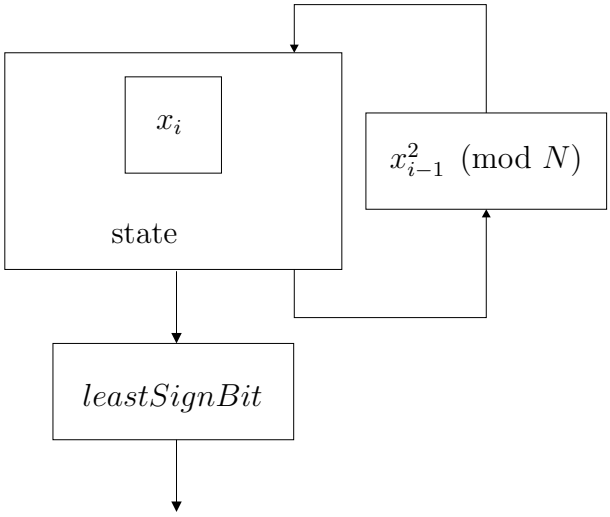

Figure 11.1: General structure of BBS

11.2 State Space

The generator is seeded with

$$x_0 \equiv x^2 \pmod N.$$

The variable x_i represents, for all $i \geq 1$, the current state of the generator, which is only changed by the transition function T.

For a given N the strength of the generator is limited to $\phi(N)$. Here, ϕ is Euler's totient function, which specifies the number of elements in $\mathbb{Z}_N$ that are relatively prime to N and consequently, it represents the quantity of all possible seeds. Not all seeds produce an adequate period length. Thus, in practice, the strength is much smaller than $\phi(N)$.

Since no input is processed, the generator is periodic. The authors show in [BBS86, p. 377, Theorem 6] that $\lambda(\lambda(N))$ is a multiple of the period length of the output. Here, $\lambda(N)$ is Carmichael's function.

Definition 11.1 (Carmichael's λ function)
Let $M = 2^e P_1^{e_1} \dots P_k^{e_k}$, where $P_1, \dots . P_k$ are distinct odd primes. Carmichael's λ function *is defined by*

$$\lambda(2^e) = \begin{cases} 2^{e-1} & \text{if } e = 1 \text{ or } 2, \\ 2^{e-2} & \text{if } e > 2, \end{cases}$$

and $\lambda(M) = lcm[\lambda(2^e), (P_1 - 1)P_1^{e_1 - 1}, \dots, (P_k - 1)P_k^{e_k - 1}]$.

Furthermore, Blum, Blum, and Shub demonstrate in [BBS86, p. 377 f.] that the full period can only be reached if the following conditions are satisfied.

- P and Q are special prime numbers (see Definition 11.2), where

$$\begin{aligned} P &= 2P_1 + 1, \\ Q &= 2Q_1 + 1, \end{aligned}$$

  and P_1 and Q_1 are prime numbers corresponding to (11.1).

- Since 2 is a quadratic residue with respect to at least one of the two prime numbers P_1 and Q_1, there exists a y such that

$$\begin{aligned} y^2 &\equiv 2 \pmod{P_1} \text{ or} \\ y^2 &\equiv 2 \pmod{Q_1}. \end{aligned}$$

- For the seed $x_0 = x^2$, the smallest integer k such that $x_0^k \equiv 1 \pmod{N}$ must be equal to $\frac{\lambda(N)}{2}$.

Definition 11.2 (Special prime number)
A prime P is special *if*

$$P = 2P_1 + 1, \tag{11.1}$$
$$P_1 = 2P_2 + 1, \tag{11.2}$$

and P_1, P_2 are odd primes.

Due to the requirements above we see that only a limited number of Blum integers N and seeds x guarantee a full period length. Blum, Blum, and Shub showed that there exists an algorithm that determines the period length of a specific seed in polynomial time.

11.3 Transition Function

The transition function $T : \mathcal{S} \to \mathcal{S}$ depends only on the current state of generator and not on any external input

$$x_i \equiv x_{i-1}^2 \pmod{N} \qquad \forall i \geq 1.$$

11.4 Output Function

The original BBS generator takes the least significant bit of x_i as an output,

$$o_n = leastSignBit(x_i).$$

The performance of the generator can be increased without loosing security, by using the least significant $\log_2(l_i)$ bits as an output, where l_i is the length of the state x_i. Therefore, more bits can be produced during each iteration.

An advantage of this generator is that the i'th output can be calculated directly by using the least significant bit(s) from

$$x_i = x_0^{2^i \bmod ((P-1)(Q-1))} \pmod{N}.$$

Therefore, several output bits can be calculated in parallel, which also improves the speed of the generator.

11.5 Security

The BBS generator is proven to be next-bit unpredictable to the left and to the right if the prime factors of N are unknown. The security is thus based on the inability of factoring large integers into their prime components. The same security assumption is used for RSA encryption.

For a good BBS generator with a long period time the seed x_0 and the prime numbers P and Q must satisfy several requirements. It is not possible to use every arbitrary random number for seeding the generator and, consequently, the strength of the generator is reduced. Since this generator does not process any input, the unpredictability of the output is completely based on the unpredictability of the seed. If the seed is guessed once the whole output of the generator can be calculated. Due to this property, a reduced strength has a more severe impact than on entropy gathering or hybrid generators.

The BBS generator is certainly also vulnerable to all PRNG specific attacks. If the process, which holds the generator gets forked, then all future generated random numbers of the original and the cloned process are identically.

11.6 Empirical Results

The BBS generator passed the NIST test suit as can be seen in [Sot99]. The throughput of the generator was measured in [Ced00] and reached a value of 3 Kbits/s on a Pentium III 700 processor. The generator is thus much slower than AES or our hybrid generators.

11.7 Portability

The BBS generator is a mathematical algorithm and does not process any input. It can therefore be implemented in any programming language and any operating system.

11.8 Conclusion

The BBS generator does not process any input and can thus be applied when a reconstructible sequence of random numbers is desired like in the case of stream ciphers.

However, this also means that the generator never recovers from a compromised state or seed. It is highly vulnerable to any attack which clones the process. Another flaw of this generator is the limited number of appropriate seeds, since not all seeds guarantee the full period length of the generator. For large N and small amounts of required random bits, this may not be a severe problem.

The costly search of accurate seeds makes the BBS generator inappropriate in connection with frequent random reseeding. Consequently, it is not suitable for producing large amounts of random numbers. In any case the quality and unpredictability of the seed should always be checked carefully, before using the generator.

Factsheet for BBS	
Cryptographic Primitives used	None
Strength	$\leq \log_2 \phi(N)$
Statistical Test Results available	NIST test suite
Speed	3 Kbits/s
Portability	Yes

Table 11.1: Summary of BBS

Chapter 12

AES

AES (Advanced Encryption Standard) became a Federal Information Processing Standard for encrypting and decrypting data on November 26th, 2001 (see FIPS PUB 197 [Nat01a]). It represents limited cases of the Rijndael algorithm, which was designed by Daemen and Rijmen [DR02]. For AES the length of the data blocks is fixed to 128 bits and the length of the cipher key can vary between 128, 192, or 256 bits.

Hellekalek and Wegenkittl proposed in [HW03] the usage of AES as a random number generator for statistical purpose. We will study its qualities as a RNG for cryptographic applications. Generally, every other cryptographic block cipher could be applied as random number generator instead of AES. The benefits of AES are the high speed and the possible key sizes up to 256 bits. In comparison to this, Triple-DES has only a key length of 192 bits. Furthermore, the quality of the generator was exhaustively tested in [HW03] by means of state-of-the-art statistical tests.

12.1 General Structure

AES is originally a symmetric block cipher for encrypting data. [HW03] discusses two different ways of using AES as a random number generator, AES in *Counter mode* and AES in *PRNG mode*. The first one encrypts the value of a counter, the second one iteratively applies the encryption algorithm on the inner state of the generator. Neither of the two modes process any input, therefore both versions of the AES generator fall into the PRNG category.

In our further discussion we will fix the length of the cipher key to 128 bits as was done in all the tests of [HW03]. However, for optimal strength we suggest to use a key of length 256 bits. $E_{\mathcal{K}}(x)$ represents the encryption of x using AES and the key $\mathcal{K}$.

The fist kind of random number generator discussed in [HW03] uses AES in Counter mode (see Figure 12.1). A 128 bit Counter $\mathcal{C}$ starts in c_0 and generates the sequence $c_0, c_1, ...$, where $c_i = c_{i-1} + 1 \pmod{2^{128}}$. The output of the generator at time i is $E_{\mathcal{K}}(c_i)$. As c_i has 128 bits, the counter and the output have both the full period length of 2^{128}. Furthermore, this mode allows to calculate each output value independently of its successor,

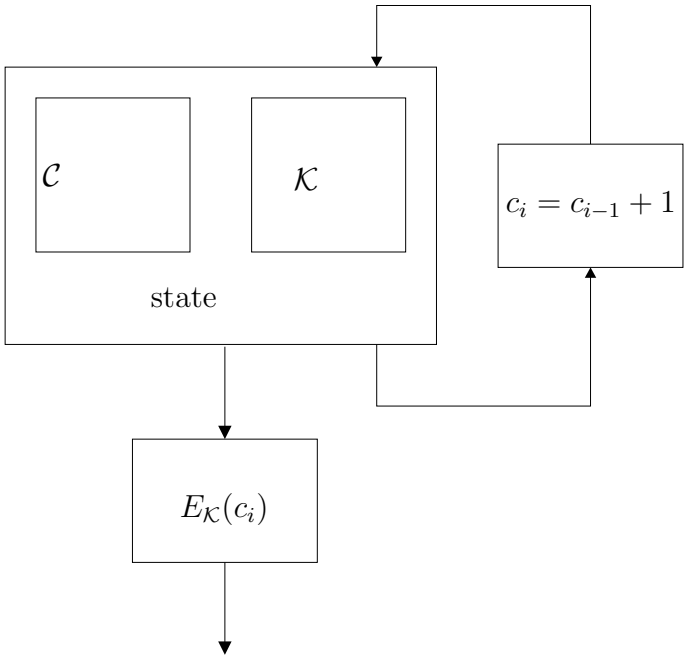

Figure 12.1: General structure of AES in Counter mode

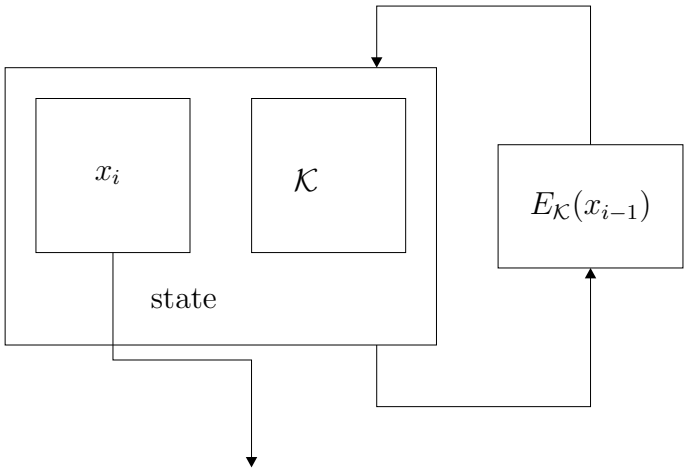

Figure 12.2: General structure of AES in PRNG mode

which supports parallel processing.

The second kind uses AES in PRNG mode (see Figure 12.2). The sequence of random numbers $x_0, x_1, ...$ is produced by successive usage of AES, e.g. in Output Feedback mode (OFB). This means the output x_i is generated by $x_i := E_{\mathcal{K}}^{(i)}(x_0) = E_{\mathcal{K}}(x_{i-1})$. The period length of the output may be less than 2^{128}, if there exist a $j < 2^{128}$ such that $E_{\mathcal{K}}(x_{i-1}) = x_{i-j}$.

12.2 State Space

The internal state of the generator consists of the cipher key and the current value of the counter (Counter mode) or the current value of x_i (PRNG mode), respectively.

In Counter mode, an adversary has to guess the key $\mathcal{K}$ and the current content c_i of the counter. Consequently, in our setting we reach a strength of 256 bits. This value can be improved to 320 or 384 bits, by using key sizes of 192 or 256 bits.

In PRNG mode, the current value of x_i is also the output of the generator and can be therefore not counted as part of the secret internal state. An adversary only has to guess the key $\mathcal{K}$ to predict the generated random numbers. Thus, this mode achieves a strength of 128 bits in our setting or 192 or 256 bits, with longer choices of keys.

By using Rijndael instead of AES with a block size of 256 bits and a key length of 256 bits, we could even enhance the strength up to 612 bits in Counter mode and respectively, up to 256 bits in PRNG mode.

[HW03] discusses AES as a RNG for statistic applications, the main focus was set on the statistical quality of the output. Hellekalek and Wegenkittl studied the impact of different structures and regularities in the key or the plaintext (i.e., c_i or x_i, respectively) on the output of the generator. The article showed that the generated random numbers passed all statistical tests independent of any regularities in the seed. Nevertheless, if the generators are used in a cryptographic context, it is necessary for both generator modes that the key as well as the initial values c_0 and x_0, respectively, are seeded from a reliable random source. Only in this way the full strength of the generator can be guaranteed. For constant security, regular reseeding would be desirable. [HW03] does not mention any specific reseeding mechanism, but suggestions may be found for example in Chapter 10.

12.3 Transition Function

In Counter mode the transition function simply increments the counter,

$$c_i = c_{i-1} + 1 \pmod{2^{128}}.$$

The PRNG mode uses AES repeatedly on the inner state x_i of the generator,

$$x_i = E_{\mathcal{K}}(x_{i-1}).$$

Since the AES algorithm is also used as output function for the Counter mode we will defer the detailed description of the algorithm to Section 12.4.

Neither of the two modes processes any input and therefore the output of both generators is periodic. In either case the period length is limited to 2^{128} output blocks, which represent the size of the counter and all possible values of x_i respectively. Whereas in counter mode the full period length is reached, the period length in PRNG mode may be reduced due to the birthday paradox. If there exists a $j < 2^{128}$ such that

$$E_{\mathcal{K}}(x_{i-1}) = x_{i-j},$$

then the period length in PRNG mode is limited to j which is less than 2^{128}. By using a version of Rijndael with a block size of 256 bits we can enhance the limit of the period length to 2^{256} data blocks.

12.4 Output Function (AES)

The output function of the generator in Counter mode is the encrypting function $E_{\mathcal{K}}$ of the AES algorithm, thus for all $i \geq 1$

$$o_n = E_{\mathcal{K}}(c_i).$$

The PRNG mode does not have any output function. A part of the inner state of the generator is directly used as output, therefore,

$$o_n = x_i$$

for all $i \geq 1$.

In this section we will give a short introduction of the AES algorithm. More detailed information can be found in the Federal Standard publication FIPS PUB 197 [Nat01a] and in Daemen and Rijmen's monograph [DR02].

AES is a nonlinear block cipher that encrypts and decrypts input blocks of 128 bits to output blocks of 128 bits. It thereby employs a key of length 128, 192, or 256 bits. We will limit our discussion to the case of a 128 bit key as was used for the tests in [HW03].

The algorithm works on instances of bytes and words (4 bytes). In the following, we will use the hexadecimal representation $\{h_1h_2\}$ to display a byte. In the description of the algorithm a byte is interpreted as a polynomial of degree seven with coefficients in the finite field $GF(2)$, e.g., (01100011) represents the polynomial $x^6 + x^5 + x + 1$. The addition of two polynomials happens coefficient-wise, the multiplication is done modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ and is therefore invertible for all non-zero polynomials. The polynomials together with the addition and the multiplication form a finite field $GF(2^8)$.

A 32-bit word is interpreted as a polynomial of degree three with coefficients in $GF(2^8)$. The addition is again done coefficient-wise, the multiplication is done modulo

the polynomial $c(x) = x^4 + 1$. This polynomial is not irreducible and therefore not all polynomials have an inverse. However, the AES algorithm only uses this multiplication in connection with a constant polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, which does have an inverse. The choices of $a(x)$ and $c(x)$ allows an efficient implementation of the multiplication. If $b = (b_0, b_1, b_2, b_3)$ represents the word we want to multiply with a, then the result $d = (d_0, d_1, d_2, d_3)$ is determined by

$$\begin{aligned}
d_0 &= \{02\} \bullet b_0 \oplus \{03\} \bullet b_1 \oplus b_2 \oplus b_3, \\
d_1 &= b_0 \oplus \{02\} \bullet b_1 \oplus \{03\} \bullet b_2 \oplus b_3, \\
d_2 &= b_0 \oplus b_1 \oplus \{02\} \bullet b_2 \oplus \{03\} \bullet b_3, \quad \text{and} \\
d_3 &= \{03\} \bullet b_0 \oplus b_0 \oplus b_2 \oplus \{02\} \bullet b_3.
\end{aligned}$$

In this equation, $\bullet$ represents the multiplication in $GF(2^8)$ and $\oplus$ the bit-wise XOR operation. The multiplication $\{02\} \bullet b_i$, $0 \leq i \leq 3$, can be implemented by a shift of b_i one bit to left and a bit-wise XOR with the byte $\{1b\}$. Furthermore, $\{03\} \bullet b_i$ is nothing else than $\{02\} \bullet b_i \oplus b_i$. Consequently, the whole multiplication with a can be efficiently implemented by using a couple of XOR and shift operations.

All calculations are done on the `state`, a 4x4 byte matrix. At the beginning, the input block of 16 bytes $in_0, in_1, \ldots, in_{15}$ is copied into the `state`.

$$in_0, in_1, \ldots, in_{15} \Rightarrow \begin{array}{|c|c|c|c|} \hline S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ \hline S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ \hline S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ \hline S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \\ \hline \end{array}$$

Here, $S_{r,c}$ represents the byte in `state` at row r and column c and we set $S_{r,c} = in_{r+4c}$. In a next step, an initial 4-word `RoundKey` is added to the state, by combining each word with a column of the `state`. Subsequently, a sequence of manipulations is applied to the `state` 10 times in a row. The number of iterations is necessary to prevent short cut attacks. Such an attack would allow to obtain the key with less effort than brute force. For more information about design decisions see [DR02]. Each iteration consists of the following manipulations:

- The first manipulation does a byte substitution by means of a non-linear, inverse, 2-dimensional matrix, the `s-box`. This substitution represents two steps. In the beginning the inverse of each byte is calculated. $\{00\}$ is assigned to itself. The second step applies the following affine transformation on the byte,

  $$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus d_i,$$

  where b_i represents the i'th bit of the byte we want to transform, b'_i is the i'th bit of the transformed byte, and d_i is the i'th bit of the constant byte $d = \{63\}$. The operator $\oplus$ denotes the XOR operation. Both steps are combined together in the `s-box`, where each original byte is assigned to its transformed value.

- The next manipulation rotates each row of the `state` by r positions, where $0 \leq r \leq 3$ represents the number of the row. For example the third row rotates like

$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$\Rightarrow$	$S_{2,2}$	$S_{2,3}$	$S_{2,0}$	$S_{2,1}$

- In the third step the columns of `state` are mixed by multiplying them with the invertible polynomial $a(x)$. This manipulation is done in all iterations except of the last one.

- In the end a new `RoundKey` is added to the `state`.

After the 10 iterations, the `state` matrix contains the encrypted data blocks.

The `RoundKeys` are extracted from the cipher key $\mathcal{K}$ and are stored in a linear (4*11)-word array. The first four word of the array are equal to the four word of the key. The remaining word are generated by repetitively using the `S-box`, by rotating the bytes within a word, and by multiplying the words with polynomial of the form x^{i-1}, where $i \geq 1$ and $x = \{02\}$. The exact algorithm can be found in the literature mentioned above.

12.5 Security

AES was developed and analyzed to be highly secure against cryptographic attacks. Consequently, this security also holds for both generator modes. Part of the security feature is realized by the consideration of *confusion* and *diffusion*. Confusion means that no analysis of the ciphertext (in our case the output of the generator) gives any information about the key. Diffusion means that, independently of the key, no simple structure in the plaintext should lead to simple structures in the ciphertext. For the use of AES as an RNG this means that it is practically impossible to gain the inner state of the generator out of the output, or to guess the output if part of the inner state is known to the adversary.

This also means that if we would reseed AES from an entropy pool and an adversary would be able to inject some regularity into the pool by a chosen input attack, this would have no effect on the structure of the output. Only the strength of the generator could be reduced. If the reduced strength is still large enough to resist a brute force attack on the inner state, then the generated random numbers may still be secure.

However, a severe problem occurs if the inner state of the generator is compromised once. Due to the bidirectional structure of both generator modes, it would then be possible to calculate all future and previous random numbers. Thus, it is important for the RNGs to provide a high strength, which makes it harder for an adversary to guess the current state. This fact makes AES in PRNG mode less suitable for cryptographic applications, since this generator uses part of the inner state directly as an output which lowers the strength by 128 bits.

A possible challenge of AES in Counter mode would be a timing attack on the value of the counter (see Section 8.1.2). If the content of the counter is known by an adversary,

then the strength of this mode would be reduced by the 128 bits of the counter. We can prevent these attacks by using a counter which always takes the same amount of time for incrementing its value.

Both generator modes are part of the PRNG category, consequently, their output is periodic. To avoid that any adversary may benefit from this fact, we have to provide frequent reseeding from a high quality random source.

12.6 Empirical Results

Statistical Tests

In [HW03], Hellekalek and Wegenkittl tested AES with respect to its usage as a RNG for stochastic simulations. They studied possible impacts of irregularities in the key or the plaintext on the output of the generators. For this purpose, they used four different settings:

1. In the first setting, the key $\mathcal{K}$ and the initial value x_0 where both set to all zeros and a sequence
$$x_i = E_{\mathcal{K}}(x_{i-1})$$
for $i \geq 1$ was generated.

2. Subsequently, $\mathcal{K}$ was set to all zeros and a sequence of highly patterned plaintext was encrypted. Each plaintext consisted of exactly six ones, which were set into all possible places in the 128 bits of the input block.

3. In the third setting $\mathcal{K}$ was again set to all zeros and the content c_i of a 128-bit counter was encrypted,
$$x_i = E_{\mathcal{K}}(c_i), \text{ for } i \geq 1.$$

4. In the last setting the plaintext p_0 was fixed to all zeros and for each new output the key was set to c_i, the value of a counter,
$$x_i = E_{c_i}(p_0), \text{ for } i \geq 1.$$

In all settings, the output was transformed into a binary sequence by concatenating the output blocks.

Subsequently, two different test setups were applied to the produced binary data. In both setups, the main test was employed 16 times in a row to each parameter pair (n, d), where n is the length of input sample and d is dimension of the overlapping tuples. The results of each parameter pair was then compared to its corresponding distribution by calculating the 1-sided Kolmogorov-Smirnov p-values.

The goal of the first setup was to detect irregularities that were originated in the byte-orientated design of AES. For this purpose, only every 8th bit of the original sequence

was used for the test. Subsequently, this data was used as input for an overlapping serial test with parameters $d \in \{1, 2, 4, 8, 16\}$ and $n \in \{2^{20}, 2^{21}, \ldots, 2^{28}\}$. The corresponding 16 results were then compared with the chi-square distribution of $2^d - 2^{d-1}$ degree of freedom. The numbers of rejections of the 180 p-values at the 5% and the 1% level were 7 and 4, respectivly. This result can be regarded as totally noncritical.

The second setup used the *gambling*-test (as is explained in [Weg99]) with parameters $d \in \{32, 64, 128, 256\}$, $n \in \{2^{22}, 2^{21}, ..., 2^{28}\}$ and $t = \frac{d}{2}$, where t determines the amount of dimension reduction. The aim of this setup was to find any irregularities in the distribution of 0's and 1's even in higher dimensions. Of 112 p-values, only 5 and 1 where rejected on the 5% and the 1% level, respectivly. This indicates that in bit vectors up to the length of 256 bits (this match two output blocks) no significant irregularity could be found.

The test showed that even highly unbalanced distributions of 1's can be compensated and practical no conclusion from the output to the key can be made. For a more detailed description of the test see [HW03].

Throughput

In [DR02, p62], Daemen and Rijmen list the performance of the Rijndael algorithm for several processors. For example on an 800MHz Pentium II the algorithm could reach a throughput of 426 Mbit/s. Both generator modes mainly consists of the AES encryption (the Counter mode additionally applies an incrementation of the counter). Thus, we can assume the same magnitude of throughput for both random number generators.

12.7 Portability

The AES algorithm does not employ special properties of the hardware or the processor. Consequently, both generator modes may be used on arbitrary machines and operating systems. C/C++ code of the AES algorithm alone can be found for example at `http://fp.gladman.plus.com/cryptography\_technology/rijndael/`.

12.8 Conclusion

AES was examined as a PRNG for stochastic applications in [HW03]. The test showed that the quality of the output of a random number generator using AES does not depend on the structure of the key or the plaintext. Thus, any key and initial value arbitrarily chosen from a random source produces binary sequences of high stochastic quality.

For cryptographic applications we would suggest to use AES in Counter mode, since it provides a higher strength and period length than the PRNG mode. The AES generator stands out from other PRNGs because it combines high speed with cryptographic security. Consequently, if we combine AES in Counter mode with a reliable random source and a suitable reseeding mechanism, we get a fast RNG for cryptographic applications.

Factsheet for AES	
Cryptographic Primitives used	AES
Strength	Counter mode: 256 bits PRNG mode: 128 bits
Statistical Test Results available	Overlapping serial test, gambling test
Speed	426 Mbits/s
Portability	Yes

Table 12.1: Summary of AES

Chapter 13

HAVEGE

The HAVEGE (HArdware Volatile Entropy Gathering and Expansion) generator produces random numbers using the uncertainties which appear in the behavior of the processor after an interrupt. This generator was developed by Sendrier and Seznec at INRIA (Paris, FR) and is described in [SS02] and [SS03]. The authors distinguish between HAVEG (HArdware Volatile Entropy Gathering) and HAVEGE. Whereas HAVEG gathers entropy only in a passive way, HAVEGE uses the data already collected to additionally affect the behavior of the processor. The main part of this chapter is devoted to HAVEGE, but a description of HAVEG can be found in Section 13.2.1. In the following, many statements apply to both generators. For simplicity we will discuss only the more powerful version, HAVEGE, in such cases.

HAVEGE extracts entropy from the uncertainty that is injected by an interrupt into the behavior of the processor. Most random number generators that use entropy gathering techniques obtain their entropy from *external* events, like mouse movements or input from a keyboard. Each of those events causes at least one interrupt and thus changes some *inner* states of the processor. In which way those states are changed will be discussed later (see Section 13.1). HAVEGE gathers the uncertainty produced in the behavior of the processor by means of the hardware clock counter. Sendrier and Seznec showed in [SS02] that during one operating system interrupt, thousands of volatile states are changed within the processor. Thus, the amount of entropy which can be collected by this technique is considerably larger than the entropy of the original event.

Therefore, even HAVEG, which only uses the entropy gathering mechanism, still achieves a much higher throughput (about 100 Kbits/s) then other entropy gathering generators. For example, /dev/random only delivers a few bytes per second on an inactive machine.

Additionally, HAVEGE uses random data already collected, to affect the behavior of the processor itself. Consequently, this generator is less dependent on the occurrence of interrupts and achieves a throughput of more than 100 Mbits/s.

13.1 General Structure

Before we discuss the structure of HAVEGE, we shall give a short introduction to those components of the processor which are important for understanding the functionality of the generator. The intention of our introduction is to give an overview over the operation mode of the processor. However, implementations of specific processors may vary from our description. A detailed discussion of this topic can be found in [HP02].

13.1.1 Optimization Techniques of the Processor

Today's processors use a multitude of different optimization techniques to enhance their speed. Examples of such techniques are the instruction cache, the data cache, the instruction pipeline, and the branch predictor. We will shortly explain those techniques and will show in which way HAVEGE uses them to collect entropy.

An important optimization technique is the so-called *cache*. A computer contains memory devices of different speed. Since faster memory is also more expensive, the storage

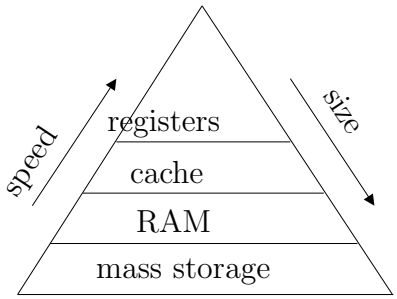

Figure 13.1: Memory devices

volume of the different devices decreases with increasing speed. To take this fact into account, computer designer have designed architecture to connect mass storage device, which are the slowest memory component to the fastest, the registers. Another fact to take into account is Moore's law [Moo65]: The law states that the number of transistors per integrated circuit within a processor doubles every 18 months, which means that the speed is doubling virtually every 2 years. Another version of Moor's law predict that the speed of memory devices is doubling only every 10 year. To reduce this gap, fast memory component like RAM or cache have been designed to increase the locality of data and hide the latency of memory.

Generally, the cache is a fast memory, which temporally stores (*caches*) often used data from a slower memory. As a consequence we not always have to access the slower memory, which highly reduces the access time of the data. For HAVEGE the instruction cache and the data L1 cache are important. Both are located within the processor and are applied to cache the instructions and respectively the data for the processor. To simplify matters we

will only discuss the data cache. Since instructions are virtually nothing else than data, the instruction cache works in the same manner.

If the processor needs specific data, it first checks if those data are already located in the cache. If this is the case, the data can be used directly from the cache. This situation is called a *hit*. If those data are not yet in the cache, they are first loaded from the slower memory into the cache and used by the processor afterwards. This slow case is called a *miss*. We have to consider that the cache is much smaller than slower memory. Thus, the cache is not able to store all data. Nevertheless, the operating system tries to minimize the number of misses.

Another optimization technique is based on the *instruction pipeline*. An instruction can be partitioned into five different steps:

- *IF (Instruction Fetch):* loads the instruction,
- *ID (Instruction Decode):* decodes the instruction and loads the register,
- *EX (EXecution):* executes the instruction,
- *MEM (MEMory access):* accesses slower memory devices, and
- *WB (Write Back):* writes result into register.

Each phase is handled by a different unit of the processor. If a new instruction (I_2) only starts after the previous instruction (I_1) has finished all its phases, then the different units are idle most of the time. The instruction pipeline is used to improve the behavior of the

IF	I_1					I_2	
ID		I_1					I_2
EX			I_1				
MEM				I_1			
WB					I_1		

Table 13.1: Without instruction pipeline

processor. As soon as a unit has finished its work with one instruction, it starts to work on the subsequent one. Since the different units are not idle any more, the number of instructions that can be processed in a given time highly increases. However, problems occur if there exist dependencies between the individual instructions. How should the pipeline behave if a program contains statements of the following form?
`IF` I_1
`THEN` I_2, I_3, ...
`ELSE` I'_2, I'_3, ...
Only after I_1 was finished, the processor knows which instructions will have to be executed

IF	I_1	I_2	I_3	I_4	I_5	I_6	I_7
ID		I_1	I_2	I_3	I_4	I_5	I_6
EX			I_1	I_2	I_3	I_4	I_5
MEM				I_1	I_2	I_3	I_4
WB					I_1	I_2	I_3

Table 13.2: Using the instruction pipeline

next. Basically the processor decides, based on rules, whether to take the main branch (I_2, I_3, ...) or the alternative branch (I'_2, I'_3, ...) and includes the corresponding instructions into the pipeline. If at the end of I_1 it turns out that the wrong branch was chosen, then all calculations after I_1 were useless and the new branch must be loaded into the pipeline. Consequently, much time is lost by choosing the wrong branch. Since a general program contains many `IF/ELSE` statements, the processor needs a mechanism that is as reliable as possible to predict the correct branch. Such a mechanism is called a *branch predictor*.

A simple branch predictor is presented in Figure 13.2. Each `IF/ELSE` decision is

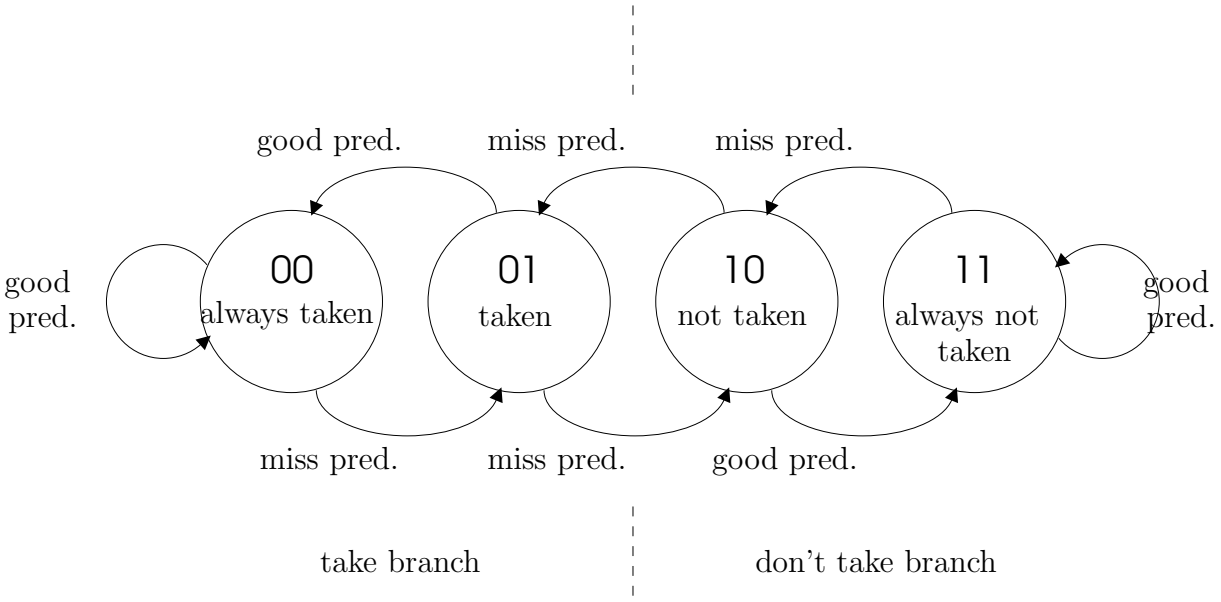

Figure 13.2: Branch predictor

represented by a finite state machine with four states. If the machine is located in one of the left two states, then the main branch is taken, otherwise this branch gets rejected. The transition from one state to another depends on the fact if the prediction of the branch was correct. Even if this machine is quite simple it achieves a good prediction rate on average programs. The state machine can be represented by two bits, thus, the branch predictor contains a small memory and by means of a hash function assigns each `IF/ELSE` decision two bits of this memory.

13.1.2 Functionality of HAVEGE

Now that we know how the different optimization techniques work, the question remains in which way these techniques produce entropy. HAVEGE measures the number of hardware clock cycles, which are required to process a short sequence of instructions. Since these instructions use the optimization techniques described above, the number of required clock cycles highly depends on the current state of those techniques.

In a computer, many processes are executed simultaneously, but the processor is only able to handle one process at a time. Which process has access to the processor at which time is controlled by the operating system. This control is called *scheduling*. While a process allocates the processor, it writes its own data into the cache. After a new process is loaded into the processor, the probability of a miss in the cache or a false prediction of a branch is much higher than if the process would have been working continuously on the processor. This is due to the fact that the data of the previous process may still be located in the processor. Figure 13.3 demonstrates that the behavior of the process P_1 may differ

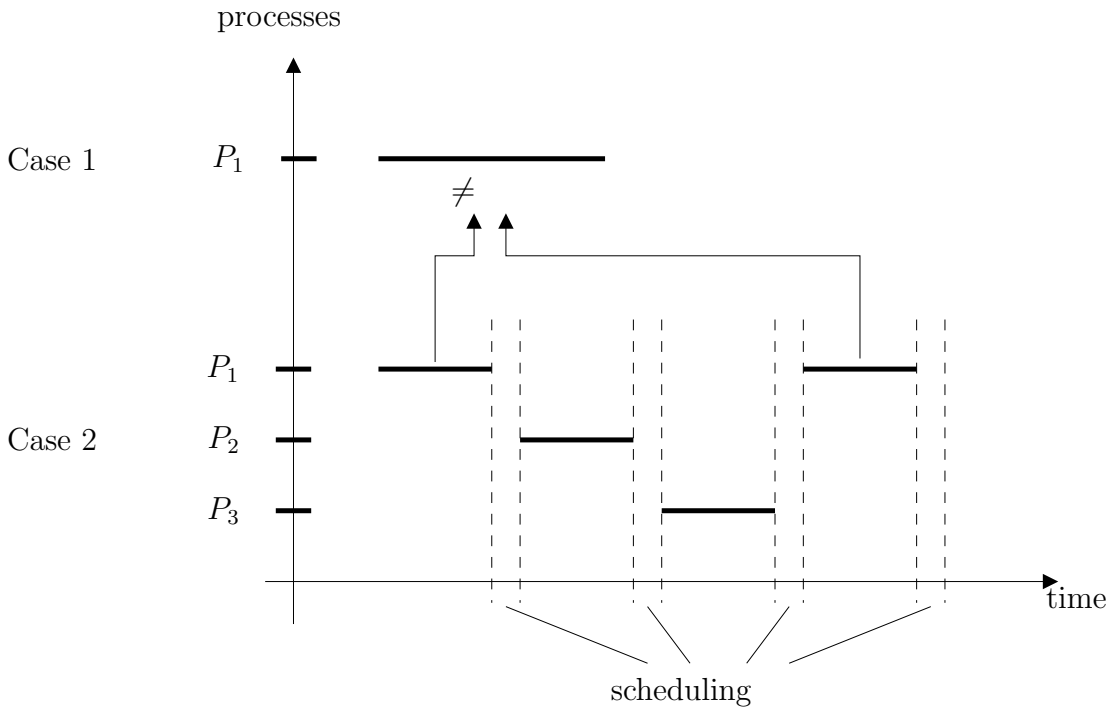

Figure 13.3: Scheduling

between Case 1 and Case 2. The figure describes which process allocates the processor at which time. The time gaps between the processes in the second case arise from the cost of scheduling. In the first case the process was continuously assigned to the processor, in the second case the calculation was interrupted by the processes P_2 and P_3.

Altogether we see that the result of the hardware clock cycle measure of HAVEGE highly depends on the number and type of processes which are handled in the meantime.

Each processed interrupt changes the measured value of clock cycles, because on the one hand the handling of the interrupt uses the processor and on the other hand the interrupt may change the order of the executed processes. Thus, each interrupt injects entropy in the number of required hardware clock cycles. Sendrier and Seznec state in [SS02, p.12] that on average, HAVEG is able to collect at least 8K-16K of unpredictable bits on each operating system interrupt by means of only this gathering technique.

13.1.3 General Structure

HAVEGE contains a table, the so-called `Walk`-Table, and two pointers `PT` and `PT2`. The two pointers represent the current position of two simultaneous walks within the table. The design idea of the table was taken from [BD76]. However, the security assumption of HAVEGE is not based on this article but on the behavior of the processor.

Initially, the table gets filled with random numbers using HAVEG and afterwards is continuously refreshed by the transition function. The transition function measures the number of hardware clock cycles that were required to execute a short sequence of instructions, which depend on the current states of the branch predictor and of the instruction cache.

The course of the two walks is determined by the content of the table. The size of the table was chosen to be twice as big as the data L1 cache. Therefore, the probability is about 50% that the data of the table, which gets accessed by the two walks, is located in the cache . Hence the number of required clock cycles is not only affected by the occurrence of interrupts but also by the content of the table.

The output function just combines the current values of the two walks. Since the content of the table and, thus, the course of the two walks can be assumed to be random, this simple mechanism is sufficient to generate good random numbers.

We include the inner states of the processor into the inner state of the generator, because they have an essential impact on the produced random numbers.

13.2 State Space

The internal state of HAVEGE not only consists of the `Walk`-Table and the two pointers, which represent the two random walks within the table, but also of all the volatile states inside of the processor, which affect the number of required clock cycles. Therefore, the complete state of HAVEGE cannot be observed without freezing the clock of the processor, since each attempt to read the inner state of the processor alters it at the same time. The size of the `Walk`-Table was chosen to be twice as big as the size of the data L1 cache. As an example, the implementation in [SS03] uses a table of 8K 4-byte integers.

Since the volatile states of the processor are part of the inner state of the generator, they have to be considerer when determining the strength of HAVEGE. In [SS03] the authors indicate that there are thousands of such binary states in the processor which are

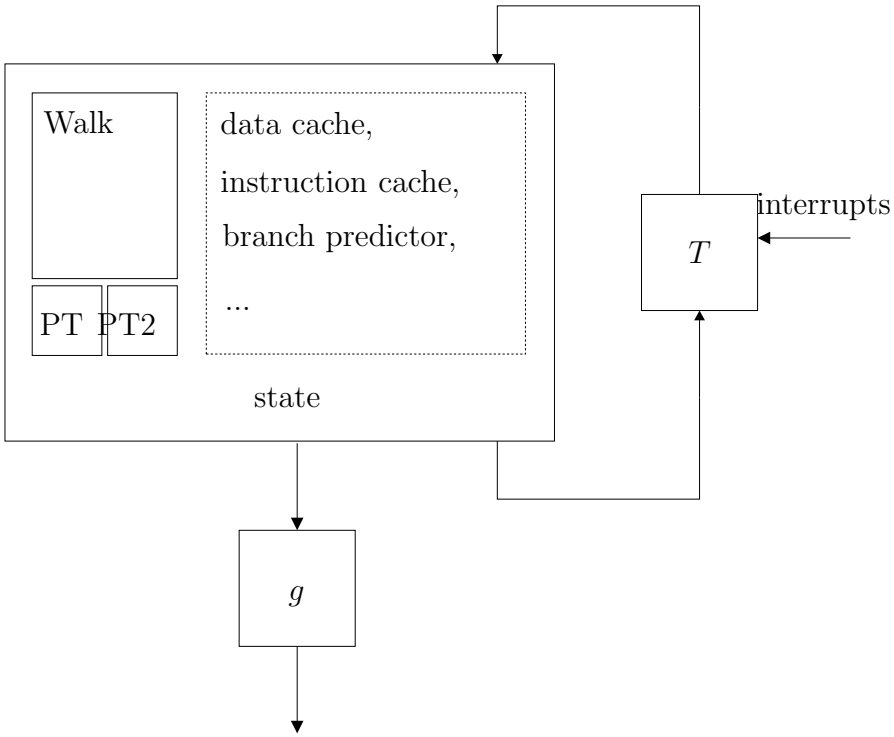

Figure 13.4: General structure of HAVEGE

changed during an interrupt and therefore influence the outcome of the generator. Thus, the strength of HAVEGE is the size of the `Walk`-Table plus thousands of volatile states.

13.2.1 Seed (HAVEG)

Initially the `Walk`-Table is seeded by HAVEG.

HAVEG is the simpler version of the two generators. It just gathers the entropy that was injected by an interrupt into the behavior of the processor. For this purpose, HAVEG uses the method `HARDTICK()`, which measures the number of hardware clock cycles since the last call of this method. The call of `HARDTICK()` is embedded into a small sequence of instructions, which uses a read and a write command as well as a conditional branch. In order to detect as many changes as possible in the instruction cache, the sequence of instructions is repeated as often in the main loop as is necessary to make the loop just fit into the instruction cache. The uncertainties in the behavior of the processor are observable in the result of `HARDTICK()`.

The output of `HARDTICK()` gets included at position K into the `Entrop`-Array by means of a simple mixing function. The mixing function consists of cyclic shift and XOR operations. The shift should compensate the fact that most of the entropy is found in the least significant bits. The XOR operations are used to combine the new input value with the old values at position K and $K+1$ in the array. Thus, the program achieves that each input value has an influence on each position of the array at least after an adequate time.

Since HAVEG gains its entropy from the chaotic behavior after an interrupt, the program counts the number of interrupts occurred. Every time when the result of `HARDTICK()` exceeds a given threshold, which empirically indicates the occurrence of an interrupt, the `INTERRUPT` counter gets increased. `NMININT` represents the minimal number of interrupts that must occur, such that the content of the `Entrop`-Array can be seen as random. As soon as `INTERRUPT` exceeds `NMININT`, the program leaves the main loop and the `Entrop`-Array contains the resulting random numbers.

The disadvantage of this program is that if no interrupts occur, the algorithm gets highly deterministic and the output contains only little entropy.

13.3 Transition Function

The transition function employs two simultaneous walks through a table filled with random numbers to additionally affect the behavior of the processor. The table is twice as big as the data L1 cache. Thus, the probability of a slower access to the table, due to a miss in the cache is about 50%. If one assumes that the `Walk`-Table is filled with random numbers, it is reasonable to claim that the behavior of the processor gets changed in a random way. However, the total absence of interrupts makes HAVEGE a pure deterministic algorithm, but the use of the table helps to compensate interrupt shortages.

A short sequence of instructions contain conditional branches, the access of the two walks on the table, and a call of the `HARDTICK()`-Method. The sequence is repeated in the main loop as often as is necessary to make the program just fit into the whole instruction cache. Thus, the influence of the current value of the branch predictor, the instruction cache, and the data L1 cache gets maximized.

The two simultaneous walks are represented by the two pointers `PT` and `PT2`. The course of the walk corresponding to `PT` is affected by its current position, the content of the table, and the result of `HARDTICK()`. In addition, the new value of `PT` is applied to refresh the content of one cell in the table. The course of the walk corresponding to `PT2` is influenced by its current position, the content of the table, and the value of `PT`. Thus, the result of `HARDTICK()` has a direct and respectively an indirect influence on the behavior of both walks. An adversary which learned the content of the table as well as of the two pointers, loses his or her knowledge about the further course of the walks as soon as one unknown result of `HARDTICK()` gets processed.

13.3.1 Input Space

Every operating system interrupt, whether it is a software or a hardware interrupt, alters the volatile states of the processor and, therefore, the state of HAVEGE. The input space is again infinite, since arbitrary many interrupts may occur during one loop of HAVEGE.

13.4 Output Function

The output function of the generator is very simple. The two pointers, which represent the current position of the two walks within the table, are combined with XOR and put into the output buffer. The XOR should keep adversary from guessing the content of the `Walk`-Table by monitoring the output of the generator.

13.5 Empirical Results

Statistical Tests

The quality of HAVEGE and HAVEG was tested in [SS02] and [SS03] by means of a specific battery of tests. The single tests were performed on 16 Mbyte sequences of random numbers, and each step of the battery was applied to several sequences.

The battery consists of four steps. In the first step the less time consuming *ent* test [Wal98] was used to filter out the worst sequences. In the second and the third step the FIPS-140-2 test suite [Nat01b] and respectively the NIST statistical suit [R$^+$01] were applied. In both cases, a generator passes the tests if the result of the tests corresponds to the average results of the Mersenne twister [MN98] pseudorandom number generator. According to [SS03], the Mersenne twister is known to reliably satisfy uniform distribution properties. In the fourth step the DIEHARD test suit [Mar95] was performed.

HAVEGE consistently passed this test battery on all tested platforms (UltraSparcII, Solaris, Pentium III). This means that the output of HAVEGE is as reliable as pseudorandom number generators like the Mersenne twister, without suffering from the general security problems of PRNGs which arise from a compromised state.

Throughput

HAVEG exhaustively uses the the entropy of an interrupt and thus achieves a much higher throughput than general entropy gathering generators. Sendrier and Seznec showed in [SS02], using an empirical entropy estimation, that HAVEG collects between 8K (Iitanium/Linux) and 64K (Solaris/UtrasparcII) of random bits during a system interrupt. This corresponds to a throughput of a few 100 Kbits per second. For the empirical entropy estimation, they determined for each machine, the necessary number `NMININT` of interrupts, such that the content of the fixed size `Entrop`-Array continuously passes the battery of test. From this value they concluded the number of random bits collected per system interrupt.

Since HAVEGE is less dependent on the occurrence of interrupts it achieves a much higher throughput. The authors state in [SS03] that HAVEGE needs about 920 million cycles on a Pentium II to generate 32 Mbytes of random numbers. This is equal to a throughput of approximately 280 Mbits per second.

13.6 Security

The security of HAVEGE is built on the large unobservable inner state of the generator. Each effort of reading the internal state results in an interrupt and, therefore, alters it at the same time. The only way of reading the total state would be to freeze the hardware clock, which is only possible in special computer labs for research reasons. The high variety of possible internal states and of ways to change them makes it practical impossible to reproduce a sequence of random numbers.

The XOR in the output function prevents that information about the `Walk`-Table may be gained from the output.

In contrast to other generators like Yarrow (see Chapter 10) HAVEGE reseeds its internal state permanently. Therefore, even if an adversary is able to learn the content of the `Walk`-Table and the two pointers, he or she loses track of the walks and, thus, of the future random numbers as soon as the next `HARDTICK()` result is processed. This means that the generator is able to recover very fast from a compromised state.

INRIA still studies different possibilities which may reduce the uncertainty in the behavior of the processor, like for example flooding the processor with user defined interrupts or changing the temperature of the processor. However, since the correlations between the different influences are very complex it is unlikely that an adversary is able to practically affect the result of the generator.

13.7 Portability

HAVEGE is a simple program, which works on user level. It does not use any operating system calls and can therefore be used together with all machines and operating systems that use optimization techniques like branch predictors or caches. To run HAVEGE on a certain computer we simply have to adjust some parameters which correspond to the specific sizes of the branch predictor, the instruction cache and the data L1 cache. Such parameters are for example the number of iterations in the code or the size of the `Walk`-Table.

An implementation and adjustments for several processors can be found at `http://www.irisa.fr/caps/projects/hipsor/HAVEGE.html`.

13.8 Conclusion

HAVEGE is an innovative concept that exhaustively uses the entropy that an interrupt injects into the behavior of the processor. The quality of the output and the throughput of the generator is at least as good as of general pseudorandom number generators. In addition, the security of HAVEGE is hardly reducible. Each attempt to monitor the inner state of the processor alters it and the continuous reseeding of the `Walk`-Table prevents

compromised state attacks.

HAVEG, on its own, is a pure entropy gathering generator. It reaches a higher throughput than other generators of this kind and may be used alone or in combination with other generators.

Factsheet of HAVEGE	
Cryptographic Primitives used	None
Strength	size of table + thousands of volatile states
Statistical Test Results available	NIST test suite, DIEHARD battery
Speed	HAVEG: 8K-16K bits/s HAVEGE: 280 Mbits/s
Portability	Yes

Table 13.3: Summary of HAVEGE

Chapter 14

Summary of Part II

In the previous chapters we introduced five different random number generators. Each of them falls into one of the tree categories we described in Chapter 7 and was chosen for a special reason.

/dev/random is a classical RNG within the Linux kernel and implements an entropy gathering generator. It may be used to extract small amounts of random numbers from an ordinary computer.

Yarrow is a generic concept, which offers suggestions how to combine entropy gathering with a simple PRNG to achieve a higher throughput. Additionally, the designers tried to take several possible attacks on cryptographic RNGs into account.

BBS is a typical example of a PRNG. After it was seeded once it creates random numbers without processing any further input. The special property of this generator is that it is proven to be next-bit unpredictable to the left and to the right. A proof of random qualities is rare in the class of cryptographic RNGs. Most of the time the assumption concerning the quality of the generator are based on empirical results and statistical tests. However, BBS has the disadvantage of low speed and relying on an external random source for reseeding.

AES in Countermode is another version of a PRNG. In this case the quality of the output was not confirmed by a mathematical proof but by statistical tests. Nevertheless, besides resistance against cryptanalytic attacks and nice stochastic properties, AES provides a high throughput. Like all PRNGs it also requires a reliable random source for reseeding.

HAVEGE extends the concept of entropy gathering from an ordinary computer. It does not directly use the occurrence of external events but the uncertainties the corresponding interrupts inject into the behavior of the processor. In addition, the chaotic behavior of the processor is enhanced by means of a table which is continuously filled with random numbers. Thus, HAVEGE achieves a throughput of the same magnitude as fast PRNGs and is still able to provide a continuous reseeding of the inner state and a high strength of the generator.

For cryptographic applications which demand high speed and high unpredictability of

the output we recommend to use HAVEGE. If another reliable but slow random source is available, or if the random sequence should be reconstructible from a shorter seed like in the case of stream ciphers, then we suggest to use AES in connection with an appropriate reseeding mechanism. However, the reader has to decide him/herself which generator fits best for his or her specific needs.

	/dev/random	Yarrow	BBS	HAVEGE	AES
Cryptographic Primitives used	SHA-1 or MD5	SHA-1 3DES	None	None	AES
Strength	1024 bits	160 bits	$\leq \log_2 \phi(N)$	size of table + thousands of volatile states	Counter mode: 256 bits PRNG mode: 128 bits
Statistical Test Results available	DIEHARD battery	None	NIST test suite	NIST test suite, DIEHARD battery	Overlapping serial test, gambling test
Speed	8 - 12K bits/s	No results	3 Kbits/s	HAVEG: 8K-16K bits/s HAVEGE: 280 Mbits/s	426 Mbits/s
Portability	part of Linux Kernel	Yes	Yes	Yes	Yes

Table 14.1: Summary of generators

List of Figures

Bibliography

[BBS86] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal of Computing*, **15**(2):364–383, 1986.

[BD76] C. Bays and S. D. Durham. Improving a poor random number generator. *ACM Transactions on Mathematical Software (TOMS)*, **2**(1):59–64, 1976.

[Ced00] M. Cederholm. Randomizer. `http://www.pierssen.com/arcview/upload/esoterica/randomizer.html`, September 2000.

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, New York, NY, USA, 1991.

[Dev96] L. Devroye. Random variate generation in one line of code. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 265–272, Coronado, CA, USA, 1996. ACM Press.

[DLS97] B. Dole, S. Lodin, and E. Spafford. Misplaced trust: Kerberos 4 session keys. In *Proceedings of the 1997 Symposium on Network and Distributed System Security*, page 60, San Diego, CA, USA, 1997. IEEE Computer Society.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag, Berlin, Germany, 2002.

[EHS02] S. C. Evans, J. E. Hershey, and G. Saulnier. Kolmogorov complexity estimation and analysis. Technical Information Series 2002GRC177, GE Global Research, October 2002.

[Ell95] C. Ellison. *Cryptographic Random Numbers*. IEEE P1363 Appendix E, November 1995. Draft version 1.0, `http://world.std.com/~cme/P1363/ranno.html`.

[Fei58] A. Feinstein. *Foundations of Information Theory*. McGraw-Hill Electrical and Electronic Engineering Series. McGraw-Hill Book Company, New York-Toronto-London, 1958.

[FP85] U. Fincke and M. Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computations*, **44**:463–471, April 1985.

[Gol95] O. Goldreich. Foundation of cryptography - fragments of a book. Both versions are available from `http://theory.lcs.mit.edu/~oded/frag.html`, February 1995. Revised version, Janurary 1998.

[Gol99] O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Algorithms and Combinatorics. Springer-Verlag, Berlin, Germany, 1999.

[Gra99] M. Graffam. /dev/random analysis on linux/x86. `http://www.privacy.nb.ca/cryptography/archives/coderpunks/new/1999-04/0099.h%tml`, April 1999. Posting to the codepunks mailing list.

[Gut98] P. Gutmann. Software generation of practically strong random numbers. In *7th USENIX Security Symposium*, San Antonio, TX, USA, January 1998.

[GW96] I. Goldberg and D. Wagner. Randomness and the netscape browser. *Dr. Dobbs Journal*, January 1996. `http://www.ddj.com/documents/s=965/ddj9601h/`.

[Hel98] P. Hellekalek. Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation*, **46**:485–505, 1998.

[Hes00] D. G. Hesprich. Re: Problem compiling OpenSSL for RSA support. `http://www.mail-archive.com/openssl-dev@openssl.org/msg04496.html`, March 2000. Posting to the openssl-dev mailing list.

[HL00] W. Hörmann and J. Leydold. UNURAN - a library for universal non-uniform random number generators. Available from `ftp://statistik.wu-wien.ac.at/src/unuran/`, 2000. Institut für Statistik, WU Wien, A-1090 Wien, Austria.

[HP02] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, Oxford, UK, third edition, 2002.

[HW03] P. Hellekalek and S. Wegenkittl. Empirical evidence concerning AES. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, **13**(4):322–333, 2003.

[Knu98] D. E. Knuth. *The Art of Computer Programming. Vol. 2: Seminumerical algorithms*. Addison-Wesley, Bonn, Germany, third edition, 1998.

[Kol65] A. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, **1**:1–7, 1965.

[KSF99] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Proceedings of the 6th Annual International Workshop on Selected Areas in Cryptography*, pages 13–33, Kingston, Ontario, Canada, 1999. Springer-Verlag.

[KSWH98] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. *Lecture Notes in Computer Science*, **1372**:168–188, 1998.

[L'E94] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, **53**:77–120, 1994.

[L'E04] P. L'Ecuyer. *Random Number Generation*, chapter 2, pages 35–70. Springer-Verlag, Berlin, Germany, 2004. Handbook of Computational Statistics.

[LLL82] A.K. Lenstra, H.W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, **261**:515–534, 1982.

[LV93] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, Germany, 1993.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, **22**:75–81, 1976.

[Mar95] G. Marsaglia. Diehard battery of tests of randomness. Available from `http://stat.fsu.edu/pub/diehard/`, 1995.

[Mau92] U. Maurer. A universal statistical test for random bit generators. *Journal of Cryptology*, **5**:89–105, 1992.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, **8**(1):3–30, 1998.

[Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics*, **38**(8):114–117, April 1965. Available from `http://www.intel.com/research/silicon/mooreslaw.htm`.

[MvOV01] A. J. Menezes, P. C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press Series on Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, USA, 2001.

[Nat01a] National Institute of Standards and Technology (NIST). *Federal Information Processing Standards Publication (FIPS PUB) 197*, 2001. Available from `http://csrc.nist.gov/publications`.

[Nat01b] National Institute of Standards and Technology (NIST). *Security Requirements for Cryptographic Modules, FIPS-PUB 140-2*, 2001. Available from `http://csrc.nist.gov/publications`.

[Nat02] National Institute of Standards and Technology (NIST). *Federal Information Processing Standards Publication (FIPS PUB) 180-2, Secure Hash Standard (SHS)*, 2002. Available from `http://csrc.nist.gov/publications`.

[Nie92] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods.* CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1992.

[Ö04] F. Österreicher. Skriptum zur Lehrveranstaltung Informationstheorie. Available from `http://www.uni-salzburg.at/mat/staff/oesterreicher/`, 2004. Lecture notes in German.

[POS05] POSIX. IEEE's portable application standards committee (PASC). `http://www.pasc.org/`, March 2005.

[R+01] A. Rukhin et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* NIST Special Publication 800-22, May 2001. Available from `http://csrc.nist.gov/rng/`.

[Riv92] R. Rivest. *RFC 1321 - The MD5 Message-Digest Algorithm.* MIT Laboratory for Computer Science and RSA Data Security, Inc., 1992. Available from `http://csrc.nist.gov/publications`.

[Sch93] B. Schneier. *Applied Cryptography.* Wiley, New York, NY, USA, 1993.

[Sch03] B. Schneier. Yarrow, a secure pseudorandom number generator. `http://www.schneier.com/yarrow.html`, November 2003.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, **27**:623–656, 1948.

[Sot99] J. Soto. Statistical testing of RNGs. Available from `http://csrc.nist.gov/rng/`, April 1999.

[SS02] N. Sendrier and A. Seznec. HArdware Volatile Entropy Gathering and Expansion: generating unpredictable random numbers at user level. Technical report, INRIA, 2002.

[SS03] N. Sendrier and A. Seznec. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Transaction on Modeling and Computer Simulations*, **13**(4):334–346, 2003.

[SW49] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, IL, USA, 1949. Republished in paperback 1963.

[Top74] F. Topsøe. *Informationstheorie*. Teubner Studienbücher. Teubner, Stuttgart, Germany, 1974.

[Ts'99] T. Ts'o. random.c – A strong random number generator. Linux Kernel 2.4.20, `http://www.iglu.org.il/lxr/source/drivers/char/random.c`, September 1999.

[Wal98] J. Walker. ENT a pseudorandom number sequence test program. Available from `http://www.fourmilab.ch/random/`, October 1998.

[Weg98] S. Wegenkittl. *Generalized ϕ-Divergence and Frequency Analysis in Markov Chains*. PhD thesis, University of Salzburg, 1998.

[Weg99] S. Wegenkittl. Monkeys, gambling, and return times: assessing pseudorandomness. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 625–631, New York, NY, USA, December 1999. ACM Press.

[Weg01] S. Wegenkittl. Entropy estimators and serial tests for ergodic chains. *IEEE Transactions on Information Theory*, **47**(6):2480 – 2489, 2001.

[Weg02] S. Wegenkittl. Entropy based tests for randomness and applications to cryptographic generators. `http://random.mat.sbg.ac.at/ftp/pub/data/slides.pdf`, June 2002. Slides.

[Wil93] R. N. Williams. A painless guide to CRC error detection algorithms. `ftp://ftp.rocksoft.com/papers/crc_v3.txt`, August 1993.

Curriculum Vitae

Name:	Andrea Röck
Date of birth:	12th of September, 1979
Place of birth:	Innsbruck, Austria
Citizenship:	Austria

1985–1989 Primary School, Völs, Austria

1989–1993 Bundesrealgymnasium, Innsbruck, Austria

1993–1998 Technical High School for Communications Engineering, Innsbruck, Austria
Diploma: Reifeprüfungszeugnis with excellent success

Since 1998 Applied Computer Science
at the University of Salzburg, Austria

Since 2000 Mathematics at the University of Salzburg, Austria

2001–2002 Bowling Green State University, Ohio, U.S.A
Computer Science
Graduation as Master of Science

October 2004 Internship at Project CODES, INRIA Rocquencourt, Paris, France