ÉCOLE POLYTECHNIQUE

INRIA ROCQUENCOURT

Thèse de Doctorat

présentée à

L'ÉCOLE POLYTECHNIQUE

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité **Informatique**

soutenue le 18 mai 2009 par

Andrea RÖCK

Quantifying Studies of (Pseudo) Random Number Generation for Cryptography.

Jury

Rapporteurs

Thierry Berger	Université de Limoges, France
Andrew Klapper	University of Kentucky, USA

Directeur de Thèse

Nicolas Sendrier	INRIA Paris-Rocquencourt, équipe-projet SECRET, France

Examinateurs

Philippe Flajolet	INRIA Paris-Rocquencourt, équipe-projet ALGORITHMS, France
Peter Hellekalek	Universität Salzburg, Austria
Philippe Jacquet	École Polytechnique, France
Kaisa Nyberg	Helsinki University of Technology, Finland

Acknowledgement

It would not have been possible to finish this PhD thesis without the help of many people.

First, I would like to thank Prof. Peter Hellekalek for introducing me to the very rich topic of cryptography. Next, I express my gratitude towards Nicolas Sendrier for being my supervisor during my PhD at Inria Paris-Rocquencourt. He gave me an interesting topic to start, encouraged me to extend my research to the subject of stream ciphers and supported me in all my decisions. A special thanks also to Cédric Lauradoux who pushed me when it was necessary. I'm very grateful to Anne Canteaut, which had always an open ear for my questions about stream ciphers or any other topic. The valuable discussions with Philippe Flajolet about the analysis of random functions are greatly appreciated. Especially, I want to express my gratitude towards my two reviewers, Thierry Berger and Andrew Klapper, which gave me precious comments and remarks. I'm grateful to Kaisa Nyberg for joining my jury and for receiving me next year in Helsinki. I'm thankful to Philippe Jacquet for assisting in my jury.

Especially, I own a lot to my parents Heinz and Elisabeth Röck and my sister Susanne Röck. They always supported me in my studies, my goings abroad, and any of my decisions. Without them, I would not be here. Also, I do not know what I would have done without the support of Yann. Every time when I was frustrated, desperate or just tired, he cheered me up and encouraged me to go on.

I spend four great years at the project SECRET (formerly known as project CODES). A special thanks to all the people with which I could share the "buró uno", the best office of the world, just beside the coffee table: Cédric, Maria, Mathieu (which we adopt), Matthieu and Yann. I must not forget to mention Christelle. She is the soul of the project, the best secretary, and a very good friend of mine. Thank you. There are many people which passed at Inria and which contribute to the good atmosphere and the discussion at the coffee table. I will mention them in alphabetic order and I hope that I do not forget anyone. I'm grateful to all of them. Avishek Adhikari, Daniel Augot, Raghav Bhaskar, Bhaskar Biswas, Céline Blondeau, Anne Canteaut, Christophe Chabot, Pascale Charpin, Mathieu Cluzeau, Maxime Côte, Frédéric Didier, Cédric Faure, Matthieu Finiasz, Fabien Galand, Benoit Gérard, Christelle Guiziou, Vincent Herbert, Stéphane Jacob, Deepak Kumar Dalai, Yann Laigle-Chapuy, Cédric Lauradoux, Françoise Levy-dit-Vehel, Ayoub Otmani, Raphael Overbeck, Maria Naya Plasencia, Ludovic Perret, Sumanta Sarkar, Nicolas Sendrier, Jean-Pierre Tillich, Marion Videau, and Alexander Zeh.

Once again I want to thank all the people that helped and supported me during my

PhD. It is due to them that I can say now: *Ça, c'est fait.*

Contents

Acknowledgement **i**

1 Introduction **1**

I General Notions **5**

2 Entropy **7**
2.1 Definition and Characteristics 7

3 Stream Cipher **13**
3.1 Classification 13
3.1.1 One-Time Pad (Vernam Cipher) 14
3.1.2 Synchronous Stream Cipher 15
3.1.3 Self-Synchronizing Stream Cipher 16
3.2 State of the Art 17

II Studies of the HAVEGE **Random Number Generator** **19**

4 Random Number Generators **21**
4.1 Tests of Randomness 22

5 HAVEGE **23**
5.1 Optimization Techniques of the Processor 24
5.1.1 Functionality of HAVEGE 27
5.2 General Structure 28
5.2.1 HAVEG 28
5.2.2 HAVEGE 28
5.3 Empirical Results 30
5.4 Security 31
5.5 Conclusion 32

6 Empirical Tests **35**
6.1 Test Setup . . . 35
6.2 Test Results . . . 36
6.2.1 Basic Data . . . 37
6.2.2 Auto Correlation . . . 40
6.2.3 Entropy . . . 42
6.2.4 Entropy of a Markov Chain . . . 43
6.3 Conclusion . . . 48

III Study of a Specific Stream Cipher **51**

7 The DRAGON Stream Cipher **53**
7.1 Introduction . . . 53
7.1.1 Inner State . . . 53
7.1.2 Key and IV Setup . . . 54
7.1.3 State Update . . . 54
7.1.4 Function F . . . 56
7.2 Published Distinguisher on DRAGON . . . 57
7.2.1 Statistical Distinguisher . . . 58
7.2.2 Linear Distinguisher . . . 58
7.3 Some Properties of DRAGON . . . 61
7.3.1 Properties of G's and H's . . . 61
7.3.2 Relations among Words . . . 62
7.3.3 Bias of Different Equations . . . 62
7.3.4 Bias of Equations . . . 64
7.4 Conclusion . . . 67

IV Random Functions **69**

8 Characteristics of Random Functions **71**
8.1 Approach to Analyze Random Functions . . . 72
8.1.1 Use of Generating Function . . . 72
8.1.2 Singularity Analysis . . . 75
8.2 Known Properties . . . 77

9 State Entropy of a Stream Cipher using a Random Function **81**
9.1 Introduction . . . 81
9.2 Estimation of Entropy . . . 83
9.2.1 Previous Work . . . 83
9.2.2 New Entropy Estimation . . . 84
9.3 Collision Attacks . . . 94

9.3.1 States after k Iterations . . . 95
9.3.2 Including Intermediate States . . . 96
9.3.3 Improvement with Distinguished Points . . . 97
9.4 Conclusion . . . 99

V FCSRs 101

10 Introduction to FCSRs 103
10.1 Characteristics of LFSRs . . . 104
10.2 Characteristics of FCSRs . . . 107
10.3 Applications of FCSRs . . . 116
10.4 Extensions of FCSRs . . . 122
10.4.1 d-FCSRs . . . 122
10.4.2 AFSR . . . 124

11 Entropy of the Inner State of an FCSR in Galois Setup 127
11.1 Introduction . . . 127
11.2 Entropy after One Iteration . . . 128
11.3 Final State Entropy . . . 130
11.3.1 Some Technical Terms . . . 131
11.3.2 Final Entropy Case by Case . . . 134
11.3.3 Complexity of the Computation . . . 136
11.4 Lower Bound of the Entropy . . . 137
11.4.1 Basis of Induction . . . 137
11.4.2 Induction Step . . . 138
11.5 Bounds for the Sums . . . 142
11.6 Conclusion . . . 144

12 Parallel generation of ℓ-sequences 145
12.1 Introduction . . . 145
12.2 Motivation . . . 146
12.3 Sub-Sequences Generators and m-Sequences . . . 148
12.4 Sub-Sequences Generators and ℓ-Sequences . . . 150
12.5 Conclusion . . . 158

Conclusion and Perspectives 159

Bibliography 176

Table of figures 178

List of algorithms 179

Chapter 1

Introduction

Random numbers are an important tool in cryptography. They are used to generate secret keys, to encrypt messages or to mask the content of certain protocols by combining the content with a random sequence. A (pseudo) Random Number Generator (RNG) produces "randomly looking" sequences from a short initial seed or from unpredictable events. We mean by randomly looking that an adversary is not able to distinguish the sequence from the outcome of independent and uniformly distributed random variables. Shannon's entropy is one example of an indicator of how hard it is for an adversary to guess a value in the sequence (Chapter 2).

We consider RNGs in two different settings. The first one discusses a random number generator which produces sequences from parameters that we cannot control. Thus, we are not able to produce the same sequence twice. Such generators can be used to generate secret keys. However, we always must be careful whether the output of the generator does contain enough uncertainty/entropy. This was the problem in the random number generator of the Netscape browser used for the Secure Sockets Layer (SSL) protocol. The generator was seeded only from the time of the day, the process ID and the parent-process ID. These values do not contain enough uncertainty, which allows an attack on the protocol [GW96].

The second class of random number generators are completely deterministic algorithms. If we know the initial value, we are able to reconstruct the whole sequence. Thus, they are often referred to as Pseudo RNGs (PRNGs). An area of application of such generators are synchronous stream ciphers (Chapter 3). The initial value depends on the secret key and the public Initialization Vector (IV), and the generated sequence is directly combined with the plaintext.

This document consists of six different parts: In Part I, we explain some general notions which we are going to use later. We give a short introduction to Shannon's entropy in Chapter 2. An overview of stream cipher designs is given in Chapter 3.

The rest of the document is devoted to the research I have done during my PhD. In Part II we consider the HAVEGE random number generator [SS03]. We first give a short introduction to RNGs in Chapter 4. HAVEGE falls into the first category of RNGs, *i.e.* it creates unique sequences depending on unobservable parameters. Modern processors contain several optimization techniques to increase their speed, like data or instruction

caches or branch predictors. The HAVEGE generator uses the uncertainties in those techniques to generates high quality random numbers. The inner state of the processor changes at every interrupt and is not possible to measure without altering it. In Chapter 5, we give a detailed description of the functionality of HAVEGE. We examined the statistical properties of the data which the generator collects directly from the processor, without any post-processing. The results of these studies can be found in Chapter 6.

The following parts consider stream ciphers or components of stream ciphers. In Part III, we examine a specific stream cipher. We chose the Dragon cipher [CHM+04, DHS08], which was a candidate of the European eSTREAM project [eST]. In the first section of Chapter 7, we give a description of the cipher. At the same time as we studied Dragon, two distinguishers on the cipher were published [EM05, CP07]. We present these two works in Chapter 7.2. They cannot be counted as real attacks on Dragon since they use much more keystream bits than are allowed. Finally, in Section 7.3, we give the results of our own studies. We considered different properties of the components of Dragon; however, we were not able to improve the previous mentioned distinguishers.

In Part IV, we study a general model of a stream cipher which uses a random function to update its inner state. We give an overview over random functions and their properties in Chapter 8. The model of the stream cipher itself is presented in Chapter 9. Let us assume that the initial inner state of the stream cipher is uniformly distributed. The use of a random function instead of a random permutation to update the inner state introduces a loss of entropy. We study this loss in detail in Section 9.2. In the last part of this chapter, we consider some state collision attacks on our model. These attacks were inspired by some attacks [HK05] on the MICKEY stream cipher [BD05, BD08] and try to exploit the loss of entropy in the inner state. We could show that in our model, these attacks are not more efficient than a collision search in the initial states. The only way for those attacks to be efficient would be to have a class of functions which lose on average much more entropy than a random function.

In Part V, we consider Feedback with Carry Shift Registers (FCSRs) which can be used as components in stream cipher [ABL08]. FCSRs were introduced by Klapper and Goresky [KG93, KG97] and are a class of fast and non-linear shift registers. In Chapter 10, we give an introduction to FCSRs, including their applications in cryptography and some possible extensions of the original definition. Again, we assume that the initial state of an FCSR is chosen uniformly, using all possible state and carry bits. A Galois FCSR will then lose entropy already after some iterations. In Chapter 11, we consider this loss after one iteration and after it reached a stable value. We could show that the entropy never decreases under the size of the main register. The next chapter, Chapter 12, considers the implementation of FCSRs. We studied the possibilities to parallelize the output of an FCSR. We could show that the method of separated sub-FCSRs is not satisfying. However, partitioning the original FCSR into interconnected sub-registers allows an efficient parallelization even if we have to consider the carry bits.

Finally, in Chapter 12.5, we conclude this work and give some perspectives of possible future research.

Figure 1.1: Statue of Claude E. Shannon, MIT, Cambridge, USA.

My greatest concern was what to call it. I thought of calling it 'information,' but the word was overly used, so I decided to call it 'uncertainty.' When I discussed it with John von Neumann, he had a better idea. Von Neumann told me, 'You should call it entropy, for two reasons. In the first place your uncertainty function has been used in statistical mechanics under that name, so it already has a name. In the second place, and more important, no one really knows what entropy really is, so in a debate you will always have the advantage.' (Claude E. Shannon [TM71, p. 180])

Part I

General Notions

Chapter 2

Entropy

Entropy is an often used term when describing cryptographic primitives. It is a measure for the uncertainty about the output of a data source and was introduced for the first time 1948 by 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]

Entropy belongs to the area of information theory. In this chapter we deal with the definition of entropy and some of its properties.

Shannon's fundamental work on entropy can be found in [SW49]. Further information on this topic is contained in Cover and Thomas's monograph [CT91].

2.1 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 encoded into a symbol by a transmitter and is sent over a defective channel. At the other end of the channel, the received symbol is decoded by the receiver and forwarded to the destination (see Figure 2.1).

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. Hartley [Har28] claimed that if an element is chosen uniformly out of a set Ω_n of size $|\Omega_n| = n$, then the information of this choice is determined by

$$I(\Omega_n) = \log_2(n) \; . \tag{2.1}$$

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 we have the sample space $\Omega_n = \{\omega_1, \omega_2, \ldots, \omega_n\}$ and

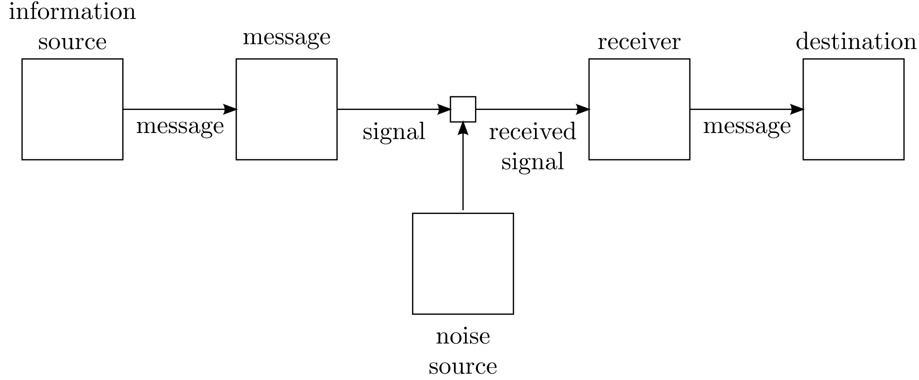


Figure 2.1: Schematic diagram of a general communication system [SW49].

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

$$H(p_1, p_2, \ldots, p_n) = H(p_1, p_2, \ldots, p_{n-2}, p) + p \times H\left(\frac{p_{n-1}}{p}, \frac{p_n}{p}\right). \tag{2.2}$$

We will illustrate this property by an example (see Figure 2.2). 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 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 $\frac{1}{2}$.

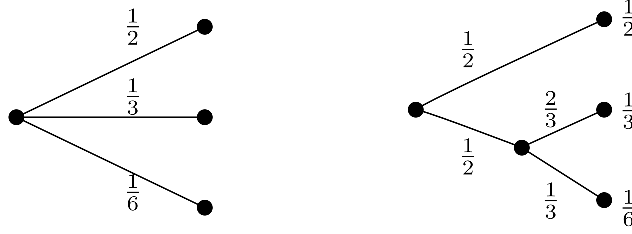


Figure 2.2: Decomposition of a choice from three possibilities [SW49].

We then have the following theorem.

Theorem 2.1 ([SW49]) *Let $P = \{p_i, 1 \leq i \leq n\}$ be the probability distribution on $\Omega_n = \{\omega_1, \omega_2, \ldots, \omega_n\}$. The only function H satisfying the three assumptions (H 1), (H 2), and (H 3) above is of the form:*

$$H = K \sum_{i=1}^{n} p_i \log \frac{1}{p_i} \ , \tag{2.3}$$

where K is a positive constant.

The result of Theorem 2.1 motivates the definition of entropy for discrete random variables.

Definition 2.2 (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*

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

By convention, $0 \log_2 \frac{1}{0} = 0$, which is supported by the asymptotic behavior of the function $f(x) = x \log_2 \frac{1}{x}$, when x tends towards zero,

$$\lim_{x \to 0} x \log_2 \frac{1}{x} = 0 \ .$$

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 (2.3). 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 Euler's 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 6.2.4 we will discuss this topic in greater detail. Here, we focus our discussion on the independent case. For a better understanding of the term entropy, we shall consider two different examples.

Example 2.1 *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* $p_1^{(3)} = \frac{1}{2}$*,* $p_2^{(3)} = \frac{1}{4}$*, and* $p_3^{(3)} = p_4^{(3)} = \frac{1}{8}$*. This implies the following values of entropy.*

$$\begin{aligned} H(X_1) &= 2\,, \\ H(X_2) &= 3\,. \end{aligned}$$

Generally, if X *is* uniformly distributed *on a sample space* Ω*, then* $H(X) = \log_2 |\Omega|$*, which is conform to (2.1). Thus, Hartley's notion of information is a special case of Shannon's notion of entropy. Concerning* X_3*, we have*

$$H(X_3) = \frac{7}{4}\,.$$

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

Theorem 2.3 ([CT91]) *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*.*

Remark 2.4 *This characteristic of the uniform distribution is often used in relation with random number generators (RNGs, see Chapter 4). 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 2.2 (Minimal number of binary questions [CT91]) *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 in the set $\{1, 3, 4, 7\}$?". Such a strategy is equivalent to binary code. Each element $x \in \Omega$ is determined by a finite sequence of responses of yes $= 1$ or no $= 0$. Let EQ denote the expected number of questions in an optimal scheme. Considering the properties of a Huffman code we know that*

$$H(X) \leq EQ \leq H(X) + 1 \; .$$

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

Until now, we have only considered the case of a single random variable. If we have two random variables X, Y we can define the joint entropy $H(X, Y)$ as well as the conditional entropy $H(X|Y)$. The first value gives the entropy of X and Y together, whereas the second one defines the entropy of X if the value of Y is known.

Definition 2.5 (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*

$$H(X, Y) = \sum_{x \in \Omega_X} \sum_{y \in \Omega_Y} p(x, y) \log_2 \frac{1}{p(x, y)} \; .$$

Definition 2.6 (Conditional Entropy) *Let X, Y be two discrete random variable over the state space Ω_X and Ω_Y. We define by $p(x, y)$ the joint probability $X = x$ and $Y = y$ and by $p(x|y)$ the probability of $X = x$ under the condition that $Y = y$. Then the* conditional

entropy $H(X|Y)$ *is defined as*

$$\begin{aligned} H(X|Y) &= \sum_{y\in\Omega_Y} p(y)H(X|Y=y)\ , \\ &= \sum_{y\in\Omega_Y} p(y) \sum_{x\in\Omega_x} p(x|y)\log_2 \frac{1}{p(x|y)}\ , \\ &= \sum_{y\in\Omega_Y}\sum_{x\in\Omega_x} p(x,y)\log_2 \frac{1}{p(x|y)}\ . \end{aligned}$$

The next theorem shows the connection between these two definitions:

Theorem 2.7 (Chain rule [CT91])

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

From this follows that $H(X,Y) = H(X) + H(Y)$ if and only if X and Y are independent, *i.e.* $H(Y|X) = H(Y)$.

Chapter 3

Stream Cipher

Symmetric key encryption schemes are divided in two main classes: block ciphers and stream ciphers. The fist class works on blocks of sizes of ≥ 64 bits. In the basic mode (electronic-codebook mode, ECB) each block is encrypted by the same mechanism only depending on the key. There is no additional memory necessary. In the case of stream ciphers, we encrypt much smaller blocks of size starting from one bit. The encryption function can change from block to block, depending on the current state of the cipher [MvOV96]. However, there are interconnections between these two classes. A block cipher in output feedback (OFB) or counter (CTR) mode can be seen as a stream cipher of relative large block sizes. The cipher feedback (CFB) mode can be interpreted as self-synchronizing stream cipher. In this chapter, we consider only symmetric key stream ciphers; however, in literature we find also examples of public key stream ciphers like the scheme of Blum and Goldwasser [BG84].

Stream ciphers are often less hardware consuming than block ciphers. For example in [GB07], the authors compare the hardware performance of some stream cipher candidates of the eSTREAM project with the AES block cipher. In addition, a stream cipher encrypts each bit/word independently. This is useful in areas where the buffering of data is not possible or where the encryption should be done as soon as an element arrives. This is also useful when the bandwidth is limited and when short messages are transmitted because there is no need of padding. For synchronous stream cipher, there is no error propagation, since the encryption does not depend on previous data. In the case of self-synchronizing stream ciphers, the error propagation is limited.

In the next section, we give a classification of stream ciphers. However, the most important class is surely the synchronous stream ciphers. In Section 3.2, we give a short overview of the state of the art of stream ciphers.

3.1 Classification

There are three main classes of stream ciphers: the one-time pad, the synchronous, and the self-synchronizing stream cipher. The one-time pad has a key of the same size as the

plaintext and allows it to be used only once. The other two classes use one key for multiple encryptions. If the encryption only depends on the key, each encryption produces the same keystream which is then combined with the plaintext. Such a behavior facilitates attacks on the stream cipher. Thus, most of the stream cipher schemes use an additional public Initialization Vector (IV) which changes for each encryption. The initial state and the keystream then depend on the key and the IV.

We denote by K and IV, respectively, the key and the IV of the encryption. The stream of plaintext messages is denoted by the sequence $m_0, m_1, m_2, \ldots$ and the corresponding ciphertext by $c_0, c_1, c_2, \ldots$. If we produce a keystream from the key and the IV, we define this sequence by $z_0, z_1, z_2, \ldots$.

3.1.1 One-Time Pad (Vernam Cipher)

Let the key, the message and the ciphertext be bit sequences of length n, *i.e.* $m_i, c_i, k_i \in \{0,1\}$ for $0 \leq i < n$ and $K = (k_0, k_1, \cdots, k_{n-1})$. For a *Vernam Cipher* we combine the plaintext directly with the key by means of an XOR:

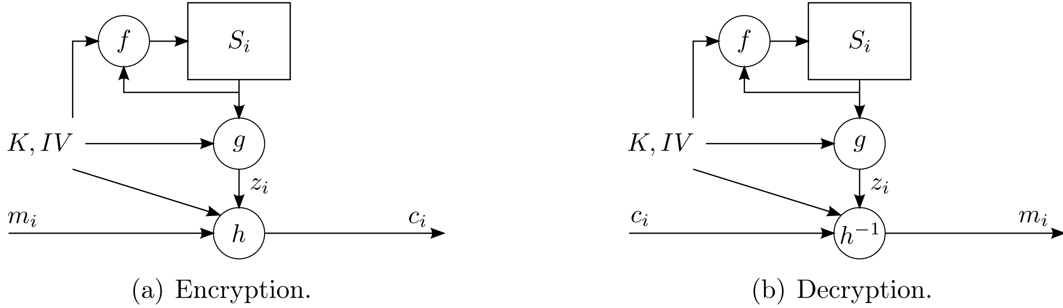
$$\begin{aligned} c_i &= m_i \oplus k_i \quad \text{(encryption)} \\ m_i &= c_i \oplus k_i \quad \text{(decryption)} \end{aligned}$$

If the key was produced by an independent and uniformly distributed random process and is used only once, this scheme is called a *one-time pad*.

Shannon [Sha49] showed that the one-time pad provides perfect security. This means that the conditional entropy of the message M knowing the ciphertext C is the same as the entropy of the original message, *i.e.* $H(M|C) = H(M)$. He also showed that the one-time pad is optimal in the sense that the previous conditions cannot be achieved with a key of size smaller than the message.

The problem of the one-time pad is that we first have to agree on a key of the same length as the message. For most applications this is not practical. The next two schemes try to produce a "random looking" keystream from a short key and IV. By random looking, we mean that we cannot distinguish the keystream from a random sequence in a complexity less than trying all possible keys.

3.1.2 Synchronous Stream Cipher



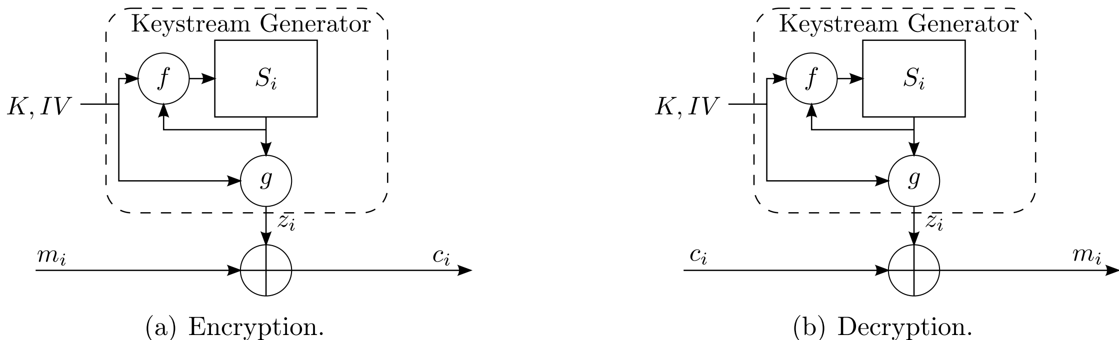
(a) Encryption. (b) Decryption.

Figure 3.1: Model of a synchronous stream cipher.

A synchronous stream cipher (Figure 3.1) consists of an internal state S_i for $i \geq 0$, a state update function f, a filter function g and a combination function h. We start from an initial state S_0 which depends on the key and the IV. For each new message m_i the state is updated by f. The keystream is generated by g from the state and is then combined with the plaintext by mean of the function h. Let h^{-1} denote the inverse function of h for a fixed z_i, *i.e.* $h^{-1}(z_i, h(z_i, m_i)) = m_i$. Then for the decryption, we only exchange the combination function by h^{-1}. In total we get:

$$\begin{aligned} S_i &= f(S_{i-1}) , \\ z_i &= g(S_i) , \\ c_i &= h(z_i, m_i) \text{ (encryption)} , \\ m_i &= h^{-1}(z_i, c_i) \text{ (decryption)} . \end{aligned}$$

The key and the IV can have an influence on all three of the functions.



(a) Encryption. (b) Decryption.

Figure 3.2: Model of a additive stream cipher.

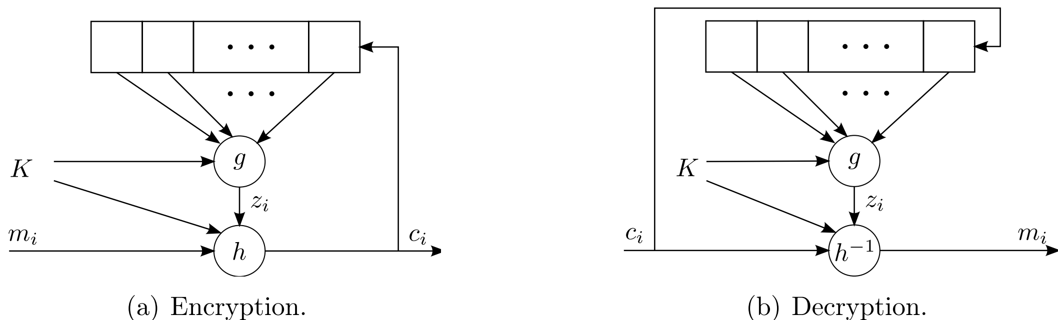
The most common version of this scheme is the additive stream cipher (Figure 3.2). As in the case of the Vernam cipher, it combines the keystream directly by XOR with the plaintext. Thus, the encryption and decryption are identical. The keystream generator works like a pseudo random number generator (PRNG). From a small initial value it produces a pseudo random sequence, the keystream.

The additive stream ciphers are not vulnerable to error propagation, since every small block is encrypted independently. Erasing or inserting such a block causes a loss of synchronization. However, most of the recent schemes include a re-synchronization mechanism.

There exist several generic attacks on stream ciphers, like correlation attacks [Sie84], fast correlation attacks [MS89, CJS00], time/memory/data tradeoff attacks [Bab95, Gol97], or algebraic attacks [CM03]. To prevent these attacks, the cipher has to fulfill several criteria:

- The keystream has to possess a large period length, otherwise it would be easy to distinguish it from a random sequence.
- Let n denote the size of the inner state S. Then n must be at least twice as big as the key size, otherwise a time/memory/data attack is possible [Bab95, Gol97, BS00].
- The filter function must be balanced. Otherwise the keystream is clearly not pseudo random.
- If f and g are linear, then we can establish a system of linear equations depending on the n bits in the inner state S. By using only n keystream bits, we have a system of n equations and n unknown variables, which is easy to solve by Gaussian elimination in only n^3 operations. Thus at least one of the two functions f, g must be nonlinear. If we have $2n$ consecutive output bits, we can apply the Berlekamp-Massey algorithm [Mas69] which has a complexity of only $\mathcal{O}(n^2)$.

3.1.3 Self-Synchronizing Stream Cipher



(a) Encryption. (b) Decryption.

Figure 3.3: Model of a self-synchronizing stream cipher.

A self-synchronizing stream cipher creates its keystream depending on the key and a fixed number of previous ciphertext digits. The idea is that even if an element of the ciphertext gets lost, the scheme is able to re-synchronize after only a short number of iterations. The problem is that these ciphers seem to be very vulnerable to chosen ciphertext attacks [ZWFB04, JDP05, JM06]. In practice, there are replaced by synchronous stream ciphers with an external re-synchronization mechanism.

3.2 State of the Art

In 2000-2003 the NESSIE (New European Schemes for Signatures, Integrity and Encryption) project was launched in search for new cryptographic primitives [NES, Pre05]. There were six stream ciphers submitted to this project; however, all of them fell to cryptanalysis. This showed that at this moment there was not yet enough knowledge about stream ciphers to build a strong scheme. In 2004, the eSTREAM project was initiated [eST, RB08] with the goal of finding a portfolio of solid stream ciphers which improves the AES in stream cipher mode in at least one criteria (throughput, space needed,...). There were 34 initial proposals, divided in two categories:

- **Profile 1**: Stream ciphers for software applications with high throughput.
- **Profile 2**: Stream ciphers for hardware applications with highly restricted resources.

After four years of examination, the final portfolio consists of seven stream ciphers (Table 3.1). The F-FCSR stream cipher was withdrawn from the original portfolio after an attack of Hell and Johansson [HJ08].

Profile 1 (SW)	**Profile 2 (HW)**
HC-128 [Wu08]	Grain v1 [HJMM08]
Rabbit [BVZ08]	MICKEY v2 [BD08]
Salsa20/12 [Ber08]	Trivium [CP08]
SOSEMANUK [BBC+08]	

Table 3.1: Final eSTREAM portfolio.

This project showed that it is possible to find different, robust designs for stream cipher that are fast in software or modest in their hardware requirements. This might respond to the question of Shamir: “Stream Ciphers: Dead or Alive?” [Sha04] that stream ciphers are still an interesting area in cryptography.

Part II

Studies of the HAVEGE Random Number Generator

Chapter 4

Random Number Generators

The generation of random numbers is a crucial topic in cryptography. We differentiate between *random number generators (RNGs)* and *pseudo random number generators (PRNGs)*.

Candidates of the first category generate non-deterministic sequences and are often based on physical reactions, like chaotic oscillators [JF99, BGL+03, EÖ05], or the timing of radioactive decay [Wal06]. These processes are often not very fast and the output is biased, thus, they require additional post-processing procedures [Dic07, Lac08]. The final output allows the creation of strong and unpredictable keys or random seeds.

A PRNG starts from a short seed and generates a long random looking sequence in a deterministic way. These generators are normally much faster than RNGs. They can be used as stream ciphers (see Chapter 3), where the key and the IV are the seed and the generated sequence is combined with the plaintext. PRNGs also find applications in systems where a big number of weaker keys is needed or in mathematical areas like quasi-Monte Carlo methods [Nie92, L'E98].

There is also a big number of hybrid generators which try to collect randomness from events that are deterministic but which depend on parameters that we assume to be unpredictable. HAVEGE falls in this last category of generators. It depends on the inner state of the processor. Since this state is changed by all processes running on a computer, we cannot predict the behavior without knowing exactly which process starts at which time. Also, we cannot measure the behavior without changing it.

For most (pseudo) random number generators, it is hard to give an exact analysis of the output sequences. In general, we want that their output is *computational indistinguishable* from a random process [GM84, Gol90]. If it is not possible to evaluate all the underlying processes, we apply suites of statistical tests. If none of them detects a "non-random", we assume that the generator produces high-quality random numbers. In the next section, we give a short overview of the most important tests for RNGs.

4.1 Tests of Randomness

We want the output of a random number generator to behave like an independent and uniformly distributed random process. The goal of statistical tests is to detect any "non-random" statistical behavior.

Knuth gave in [Knu81] a nice introduction to statistical tests for random number generators. He presented several basic statistical tests, like the *equidistribution test* which counts the frequency of each number, or the *run test* which considers the length of sequences of the same number, *e.g.* the length of "runs" of all 0's.

A more general test is Maurer's universal statistical test [Mau92] which claims to find any defect in a generator which can be modeled as an ergodic stationary source with finite memory. Wegenkittl gives an extension to this approach in [Weg01] by using a test statistic based on the entropy of an ergodic Markov chain.

The National Institute of Standards and Technology (NIST) presents an extensive guide to statistical testing in [RSN+01] as well as a test suite which can be downloaded from their site and contains a wide range of statistical tests.

The ENT pseudorandom sequence test program [Wal98] and the Marsaglia's battery of tests: Diehard [Mar95] are two other examples of available testing software for random number generators.

Further documentation about this topic can be found in the standard FIPS-140-1 [Nat91] and in the earlier versions of FIPS-140-2 [Nat01] as well as on the homepage of the pLab Project [pLa].

The minimal requirement for a random number generator is that it passes the previously mentioned tests. However, this does not guarantee complete security. The test result does not give any information if the seed or the source of entropy is known or can be manipulated.

Chapter 5

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 and is described in [SS02, 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. In our tests, we used the whole implementation of HAVEGE; however, we examined directly the collected data without the final processing step. The main part of this chapter is devoted to HAVEGE, but a description of HAVEG can be found in Section 5.2.1. 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 5.1). 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) than other entropy gathering generators. For example, the Linux RNG /DEV/RANDOM [Ts'99] 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.

This chapter is organized in the following way: In Section 5.1, we explain some of the optimization techniques in a processor and why they can be used by HAVEGE to gather

entropy. The general structure of HAVEGE is explained in Section 5.2. Subsequently, in Chapter 6 we give the results of our tests.

5.1 Optimization Techniques of the Processor

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].

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

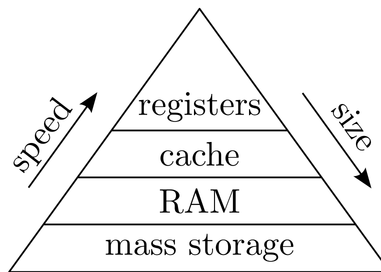


Figure 5.1: Memory devices.

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 Moore'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 do 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 5.1: 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` $I_2, I_3, \ldots$
`ELSE` $I'_2, I'_3, \ldots$

Only after I_1 was finished, the processor knows which instructions will have to be executed next. Basically the processor decides, based on rules, whether to take the main branch (I_2,

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 5.2: Using the instruction pipeline.

$I_3, \ldots$) or the alternative branch ($I'_2, I'_3, \ldots$) 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 5.2. Each `IF/ELSE` decision is repre-

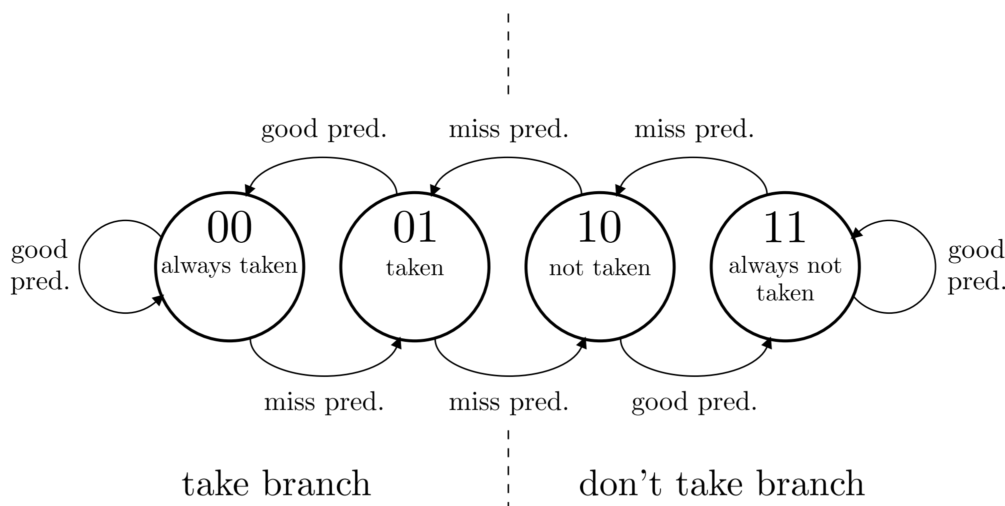


Figure 5.2: Branch predictor.

sented 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 to two bits of this memory.

5.1.1 **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 concurrently, 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 5.3 demonstrates that the behavior of the process P_1 may differ

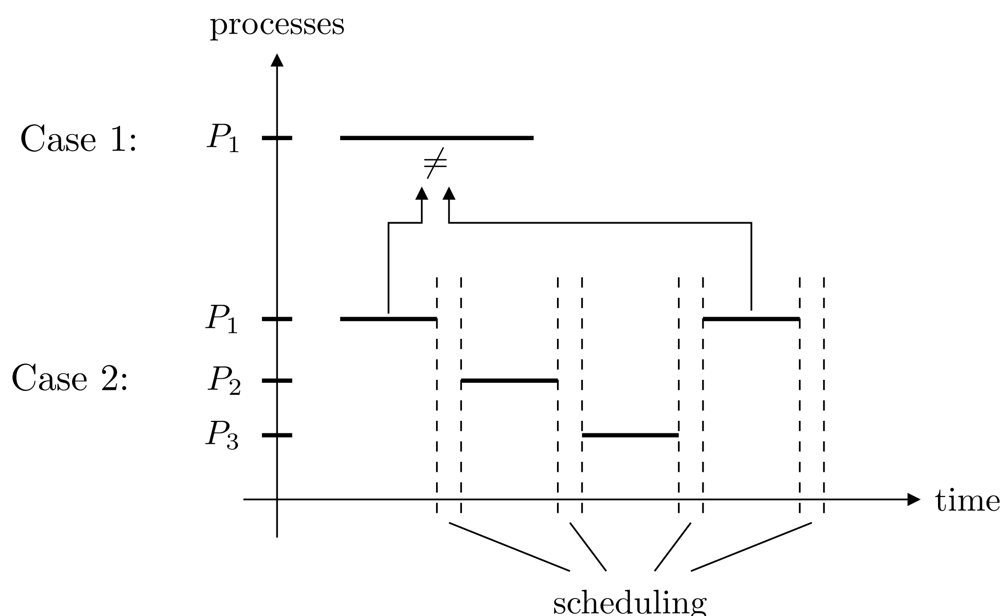


Figure 5.3: Scheduling.

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.

5.2 General Structure

We first explain the entropy gathering in HAVEG and show, subsequently, how HAVEGE extends this method by using two random walks in a big table.

5.2.1 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 returns the current value of the hardware clock counter. The method also tests, whether the number of clock cycles since the last measurements exceeds a given threshold. In this case, it assumes that an interrupt occurred and increases the counter `INTERRUPT`. The call of `HARDTICK()` is embedded into a small sequence of instructions, which uses a read and a write command to be influenced by the state of the cache, 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 positions 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 in `HARDTICK()`, as we have seen before. `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.

5.2.2 HAVEGE

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. This idea was taken from [BD76]. However, the security assumption of HAVEGE is not

based on this article but on the behavior of the processor. The principle structure of HAVEGE can be found in Algorithm 1.

Algorithm 1 Scheme of HAVEGE.

Require: int $Walk[2 * CACHESIZE], PT, PT2$.
Ensure: int $Result[SIZE]$ with $SIZE = 2^{20}$.
1: **for** j from 0 to 31 **do** {Fill $Result$ array 32 times.}
2: **while** $i < SIZE$ **do**
3: {Begin of OneIteration}
4: $PTtest \leftarrow PT >> 20$ {Shift 20 positions to the right.}
5: **if** $PTtest \equiv 1 \pmod 2$ **then**
6: $PTtest \leftarrow PTtest \oplus 3$ {The operator $\oplus$ represents a bit-wise XOR.}
7: $PTtest \leftarrow PTtest >> 1$
8: **if** $PTtest \equiv 1 \pmod 2$ **then**
9: $\cdots$ {Use in total 10 nested if-branches.}
10: **end if**
11: **end if**
12: $HARD \leftarrow HARDTICK()$
13: $Result[i] \leftarrow Result[i] \oplus Walk[PT]$
14: $Result[i+1] \leftarrow Result[i+1] \oplus Walk[PT2]$
15: $Result[i+2] \leftarrow Result[i+2] \oplus Walk[PT \oplus 1]$
16: $Result[i+3] \leftarrow Result[i+3] \oplus Walk[PT2 \oplus 4]$
17: $i \leftarrow i + 4$
18: Replace the values of $Walk[PT], Walk[PT2], Walk[PT \oplus 1], Walk[PT2 \oplus 4]$ with rotated values of the $Walk$ table XORed with $HARD$.
19: $Result[i] \leftarrow Result[i] \oplus Walk[PT \oplus 2]$
20: $Result[i+1] \leftarrow Result[i+1] \oplus Walk[PT2 \oplus 2]$
21: $Result[i+2] \leftarrow Result[i+2] \oplus Walk[PT \oplus 3]$
22: $Result[i+3] \leftarrow Result[i+3] \oplus Walk[PT2 \oplus 6]$
23: $i \leftarrow i + 4$
24: Replace the values of $Walk[PT \oplus 2], Walk[PT2 \oplus 2], Walk[PT \oplus 3], Walk[PT2 \oplus 6]$ with rotated values of the $Walk$ table XORed with $HARD$.
25: Reset $PT2$ depending on values in the $Walk$ table and PT.
26: $\cdots$ {A block similar to line 4-24.}
27: Reset PT depending on values in the $Walk$ table and $PT2$.
28: {End of OneIteration}
29: **Repeat line 3-28 as many times as to fill the instruction cache.**
30: **end while**
31: **end for**

The main work is done in a sequence of instructions which are grouped together in `OneIteration`. In this sequence, we read and write two blocks of eight integer words in the `Walk` table, we read twice the value of `HARDTICK` and we output 16 integer words in

the `Result` array. In addition, there are two big nested if-branches which depend on the current value of the pointer `PT`. The code of `OneIteration` is repeated several times to fill the instruction cache as well as possible.

The course of the random walks of the two pointers `PT` and `PT2` is determined by the content of the table, so the decisions in the if-branches depend on the content of the table. In addition, 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. 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.

The result of `HARDTICK` is each time merged into the content of the `Walk` table by XORing with eight different, rotated entries. The rotation is done to distribute the higher entropy in the last bits. The values which are put into the `Result` array are taken directly from some positions in the `Walk` table, depending on the position of the pointers, and are XORed with the previous values in the array. This simple mechanism is sufficient to generate good random numbers. However, for the initialization of HAVEGE the `Result` table is filled several times by the algorithm described above.

State Space

The size of the internal state is an important criterion for a PRNG. 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 considered when determining the strength of HAVEGE. In [SS03] the authors indicate that there are thousands of such binary states in the processor which are 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.

5.3 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 [Nat01] and respectively the NIST statistical suit [RSN+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 pseudo-random number generators like the Mersenne twister, without suffering from the general security problems of PRNGs which arise from a compromised state.

In Chapter 6, we do some additional analysis. This time we consider the results of `HARDTICK` before they are merged into the `Walk` table. We are interested in the general statistical properties of these sequences and study which part of the processor has the biggest influence on the uncertainties of HAVEGE.

Throughput

HAVEG exhaustively uses 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 (Itanium/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.

5.4 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 practically impossible to reproduce a sequence of random numbers.

The XOR in the output function prevents information about the Walk-Table from being gained from the output.

In contrast to other generators like Yarrow [KSF99, Sch03], 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.

It might be interesting to study 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. In Chapter 6, we examine which influence has the branch predictor and the cache on the entropy of the HARDTICK and thus, on the data produced by HAVEGE.

5.5 Conclusion

HAVEGE is an innovative concept that exhaustively uses the entropy that an interrupt injects into the behavior of the processor. It works on user level and 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/.

The quality of the output and the throughput of the generator is at least as good as for 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 `Walk` table + thousands of volatile states
Statistical Test Results available	NIST test suite, DIEHARD battery
Speed	HAVEG: 8K-16K bits/s HAVEGE: 280 Mbits/s

Table 5.3: Summary of HAVEGE.

Chapter 6

Empirical Tests

In this chapter, we examine the quality of the data which HAVEGE collects from the `HARDTICK` function. For this purpose, we use a degenerated version of the HAVEGE generator. The exact test setup is described in Section 6.1. We are interested in the structure of the data, how much uncertainty we are able to collect, and how much influence the branch predictor and the instruction cache have. In Section 6.2 we give the results of our study and we conclude in Section 6.3

6.1 Test Setup

In the original version of HAVEGE, the output of the generator is taken from the `Walk` table depending on the random walk of the two pointers `PT` and `PT2`. The content of the table as well as the course of the random walks is influenced by the result of `HARDTICK`, *i.e.* the current value of the clock cycle counter. The timing behavior of HAVEGE depends on the state of the branch predictor, the instruction and data cache, and the content of the `Walk` table. The time of the writing and reading processes is essentially influenced by these parameters.

The degenerated version of HAVEGE still writes and reads values from and into the table, to conserve the timing behavior of the generator as much as possible. However, for the statistical tests, we use directly the content of the clock cycle counter as output data of the generator. In `OneIteration` we call twice the `HARDTICK` function and write 16 values in the `Result` array. In the degenerated version of HAVEGE, two of the 16 values will contain the output of `HARDTICK`. The original HAVEGE generator produces 4 Mbytes of random data in one pass. Since we use only every 8th value of the `Result` array we are able to gather 128K integer values with one call of the program. As test data we finally use the differences of the clock cycle counter values. The test data is represented as integer values.

We tested four different versions of the degenerated HAVEGE generator:

- **Normal**: The only difference to the original version is that every 8th value of the `Result` array contains the result of `HARDTICK`. All the other test setups are based on

this version.

- **No Branch Predictor**: In file `OneIteration` we deleted the nested if-branches. Thus, we lose the influence of the branch predictor.
- **One Loop**: In the original version, the content of `OneIteration` is repeated several times to best fit into the instruction cache of the processor. In this setting, the content of `OneIteration` is only repeated once and, therefore, does not take the whole instruction cache.
- **More Loops**: This time, we repeat `OneIteration` more than the optimal number. Thus the code does not fit anymore in the instruction cache.

These four versions were applied in three different situations:

- **Idle**: No special process is running during the random number generation.
- **File Copy (Ordered)**: We copied a big file at the same time as we generated our test data. The copied file consists of the string "abcd" repeated many times, thus, it exhibits an ordered structure.
- **File Copy (Random)**: Again we copy a big file during the generation process; however, this time the file consists of random data (previously generated by HAVEGE).

In the end we tested these 4×3 cases on two different Pentium 4 computers with different cache sizes:

- **PC 1**: Pentium 4, 3.20GHz, 1024 KB cache
- **PC 2**: Pentium 4, 3.40GHz, 2048 KB cache

One application of the degenerated HAVEGE generator creates 128K integer values, which corresponds to the time differences between two calls of `HARDTICK`. For each setting, we created ten different data files of 128K integer values. In addition, we produced files for which we applied successively HAVEGE, respectively, 20, 50, 100, 200, and 500 times.

6.2 Test Results

We are interested in the structure and the quality of the data obtained by `HARDTICK`. In contrast to many statistical tests for random number generators, we do not work on bits and bytes but directly on the resulting integer values. We considered the differences between two successive calls of `HARDTICK`. We studied some basic information of the test data like the mean value, the variance, or their histograms (Section 6.2.1). In Section 6.2.2, we considered the autocorrelation of the received data. Since we wanted to know the amount of uncertainty we are able to collect, we examined which setting gives the highest entropy (Section 6.2.3). In addition, we studied whether the empirical entropy changes if we assume that the test data behaves like the output of a Markov chain (Section 6.2.4).

6.2.1 Basic Data

The resulting data ranges from minimal 136 up to some millions. In general, a clock cycle difference of over 5000 marks the occurrence of an interrupt; however, the majority of values lie under 1000. We compare the average value of different settings and different versions in Table 6.1. The results are taken over ten 128K integer files on PC 1. We can

	Idle	**File Copy (Ordered)**	**File Copy (Random)**
Normal	299.22	324.81	337.73
No Branch Pred.	218.70	245.49	244.57
One Loop	248.05	285.29	249.63
More Loops	312.95	316.29	370.45

Table 6.1: Mean (PC 1).

see that without the influence of the branch predictor or a filled instruction cache, the mean decreases. If we ignore all data values from an interrupt, *i.e.* the values larger than 5000, the mean does not changes a lot. However, the variance reduces from values over a million for the original data, down to some thousands without counting the interrupts.

In a next step, we studied which values are possible. When we considered the small window of the histogram displayed in Figure 6.1 we saw that only a part of all possible numbers appear. In the case of PC 1, all results were multiples of 8. Hence, this was not

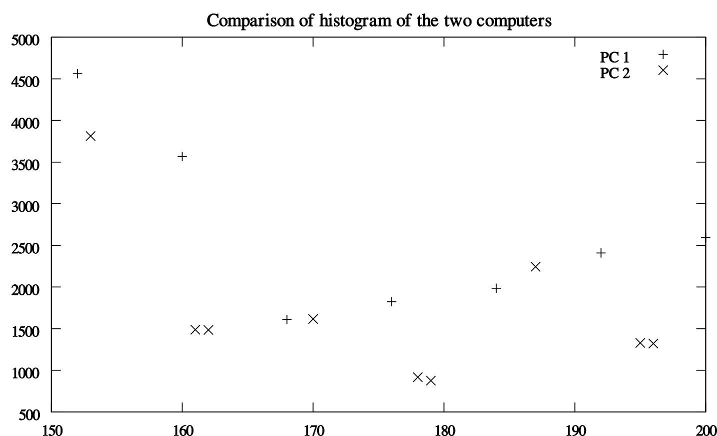


Figure 6.1: Detail of histogram in normal/idle setting.

true for PC 2. If we consider Figures 6.2, 6.3, 6.4, and 6.5 we can see that the histogram changes from version to version. However, the copying of a file has practically no influence on the specific pattern.

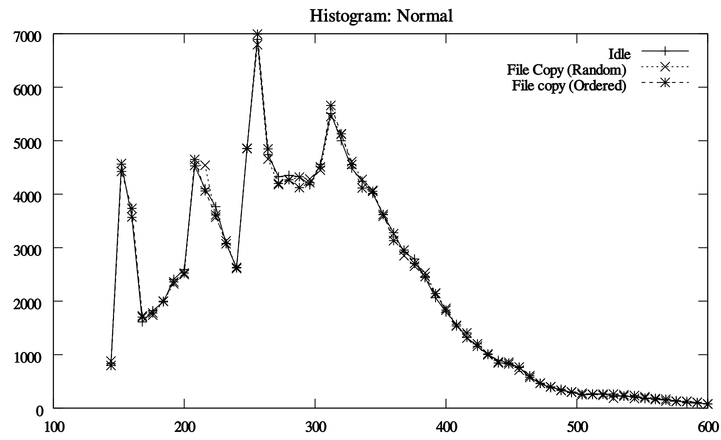


Figure 6.2: Histogram of the normal version.

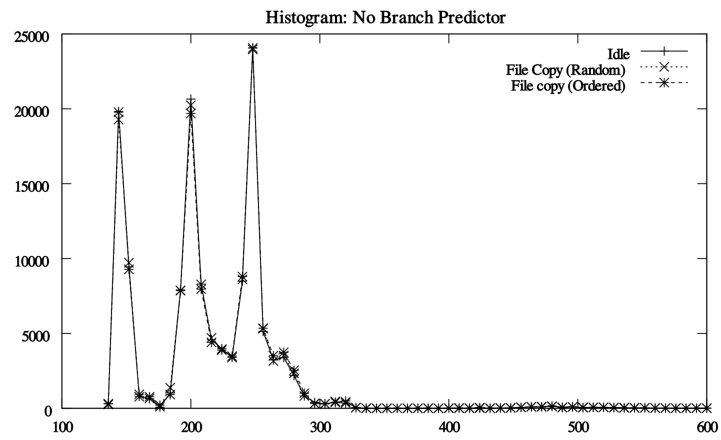


Figure 6.3: Histogram of version: No Branch Predictor.

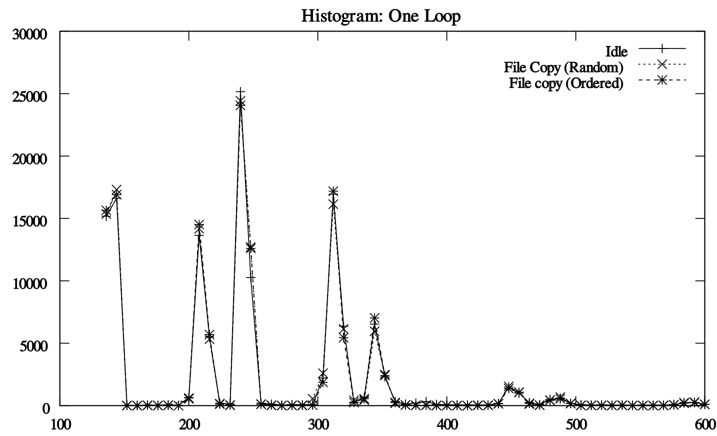


Figure 6.4: Histogram of version: One Loop.

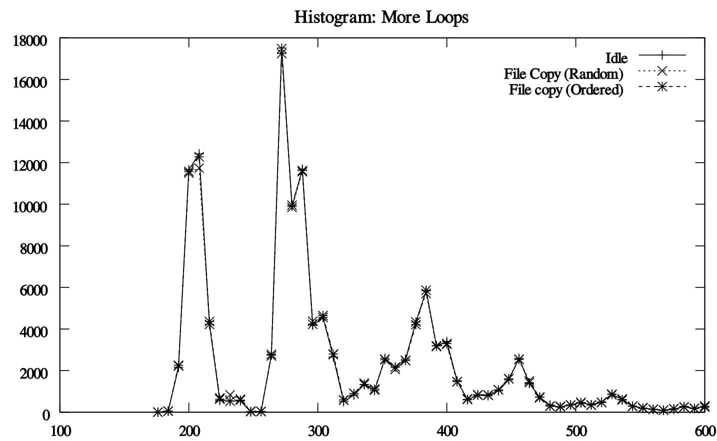


Figure 6.5: Histogram of version: More Loops.

6.2.2 Auto Correlation

In this Section, we consider the autocorrelation of our sequence. It shows whether there is a repeating pattern in the data.

Definition 6.1 *Let us denote by* $(X_t)_{t\geq 0}$ *a random process, where the mean* μ *and the variance* σ^2 *is the same for all variables* X_t*. Then, the* normalized autocorrelation *for a shift* τ *is given by*

$$R(\tau) = \frac{\mathbf{E}\left[(X_t - \mu)(X_{t+\tau} - \mu)\right]}{\sigma^2} , \tag{6.1}$$

where $\mathbf{E}$ *denotes the expectation. In the case of a discrete process* $(X_0, X_1, \ldots, X_{N-1})$ *of length* N*, this can be written as*

$$\hat{R}(\tau) = \frac{1}{N\sigma^2} \sum_{i=0}^{N-1} (X_t - \mu)(X_{t+\tau \bmod N} - \mu) . \tag{6.2}$$

We considered the autocorrelation for a shift up to 200. In Figure 6.6, we see that there is a significant pattern of size two in the correlation graph. This is due to the fact that we read twice the `HARDTICK` in each sequence of `OneIteration`. The absolute amount of the correlation is about constant for shifts up to 200. In Figure 6.7 we consider a

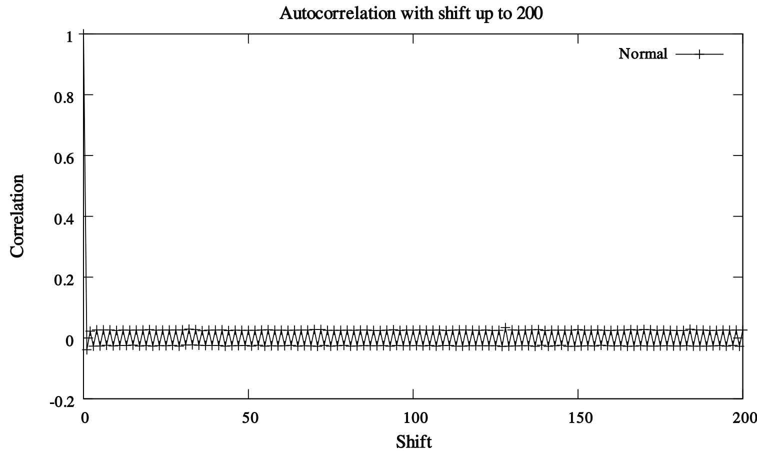


Figure 6.6: Autocorrelation in the normal/idle setting with a shift of up to 200.

smaller window of shifts, to have a better look at the correlation depending on different versions. We see that the autocorrelation is the smallest if we do not benefit from the branch predictor.

In Figures 6.8 and 6.9 we consider the file copy settings. This simultaneous activity reduces considerably the autocorrelation of the sequence. We have to zoom even more into the window to recognize the correlations. Thus, we can conclude that the autocorrelation is most important if the process is idle.

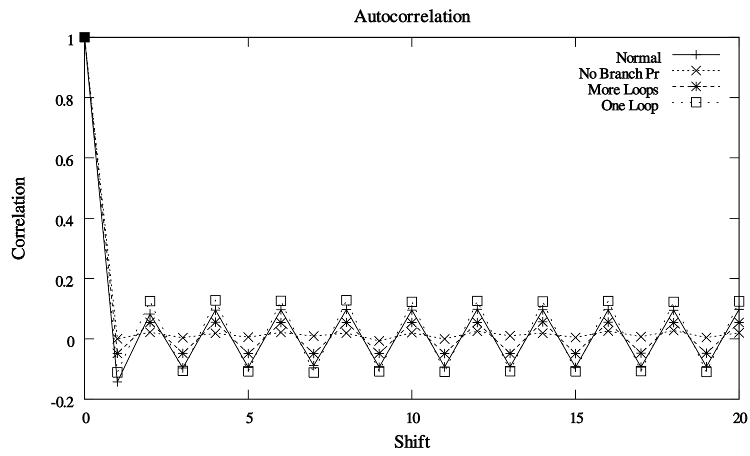


Figure 6.7: Autocorrelation in setting: Idle.

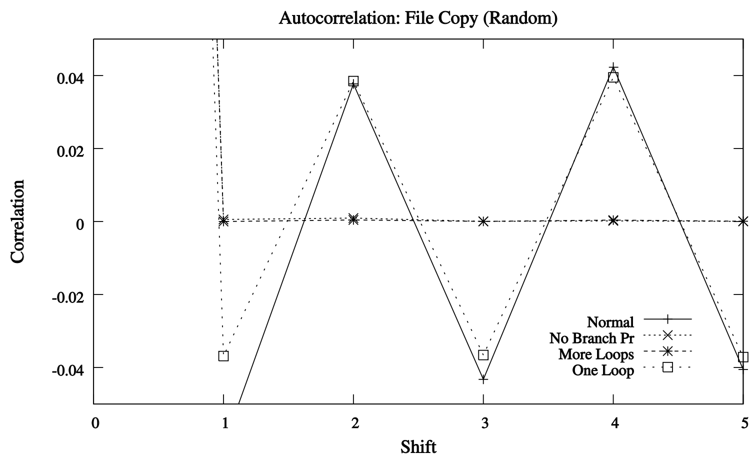


Figure 6.8: Autocorrelation in setting: File Copy (Random).

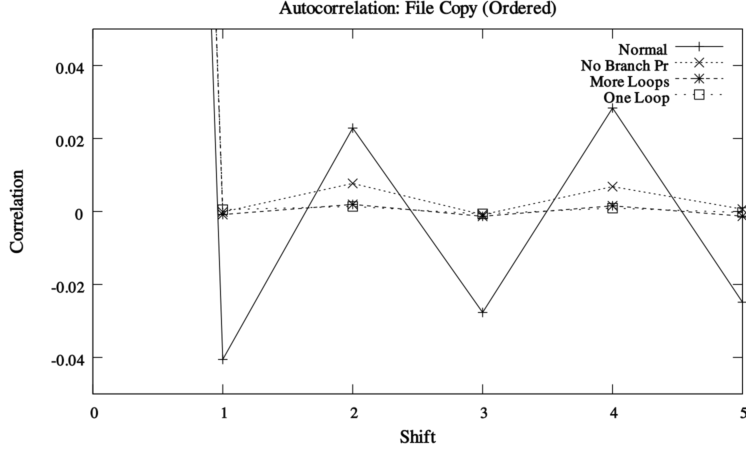


Figure 6.9: Autocorrelation in setting: File Copy (Ordered).

We saw that there is a pattern of size two in the autocorrelation. This indicates that every second value is in a similar range. However, this does not say anything about the exact values. This behavior arises from the fact that the number of commands between the two calls of `HARDTICK` are not exactly the same. Thus, all the results from the first calls are more likely to be in the same range, as well as all results from the second calls. The absolute value reduces as soon as there are other processes running at the same time on the computer.

6.2.3 Entropy

HAVEGE is a random number generator which collects its uncertainty from external events. Thus, if we assume that it behaves like a random source, we would like to know the entropy of this source. In a first step, we assume that we have an independent and identically distributed (i.i.d) sequence $X_1, X_2, \ldots, X_N$ over a finite state space S. For simplicity we assume $S = \{1, 2, \ldots, m\}$. The probability distribution of the X_i's is given by the vector $\mathbf{P} = (p_1, p_2, \ldots, p_m)$, thus, our source has the entropy

$$H(\mathbf{P}) = \sum_{i=1}^{m} p_i \log_2 \frac{1}{p_i} \ .$$

We use the convention that $0 \log \frac{1}{0} = 0$, since a zero probability has no influence on the entropy.

Definition 6.2 *Let $x_1, x_2, \ldots, x_N$ be a sample outcome of our sequence $X_1, X_2, \ldots, X_N$. We denote $\hat{\mathbf{P}} = (\hat{p_1}, \hat{p_2}, \ldots, \hat{p_m})$ the vector of relative frequencies, where $p_i = \frac{\#\{x_j | x_j = i\}}{N}$. Then, the* sample entropy *is given by*

$$H(\hat{\mathbf{P}}) = \sum_{i=1}^{m} \hat{p}_i \log_2 \frac{1}{\hat{p}_i} \ . \tag{6.3}$$

For $N \to \infty$, the sample entropy converges almost sure towards the real entropy of an i.i.d source.

	Idle	File Copy (Ordered)	File Copy (Random)
Normal	5.39	5.39	5.40
No Branch Pred.	3.73	3.75	3.75
One Loop	3.67	3.59	3.85
More Loops	4.71	4.72	4.75

Table 6.2: Sample entropy $H(\hat{\mathbf{P}})$ on PC 1.

	Idle	File Copy (Ordered)	File Copy (Random)
Normal	5.89	5.90	5.90
No Branch Pred.	4.20	4.21	4.20
One Loop	4.31	4.33	4.34
More Loops	5.08	5.09	5.10

Table 6.3: Sample entropy $H(\hat{\mathbf{P}})$ on PC 2.

In Tables 6.2 and 6.3 we computed the sample entropy on two different computers. Each time we took the average entropy over ten different files, even if there was not much fluctuation between the values. We can observe that the entropy is not the same for the two machines. The value for PC 2, which has a larger cache, is always higher. The entropy does not change between the different situations (idle, file copy); however, there are significant differences between the various versions. We achieve the highest value for the normal version. If the version does not depend on the branch predictor or does not fill the instruction cache, the entropy is lowest. When we use more than the optimal number of `OneIteration` blocks we reduce the entropy, hence, it is still higher than in the case of using only one block. With this results, we could confirm that the branch predictor and state of the instruction cache has a big influence on the collected uncertainty. However, the value also changes from machine to machine.

6.2.4 Entropy of a Markov Chain

In the previous section we assumed that HAVEGE produces i.i.d. random numbers. This time we consider a model where the sequence is produced by an ergodic stationary Markov chain (MC). We first give a small introduction to Markov chains and their entropy estimation. In the end, we present the results on our test and discuss their relevance.

Definition 6.3 (Markov Chain of Order κ) *The sequence $(X_n)_{n \in \mathbb{N}}$ is called a* homogeneous Markov chain of order κ *with discrete time and state space $S = \{1, 2, \ldots, m\}$ if*

and only if for every $n \in \mathbb{N}$ the condition

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

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

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

Remark 6.4 *A Markov chain of order $\kappa = 0$ corresponds to an i.i.d. sequence as in Section 6.2.3.*

Markov chains are 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. Let $(X_\ell^{(d)})_{\ell \geq 1}$ define the sequence of *overlapping d-tuples* of the sequence $(X_n)_{n \geq 1}$, where $X_\ell^{(d)} = (X_\ell, \ldots, X_{\ell+d-1})$ for all $\ell \geq 1$. We observe as in [Weg98] that for each MC $(X_n)_{n \geq 1}$ of order κ, the sequence $(\tilde{X}_\ell^{(d)})_{\ell \geq 1}$ of overlapping $d - tuples$ forms a MC of order $\kappa' = 1$ if $\kappa \leq d$. Thus, in the following we limit our discussion to the special case of a MC of order $\kappa = 1$. For more details see [Weg98]. The advantage of a MC of order $\kappa = 1$ is that it can be easily defined by means of a transition matrix.

Definition 6.5 (Homogenous 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 a 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 $(S, \mathbb{P}, \mathbf{P}_0)$ is 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*

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

is satisfied for all $(i_0, \ldots, i_n, j) \in S^{n+2}$, for which $Pr[X_0 = i_0, \ldots, X_n = i_n] > 0$. 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 6.6 (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 6.7 (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. 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 the following definitions.

Definition 6.8 (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* $p_{ij}^{(n)} > 0$.

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

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

Definition 6.9 (Aperiodicity [Weg98]) *An irreducible chain is called* aperiodic *(or acyclic) if the period* p *equals 1.*

The following lemma holds.

Lemma 6.10 ([Weg98]) *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*

$$p_j = \lim_{n \to \infty} p_{ij}^{(n)} \tag{6.4}$$

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

Markov chains that satisfy Condition (6.4), are called *ergodic*. According to this, the behavior of an ergodic Markov chain 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 6.11 (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.

Now that we have all necessary definitions, we can deal with the entropy estimation of Markov chains. To do so, we need some notation.

Remark 6.12 (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* $(x_i, x_{i+1}, \ldots, x_j)$.
- $(\tilde{X}_\ell^{(d)})_{\ell \geq 1}$ *is the sequence of overlapping d-tuples defined in a cyclic way on the sequence $(X_\ell)_{\ell=1}^n$ of length n, such that $\tilde{X}_\ell^{(d)} = X_r^{(d)}$ for all $\ell \geq 1$ if $\ell - 1 \equiv r - 1 \pmod{n}$ and $0 \leq r \leq n-1$.*
- $\hat{P}^{(d)}(n) = (\hat{p}_{\mathbf{i}}^{(d)}(n))_{\mathbf{i} \in S^d}$ *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 can estimate the entropy of the MC using the relative frequency of overlapping d-tuples,

$$\hat{H}^{(d)} = -\sum_{\mathbf{i} \in S^d} \hat{p}_{\mathbf{i}}^{(d)}(n) \log_2 \hat{p}_{\mathbf{i}}^{(d)}(n) + \sum_{\mathbf{i} \in S^{d-1}} \hat{p}_{\mathbf{i}}^{(d-1)}(n) \log_2 \hat{p}_{\mathbf{i}}^{(d-1)}(n). \tag{6.5}$$

The value $\hat{H}^{(d)}$ 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

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

where $\mathbf{E}$ 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 $\epsilon > 0$

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

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

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

A special characteristic of $\hat{H}^{(d)}$ 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 $\hat{H}^{(d)}$ only estimates the d-dimensional marginal distribution, which may differ considerably from the entropy $H(S, \mathbb{P})$ of the chain. The estimator $\hat{H}^{(d)}$ can also be used as a statistical test for RNGs [Weg01]. It then reveals dependencies of order $< d$. In our case, we only use it to estimate the entropy.

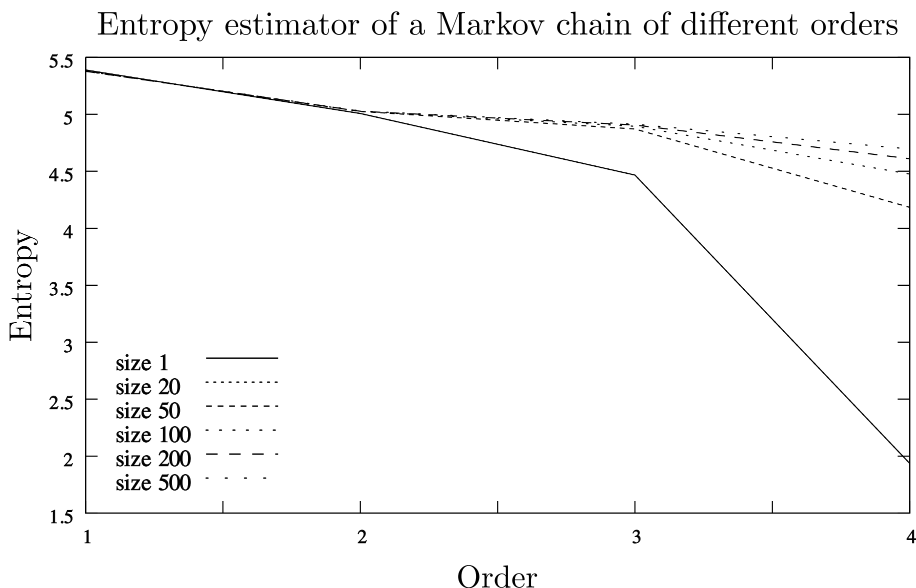


Figure 6.10: Influence of data size for Markov chain entropy estimator.

File Size	Data size	$\frac{Datasize}{\#1-tuples}$	$\frac{Datasize}{\#2-tuples}$	$\frac{Datasize}{\#3-tuples}$	$\frac{Datasize}{\#4-tuples}$
1	131,072	910.22	40.48	3.07	1.1
20	2,621,440	5,472.73	380.19	23.05	2.5
50	6,553,600	9,335.61	645.42	45.11	4.14
100	13,107,200	14,419.36	927.16	74.68	6.27
200	26,214,400	20,242.78	1,284.20	122.22	9.75
500	65,536,000	36,900.90	2,010.80	225.43	17.97

Table 6.4: Number of possible tuples compared with sample size.

We applied the estimator (6.5) on files containing, respectively, 1, 20, 50, 100, 200, and 500 successive calls on the degenerated HAVEGE generator. In Figure 6.10, we compare the result of the estimation for different file sizes and different values of order d. In the case of size 1, the estimated entropy decreases very fast with increasing order. However, this is not due the properties of HAVEGE but due the size of the test data.

The number of possible d-tuples in comparison to the number of available integer values is too small, as we can see in Table 6.4. With increasing d, the number of possible tuples increases exponentially, thus, we can apply the estimator only for small values of d. There is not much difference between the graph for size 200 and 500. Therefore, we assume that the estimation for size 500 is near to the accurate value. The entropy still decreases for increasing values of d. This implies that there are some dependencies in the output data, but it is not possible to model it with a MC of order < 3. Otherwise the estimated value should be constant.

In Figure 6.11, we consider the entropy estimator for files of size 500 and for different versions of the degenerated HAVEGE generator. As in Tables 6.2 and 6.3, the entropy is highest in the normal version and lowest if we do not apply the branch predictor. However, the decrease of entropy for higher values of d is about the same, independent of the applied version.

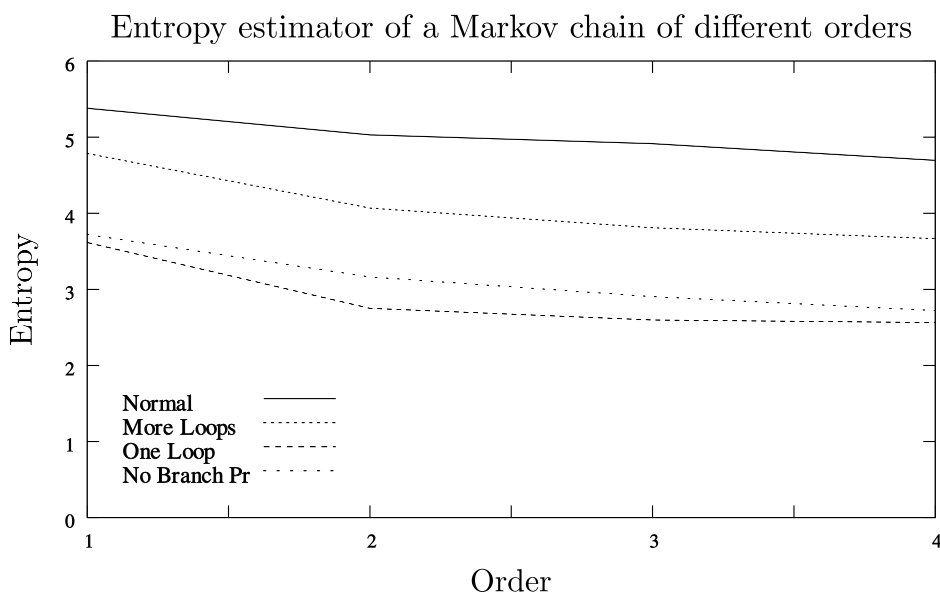


Figure 6.11: Entropy estimation for Markov chains of order $\kappa < 4$.

6.3 Conclusion

HAVEGE is divided in two parts, the collection of uncertainty from the inner state of the processor, and the post-processing of the data by means of a big table and a random walk. The final output passes all common tests of randomness. In this chapter we were interested in the quality of the data collected from the `HARDTICK` function. We considered different versions of a degenerated HAVEGE generator to study which influence has the branch predictor and the cache on the collected data. All tests were done on integer values.

We saw that the histogram depends on the applied version as well as on the used processor. In the autocorrelation, we see a clear pattern of size two. The correlation is biggest for an idle processor. When we considered the sample entropy of the test data, we could confirm that the entropy is highest if we have an optimal dependency on the branch predictor and the instruction cache. In a last step we assumed HAVEGE to behave like a Markov chain. The entropy estimation reduces for increasing order, thus we might assume a dependency structure in HAVEGE. It would be interesting to apply the tests for higher orders to see from which order on the estimation stays constant. However, the large number of possible tuples make the estimation impractical for MC of order ≥ 4.

We studied several properties of HAVEGE. However, we did not find any promising results which would justify deeper research. Thus, we ended our analysis at this point.

Part III

Study of a Specific Stream Cipher

Chapter 7

The DRAGON Stream Cipher

DRAGON [CHM+04, DHS08] is one of the 16 phase 3 candidates of the eSTREAM [eST] project. This project searched for a portfolio of promising new stream ciphers for constrained environments. They had a total of 34 initial submissions.

Together with Maria Naya-Plasencia, I studied several aspects of DRAGON. Unfortunately, we were not able to find a better attack than the distinguisher [CP07] presented by Cho and Pieprzyk. Their distinguisher needs about 2^{150} keystream words generated with the same key, which is much more than the maximal length 2^{64} of keystream allowed per key. Thus, it does not present a real attack on DRAGON.

First, we give in Section 7.1 a description of the stream cipher itself. Subsequently in Section 7.2, we show two published distinguishers on DRAGON. We present some of our own results in Section 7.3. However, until now we did not find any improvement of the attacks in Section 7.2.

7.1 Introduction

DRAGON is a 32-bit word based stream cipher which takes keys and IVs of length 128 or 256 bits (DRAGON-128, DRAGON-256) and outputs 64 bits in every iteration. A key can be used to generate at most 2^{64} bits, which represents 2^{58} iterations of the state update function.

7.1.1 Inner State

The inner state of DRAGON consists of a register of length 1024 bits and a 64 bit memory M. In the case of the key and IV setup, the large register is seen as eight 128-bit words $W_0, \ldots, W_7$. The 64 bits of M are used as additional memory. During the generation of the keystream, the main register is divided into 32 32-bit words $B_0, \ldots B_{31}$ and the M works as a 64 bit counter.

7.1.2 Key and IV Setup

The key and IV setup depends on the size of the key/IV (128 or 256 bits). Let k, iv and K, IV denote the key/IV pair of size, respectively, 128 and 256 bits. We mean by $\overline{x}$ the bitwise complement of x and by x' the value obtained by swapping the upper and lower half of x. The exact key/IV setup working on 128-bit words is presented in Figure 7.1. First, the main register is filled by combinations of the key and the IV. The external memory M is set to the constant value $M_0 = 0x0000447261676F6E$. Finally, we shift the main register 16 times by using the update function F and the memory M. The shift is done on the 128-bit words, *i.e.* $W_i(t+1) = W_{i-1}(t)$ for $1 \leq i \leq 7$. In Section 7.1.4, we give the exact description of F.

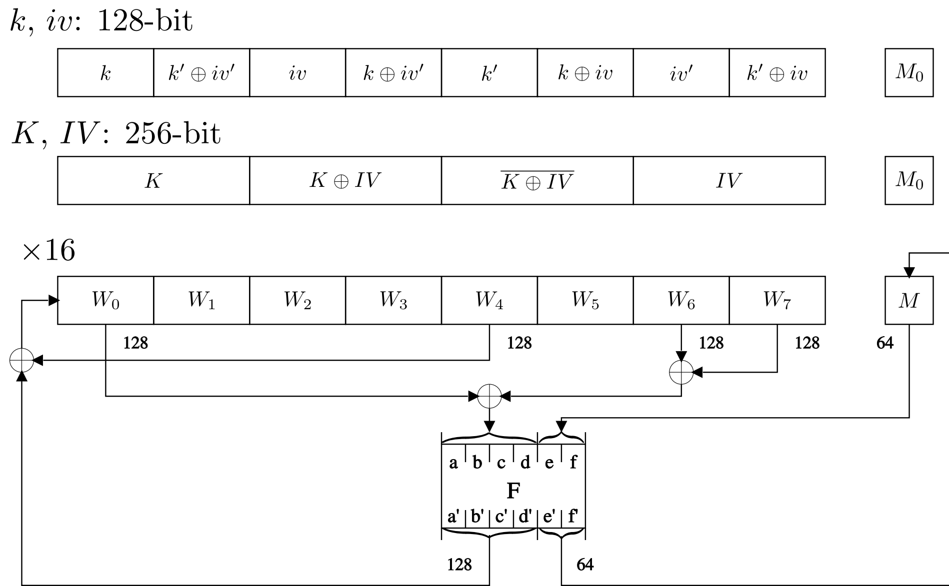


Figure 7.1: Key and IV setup in 128-bit word representation.

We can present the 16 final iterations also by means of 32-bit word operations. In this case, the main register can be partitioned in four subregisters which are interconnected by the function F. In each iteration, the subregisters are shifted by a 32-bit word. This alternative presentation can be found in Figure 7.2.

7.1.3 State Update

After the key/IV setup, DRAGON produces 64 bits of keystream at each iteration of the update function. In Figure 7.3, we describe this procedure in detail. In each iteration of the update function, we increase the value of the 64-bit counter by one. This approach ensures that the keystream has a period of at least 2^{64}. All other computations are done on 32-bit words. The nonlinear function F takes six words from the main register and the two halves M_L, M_R of the memory M. Two words of the output of F are used to generate the

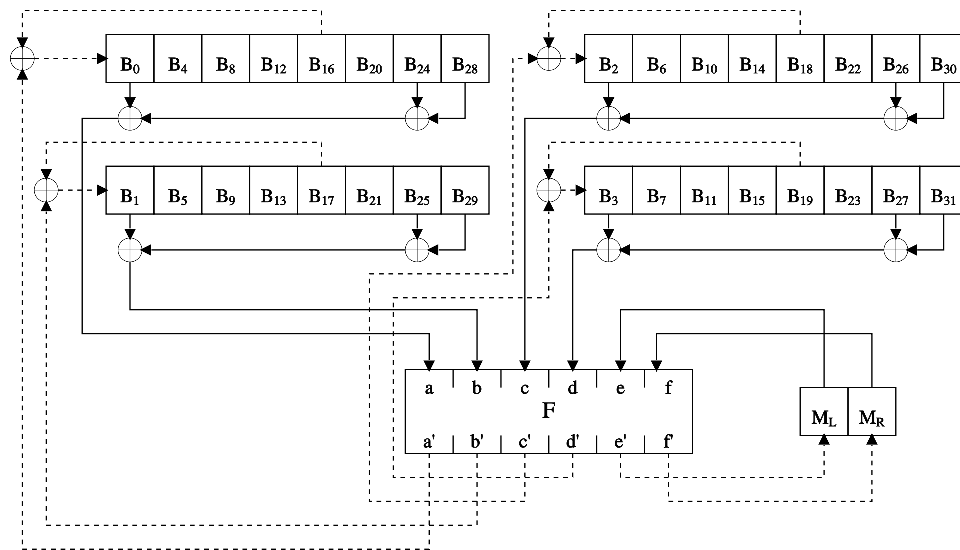


Figure 7.2: Key and IV setup in 32-bit word representation.

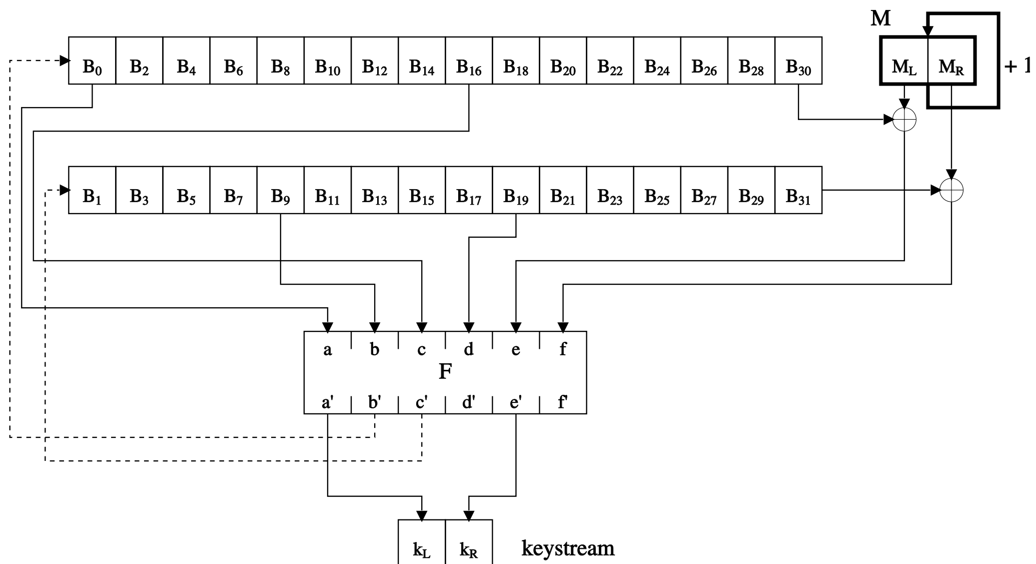


Figure 7.3: Update function.

keystream, two others are injected into the main register, and the final two are discarded. The shift of the main register is done in steps of two 32-words, *i.e.* $B_i(t+1) = B_{i-2}(t)$ for all $2 \leq i \leq 31$. We can partition the register in two subregisters of even and odd indices.

7.1.4 Function F

The function F is a *reversible, nonlinear* mapping of 192 bits to 192 bits. We present the 192-bit input and the output as a concatenation of six 32-bit words $a||b||c||d||e||f$ and $a'||b'||c'||d'||e'||f'$, respectively. Then, F is defined in the following way:

Pre-mixing Layer:

$$\begin{array}{lll} b_1 = a \oplus b & d_1 = c \oplus d & f_1 = e \oplus f \\ c_1 = c + b_1 & e_1 = e + d_1 & a_1 = a + f_1 \end{array}$$

S-box Layer:

$$\begin{array}{lll} d_2 = d_1 \oplus G_1(a_1) & f_2 = f_1 \oplus G_2(c_1) & b_2 = b_1 \oplus G_3(e_1) \\ a_2 = a_1 \oplus H_1(b_2) & c_2 = c_1 \oplus H_2(d_2) & e_2 = e_1 \oplus H_3(f_2) \end{array}$$

Post-mixing Layer:

$$\begin{array}{lll} d' = d_2 + a_2 & f' = f_2 + c_2 & b' = b_2 + e_2 \\ c' = c_2 \oplus b' & e' = e_2 \oplus d' & a' = a_2 \oplus f' \end{array}$$

The addition $+$ is done modulo 2^{32} and the XOR $\oplus$ is done bit per bit. The exact equations are given by:

$$\begin{array}{rcl}
a' &=& \Big((a+(e\oplus f)) \oplus H_1\Big[(a\oplus b)\oplus G_3\Big(e+(c\oplus d)\Big)\Big]\Big) \oplus \\
&& \Big\{\Big[(e\oplus f)\oplus G_2\Big(c+(a\oplus b)\Big)\Big] + \Big((c+(a\oplus b)) \oplus H_2\Big[(c\oplus d)\oplus G_1\Big(a+(e\oplus f)\Big)\Big]\Big)\Big\} \\
b' &=& \Big[(a\oplus b)\oplus G_3\Big(e+(c\oplus d)\Big)\Big] + \Big((e+(c\oplus d)) \oplus H_3\Big[(e\oplus f)\oplus G_2\Big(c+(a\oplus b)\Big)\Big]\Big) \\
c' &=& \Big((c+(a\oplus b)) \oplus H_2\Big[(c\oplus d)\oplus G_1\Big(a+(e\oplus f)\Big)\Big]\Big) \oplus \\
&& \Big\{\Big[(a\oplus b)\oplus G_3\Big(e+(c\oplus d)\Big)\Big] + \Big((e+(c\oplus d)) \oplus H_3\Big[(e\oplus f)\oplus G_2\Big(c+(a\oplus b)\Big)\Big]\Big)\Big\} \\
d' &=& \Big[(c\oplus d)\oplus G_1\Big(a+(e\oplus f)\Big)\Big] + \Big((a+(e\oplus f)) \oplus H_1\Big[(a\oplus b)\oplus G_3\Big(e+(c\oplus d)\Big)\Big]\Big) \\
e' &=& \Big((e+(c\oplus d)) \oplus H_3\Big[(e\oplus f)\oplus G_2\Big(c+(a\oplus b)\Big)\Big]\Big) \oplus \\
&& \Big\{\Big[(c\oplus d)\oplus G_1\Big(a+(e\oplus f)\Big)\Big] + \Big((a+(e\oplus f)) \oplus H_1\Big[(a\oplus b)\oplus G_3\Big(e+(c\oplus d)\Big)\Big]\Big)\Big\} \\
f' &=& \Big[(e\oplus f)\oplus G_2\Big(c+(a\oplus b)\Big)\Big] + \Big((c+(a\oplus b)) \oplus H_2\Big[(c\oplus d)\oplus G_1\Big(a+(e\oplus f)\Big)\Big]\Big)
\end{array} \quad (7.1)$$

S-Boxes

The functions G_1, G_2, and G_3 as well as H_1, H_2, and H_3 are mappings of 32 bits to 32 bits. They are build by combining two S-boxes S_1 and S_2, which each expands 8 bits to 32 bits. Let $x = x_0||x_1||x_2||x_3$ denote the partition of the 32-bit word x into four 8-bit words. Then, the G_i's and H_i's are defined by:

$$\begin{aligned}
G_1(x) &= S_1(x_0) \oplus S_1(x_1) \oplus S_1(x_2) \oplus S_2(x_3) \\
G_2(x) &= S_1(x_0) \oplus S_1(x_1) \oplus S_2(x_2) \oplus S_1(x_3) \\
G_3(x) &= S_1(x_0) \oplus S_2(x_1) \oplus S_1(x_2) \oplus S_1(x_3) \\
\\
H_1(x) &= S_2(x_0) \oplus S_2(x_1) \oplus S_2(x_2) \oplus S_1(x_3) \\
H_2(x) &= S_2(x_0) \oplus S_2(x_1) \oplus S_1(x_2) \oplus S_2(x_3) \\
H_3(x) &= S_2(x_0) \oplus S_1(x_1) \oplus S_2(x_2) \oplus S_2(x_3)
\end{aligned}$$

For the exact definition of the S-box tables of S_1 and S_2 we refer the reader to [CHM+04].

Inverse of F

Even if the functions G_1, G_2, G_3 and H_1, H_2, H_3 are not injective (see Section 7.3.1), the function F is reversible and F^{-1} is defined by:

Post-mixing Layer:

$$\begin{array}{lll}
c_2 = c' \oplus b' & e_2 = e' \oplus d' & a_2 = a' \oplus f' \\
d_2 = d' - a_2 & f_2 = f' - c_2 & b_2 = b' - e_2
\end{array}$$

S-box Layer:

$$\begin{array}{lll}
a_1 = a_2 \oplus H_1(b_2) & c_1 = c_2 \oplus H_2(d_2) & e_1 = e_2 \oplus H_2(f_2) \\
d_1 = d_2 \oplus G_1(a_1) & f_1 = f_2 \oplus G_2(c_1) & b_1 = b_2 \oplus G_3(e_1)
\end{array}$$

Pre-mixing Layer:

$$\begin{array}{lll}
c = c_1 - b_1 & e = e_1 - d_1 & a = a_1 - f_1 \\
b = b_1 \oplus a & d = d_1 \oplus c & f = f_1 \oplus e
\end{array}$$

7.2 Published Distinguisher on Dragon

In the previous section, we saw a short description of the Dragon stream cipher. During the evaluation of the eSTREAM candidates, two distinguisher attacks on Dragon were presented. Since both of them use much more keystream bits than the developer of Dragon allow, neither of these two approaches can be seen as a real attack.

7.2.1 Statistical Distinguisher

Englund and Maximov [EM05] proposed two statistical distinguishers on Dragon which use 2^{255} words of the keystream and runs either with a time complexity of 2^{187} and a memory complexity of 2^{32} or a time complexity of 2^{155} and a memory complexity of 2^{96}.

The distinguisher is based on the assumption that the 32-bit words in the main register are independent and uniformly distributed. This allows the authors to reduce the number of active variables in the equation of a' and e' in (7.1), since the result of an XOR with a uniformly distributed variable is again uniformly distributed. The authors reduce the equations for a' and e' in a way such that we can write $a' = a \oplus N_a$ and $e' = e \oplus N_e$ where N_a and N_e are noise variables. The distribution of N_a and N_e depends only on some uniformly distributed variables and the S-box functions $G_1, G_2, G_3, H_1, H_2,$ and H_3.

In a next step, they observe that due to the shift in the main register:

$$e(t + 15) = a(t) \oplus M_L(t + 15) \ , \tag{7.2}$$

where $M_L(t)$ is the left 32-bit word of the memory M at time t. Let $s(t) = a'(t) \oplus e'(t+15)$ be the sample obtained by combining the output words a', e' at time t and $t + 15$. Then we can write

$$s(t) = a'(t) \oplus e'(t + 15) = N_a(t) \oplus N_e(t + 15) \oplus M_L(t + 15) \ .$$

We do not know the content of $M(t)$. However, we know that it works like a 64-bit counter where $M_L(t)$ and $M_R(t)$ represents, respectively, the most and the least significant 32-bit word. So the attacker guesses the content of $M_R(0)$, this way he will know when M_L changes. We denote these changes by a new variable $\Delta(t)$ which depends on $M_R(0)$. Thus, we can write $M_L(t) = M_L(0) + \Delta(t)$, where the addition is done modulo 2^{32}. Since the value of $M_L(0)$ is the same for all samples, it has no influence on the bias. The guessing of $M_R(0)$ adds a time complexity of 2^{32} to the distinguisher.

To finish the statistical distinguisher, Englund and Maximov need to compute the distribution of the noise variables N_a and N_e from the S-box functions. For a first distinguisher, they compute the distribution of $N_a \pmod{2^n}$ and $N_e \pmod{2^n}$ using the assumption that all cells in the register are independent and uniformly distributed. The reduction modulo 2^n allows the distribution to be computed in a practical time. By using values of n up to 10, the authors found a bias of $2^{-80.57}$. In a next step, they add some noise by approximating the additions modulo 2^{32} by XORs. However, this allows the computation of the noise variable modulo 2^n with $n = 32$, which leads to a bias of $2^{-77.5}$. The authors apply this bias for a distinguisher attack using about 2^{155} key words and having a time complexity of 2^{155+32} and a memory complexity of 2^{32}. The time complexity can be reduced to 2^{155}, if we store the sample distribution of $s(t)$ for each possible $\Delta(t)$ in a table. However, this increases the memory complexity to 2^{96}.

7.2.2 Linear Distinguisher

Cho and Pieprzyk [CP07] obtain their distinguisher by applying a linear approximation to the nonlinear function F. They achieve a bias of $2^{-75.32}$ which leads to an attack which

uses $2^{150.6}$ keystream words and which guesses 59 bits of the internal memory.

Let $x, y \in \mathbb{F}_{2^n}$ be two vectors $x = (x_0, x_1, \ldots, x_{n-1})$, $y = (y_0, y_1, \ldots, y_{n-1})$ of size n. Then the standard inner product $x \cdot y$ is given by $x_0 y_0 \oplus \cdots x_{n-1} y_{n-1}$.

Definition 7.1 *For any function $f : \{0,1\}^m \rightarrow \{0,1\}^n$, any* input mask $\Lambda \in \{0,1\}^m$, *and any* output mask $\Gamma \in \{0,1\}^n$, *the* bias *of the linear approximation $\Lambda \cdot x = \Gamma \cdot f(x)$ is given by*

$$\varepsilon_f(\Lambda, \Gamma) = 2^{-m}(\#\{\Lambda \cdot x \oplus \Gamma \cdot f(x) = 0\} - \#\{\Lambda \cdot x \oplus \Gamma \cdot f(x) = 1\}) \ . \tag{7.3}$$

Then, the probability that the approximation is correct is $Pr[\Lambda \cdot x = \Gamma \cdot f(x)] = \frac{1}{2}(1 + \varepsilon_f(\Lambda, \Gamma))$.

Let + denote the the addition modulo 2^n and $x, y \in \{0,1\}^n$. For any input mask $\Lambda \in \{0,1\}^n$, *and any* output mask $\Gamma \in \{0,1\}^n$, *the* bias *of the linear approximation $\Lambda \cdot (x \oplus y) = \Gamma \cdot (x + y)$ of the addition modulo 2^n is given by*

$$\varepsilon_+(\Lambda, \Gamma) = 2^{-2n}(\#\{\Lambda \cdot (x \oplus y) \oplus \Gamma \cdot (x + y) = 0\} - \#\{\Lambda \cdot (x \oplus y) \oplus \Gamma \cdot (x + y) = 1\}) \ . \tag{7.4}$$

Then, the probability that the approximation is correct is $Pr[\Lambda \cdot (x \oplus y) = \Gamma \cdot (x + y)] = \frac{1}{2}(1 + \varepsilon_+(\Lambda, \Gamma))$.

The bias of the S-box functions can be computed by means of the bias of the S-boxes S_1 and S_2. Let the 32-bit mask Γ be composed by concatenating four 8-bit masks $\Gamma = \Gamma_0 || \Gamma_1 || \Gamma_2 || \Gamma_3$. Then we can directly write the bias for H_1 as

$$\begin{aligned} \varepsilon_{H_1}(0, \Gamma) &= \varepsilon_{S_2}(0, \Gamma) \varepsilon_{S_2}(0, \Gamma) \varepsilon_{S_2}(0, \Gamma) \varepsilon_{S_1}(0, \Gamma) \ , \\ \varepsilon_{H_1}(\Gamma, \Gamma) &= \varepsilon_{S_2}(\Gamma_0, \Gamma) \varepsilon_{S_2}(\Gamma_1, \Gamma) \varepsilon_{S_2}(\Gamma_2, \Gamma) \varepsilon_{S_1}(\Gamma_3, \Gamma) \ . \end{aligned}$$

The same method can be used for the functions G_1, G_2, G_3, H_2, and H_3. The results for $\varepsilon_{G_i}(0, \Gamma)$ and, respectively, $\varepsilon_{H_i}(0, \Gamma)$ are the same for all $1 \leq i \leq 3$, thus, we can just write $\varepsilon_G(0, \Gamma)$ and $\varepsilon_H(0, \Gamma)$. We only have to compute the bias for the two 8-bit to 32-bit S-boxes which is easy for all possible masks.

In the case $\varepsilon_+(\Gamma, \Gamma)$, the authors of the attack applied a theorem that allows the bias to be computed from the bit pattern of Γ.

Theorem 7.2 ([CP07]) *Let n and m be positive integers. Given a linear mask $\Gamma = (\gamma_0, \gamma_1, \ldots, \gamma_{n-1}) \in \{0,1\}^n$ we assume that the Hamming weight of Γ is m. We denote by a vector $W_\Gamma = (w_0, w_1, \ldots, w_{m-1})$, $w_0 < w_1 < \cdots < w_{m-1}$, the bit positions of Γ where $\gamma_i = 1$. Then the bias $\varepsilon_+(\Gamma, \Gamma)$ is determined as follows.*
If m is even, then

$$\varepsilon_+(\Gamma, \Gamma) = 2^{-d_1} \text{ where } d_1 = \sum_{i=0}^{m/2-1} (w_{2i+1} - w_{2i}) \ . \tag{7.5}$$

If m is odd, then

$$\varepsilon_+(\Gamma, \Gamma) = 2^{-d_2} \text{ where } d_2 = \sum_{i=1}^{(m-1)/2} (w_{2i} - w_{2i-1}) + w_0 \ . \tag{7.6}$$

We have seen that for any mask Γ we are able to efficiently compute the biases $\varepsilon_+(\Gamma,\Gamma)$, $\varepsilon_G(0,\Gamma)$, $\varepsilon_H(0,\Gamma)$ and $\varepsilon_{G_i}(\Gamma,\Gamma)$, $\varepsilon_{H_i}(\Gamma,\Gamma)$ for $1 \leq i \leq 3$. In the next step, Cho and Pieprzyk studied which biases we have to consider for the linear approximations $\Gamma \cdot a' = \Gamma \cdot a$ and $\Gamma \cdot e' = \Gamma \cdot e$. In the end, they just look for the mask which total bias is maximal. We will examine the case of $\Gamma \cdot a' = \Gamma \cdot a$ in detail. From (7.1) we know that

$$a' = ((a + (e \oplus f)) \oplus H_1) \oplus \{[(e \oplus f) \oplus G_2] + ((c + (a \oplus b)) \oplus H_2)\} \ .$$

If we apply Γ as input and as output mask and by the linear property of Γ we achieve

$$\Gamma \cdot a' = \Gamma \cdot (a + (e \oplus f)) \oplus \Gamma \cdot H_1 \oplus \Gamma \cdot \{[(e \oplus f) \oplus G_2] + ((c + (a \oplus b)) \oplus H_2)\} \ .$$

Now we approximate one $+$ by $\oplus$ and obtain

$$\Gamma \cdot a' = \Gamma \cdot (a + (e \oplus f)) \oplus \Gamma \cdot H_1 \oplus \Gamma \cdot (e \oplus f) \oplus \Gamma \cdot G_2 \oplus \Gamma \cdot (c + (a \oplus b)) \oplus \Gamma \cdot H_2$$

with a bias of $\varepsilon_+(\Gamma,\Gamma)$. In a next step, we use the approximations $\Gamma \cdot H_1 = 0$ and $\Gamma \cdot H_2 = 0$ to simplify the equation. From the piling-up lemma [Mat93], we know that the bias accumulates. Thus we can write

$$\Gamma \cdot a' = \Gamma \cdot (a + (e \oplus f)) \oplus \Gamma \cdot (e \oplus f) \oplus \Gamma \cdot G_2 \oplus \Gamma \cdot ((c + (a \oplus b)))$$

with a bias of $\varepsilon_+(\Gamma,\Gamma)\varepsilon_H(0,\Gamma)^2$. Now we apply the approximation $\Gamma \cdot G_2([c + (a \oplus b)]) = \Gamma \cdot [c + (a \oplus b)]$ and get

$$\Gamma \cdot a' = \Gamma \cdot (a + (e \oplus f)) \oplus \Gamma \cdot (e \oplus f)$$

with bias $\varepsilon_+(\Gamma,\Gamma)\varepsilon_H(0,\Gamma)^2\varepsilon_{G_2}(\Gamma,\Gamma)$. In a final step, we once again use the approximation for $+$ and obtain

$$\Gamma \cdot a' = \Gamma \cdot a$$

with a bias of $\varepsilon_+(\Gamma,\Gamma)^2\varepsilon_H(0,\Gamma)^2\varepsilon_{G_2}(\Gamma,\Gamma)$. The same method can be applied to obtain the bias for $\Gamma \cdot e' = \Gamma \cdot e$.

By using (7.2) the authors combined the equations of e' and a'. They could compute the mask for which the equation

$$\Gamma \cdot a'(t) = \Gamma \cdot e'(t + 15) \oplus \Gamma \cdot M_L(t + 15)$$

is maximal. For the mask $\Gamma = 0 \times 0600018D$ they obtain a total bias of $2^{-75.8}$. For an attack using this mask and $2^{151.6}$ keystream words, they need to guess 32 bits of $M_R(0)$ and 27 bits from $M_L(0)$.

In all the previous observations, we used the same mask for the input and the output. For their final results of the bias, Cho and Pieprzyk applied the same techniques as Nyberg and Wallén [NW06] in their attack on SNOW 2.0. The idea is to keep the same mask only for the words that are combined by XOR. This means that we keep Γ only for $a'(t)$ and $e'(t + 15)$. For the approximations of the S-box functions within the equations, they allow different masks. This approach allows to increase the estimation of the bias $2^{-75.32}$.

7.3 Some Properties of DRAGON

The weak points in DRAGON seem to be the S-boxes, which expand 8 bits to 32 bits and are used multiple times in the G_i/H_i functions, as well as the partition of the main register into subregisters of even and odd indices. In this section, we present some of the properties of DRAGON which we studied. Unfortunately, none of them leads to a more efficient distinguisher than in [CP07].

7.3.1 Properties of G's and H's

The S-boxes are not reversible since there are many values x, x' which map on the same output. For example for G_1:
Let x_0, x_1, x_2, x_3 be different 8 bits words and $x = x_0||x_0||x_2||x_3$ and $x' = x_1||x_1||x_2||x_3$. It is clear that $x \neq x'$ however $G_1(x) = G_1(x')$.

Lemma 7.3 *For any function H_i and G_i and $1 \leq i \leq 3$ the size of the images is*

$$\leq 253\ 2^{24} + 3\ 2^{16} \approx 0.988 \times 2^{32} .$$

Proof.
We show this property for the example of H_1. The same approach applies for all other functions H_i, G_i with $1 \leq i \leq 3$. Let $x = x_0||x_1||x_2||x_3$ and let x_i,y_i be 8 bit words for $0 \leq i \leq 3$. Then

$$H_1(x) = S_1(x_0) \oplus S_1(x_1) \oplus S_1(x_2) \oplus S_2(x_3) .$$

For given y_0, y_1, we count the number of possibilities for x's such that $H_1(x) = S_1(y_0) \oplus S_2(y_1)$. We set $x_3 = y_1$. For the values x_0, x_1, x_2 we have to count two different cases.

1. $x_0 = x_1 = x_2 = y_0$: It is easy to see that we obtain $H_1(x) = S_1(y_0) \oplus S_2(y_1)$.

2. One of the values x_0,x_1,x_2 are y_0, the remaining two values are identical and y_2. In this case it holds that $H_1(x) = S_1(y_0) \oplus S_2(y_1)$.
   There are 3 possibilities to choose the index $j \in \{0, 1, 2\}$ such that $x_j = y_0$ and $2^8 - 1$ possibilities for y_2. So in total, this case appears $3(2^8 - 1)$ times for each pair y_0,y_1.

For each pair y_0,y_1 there are $3(2^8 - 1) + 1 = (3\ 2^8 - 2)$ possible x's which produce the same value $H_1(x) = S_1(y_0) \oplus S_2(y_1)$. The rest of the time we assume that we do not have any other collisions. Thus, the image has a size of

$$\leq 2^{32} - 2^{16}(3\ 2^8 - 3) = 253\ 2^{24} + 3\ 2^{16} .$$

◇

We showed that the S-box functions G_i and H_i are not bijective and that at least $2^{16}(3\ 2^8 - 3)$ values are never reached. However, the output of those functions is always combined with at least another 32-bit word. Thus, if this additional world takes all values in $\{0, 1\}^{32}$, the combination will take them as well. We did not find any practical weakness which arises from the reduced image size.

Remark 7.4 *The S-boxes and thus the functions G_i and H_i are biased. We give an example of the resulting biases in Table 7.4.*

7.3.2 Relations among Words

Let $M_L(t)$ and $M_R(t)$ denote the 32-bit words corresponding to the left and right half of the memory M at time t. In addition, we mean by $a(t), b(t), c(t), d(t), e(t), f(t)$ and $a'(t), b'(t), c'(t), d'(t), e'(t), f'(t)$ the input and output values of the function F at time t. Since b' and c' are reinserted into the main register, we can write the relation between these words for several time indices (Table 7.1).

$\mathbf{a(t)}$	$c(t+8)$	$e(t+15) \oplus M_L(t+15)$	$b'(t-1)$
$\mathbf{b(t)}$	$d(t+5)$	$f(t+11) \oplus M_R(t+11)$	$c'(t-5)$
$\mathbf{c(t)}$	$a(t-8)$	$e(t+7) \oplus M_L(t+7)$	$b'(t-9)$
$\mathbf{d(t)}$	$b(t-5)$	$f(t+6) \oplus M_R(t+6)$	$c'(t-10)$
$\mathbf{e(t)}$	$a(t-15) \oplus M_L(t)$	$c(t-7) \oplus M_L(t)$	$b'(t-16) \oplus M_L(t)$
$\mathbf{f(t)}$	$b(t-11) \oplus M_R(t)$	$d(t-6) \oplus M_R(t)$	$c'(t-16) \oplus M_R(t)$
$\mathbf{b'(t)}$	$a(t+1)$	$c(t+9)$	$e(t+16) \oplus M_L(t+16)$
$\mathbf{c'(t)}$	$b(t+5)$	$d(t+10)$	$f(t+16) \oplus M_R(t+16)$

Table 7.1: Relation among words.

7.3.3 Bias of Different Equations

We would like to simplify the representations of $a'(t), b'(t), c'(t), d'(t), e'(t), f'(t)$ in (7.1), page 56, by applying linear approximations as in Section 7.2.2. Let $\Gamma \in \{0,1\}$ be a 32-bit mask. Cho and Pieprzyk considered in their distinguisher attack only the equations $\Gamma \cdot a' = \Gamma \cdot a$ and $\Gamma \cdot e' = \Gamma \cdot e$. We wanted to know which bias we have to consider for a bigger number of equations.

In Table 7.2, we show which biases we have to count for the linear approximation in the second column. For example for G_1, x means that we use the approximation $\Gamma \cdot G_1(x) = \Gamma \cdot x$ and, thus, we need to count the bias $\varepsilon_{G_1}(\Gamma, \Gamma)$. Whereas, 0 means that apply $\Gamma \cdot G_1(x) = 0$ and that we have to consider $\varepsilon_{G_1}(0, \Gamma)$. The column + counts the number of cases where we approximate + by $\oplus$, *i.e.* $\Gamma \cdot (x + y) = \Gamma \cdot (x \oplus y)$. Thus, we have to use the bias of $\varepsilon_+(\Gamma, \Gamma)$. The values $\mathcal{A}_1$, $\mathcal{A}_2$,... are necessary for the next table.

The problem is that the values of a',b',c',d',e' and f' are built from the same elements in the S-box layer (see Section 7.1.4). Thus, if we have for example G_1 in the equation of $d'(t)$, we have the same value of G_1 in the equation of e'. Therefore we are not allowed to use the approximations $\Gamma \cdot G_1(x) = \Gamma \cdot x$ and $\Gamma \cdot G_1 = 0$, both at time t.

We will show this on a small example: For an arbitrary function $g(x)$ we define $z_1 = g(x) \oplus y$ and $z_2 = g(x)$. Then for any mask Γ we have $\Gamma \cdot (z_1 \oplus z_2) = \Gamma \cdot y$ without any bias. If we first use the approximation $\Gamma \cdot g(x) = 0$ on z_1 and $\Gamma \cdot g(x) = \Gamma \cdot x$ on z_2 we

		G_1	G_2	G_3	H_1	H_2	H_3	$+$
$\mathcal{A}_1$	$\Gamma \cdot a' = \Gamma \cdot a$		x		0	0		2
$\mathcal{A}_2$	$\Gamma \cdot a' = \Gamma \cdot (b \oplus c)$		0		0	0		3
$\mathcal{A}_3$	$\Gamma \cdot a' = \Gamma \cdot (a \oplus c)$		0	0	x	0		3
$\mathcal{A}_4$	$\Gamma \cdot a' = \Gamma \cdot (b \oplus d)$	0	0		0	x		3
$\mathcal{E}_1$	$\Gamma \cdot e' = \Gamma \cdot e$	x			0		0	2
$\mathcal{E}_2$	$\Gamma \cdot e' = \Gamma \cdot (a \oplus f)$	0			0		0	3
$\mathcal{E}_3$	$\Gamma \cdot e' = \Gamma \cdot (a \oplus e)$	0	0		0		x	3
$\mathcal{E}_4$	$\Gamma \cdot e' = \Gamma \cdot (b \oplus f)$	0		0	x		0	3
$\mathcal{C}_1$	$\Gamma \cdot c' = \Gamma \cdot c$			x		0	0	2
$\mathcal{C}_2$	$\Gamma \cdot c' = \Gamma \cdot (d \oplus e)$			0		0	0	3
$\mathcal{C}_3$	$\Gamma \cdot c' = \Gamma \cdot (c \oplus e)$	0		0		x	0	3
$\mathcal{C}_4$	$\Gamma \cdot c' = \Gamma \cdot (d \oplus f)$		0	0		0	x	3
$\mathcal{B}_1$	$\Gamma \cdot b' = \Gamma \cdot (a \oplus b)$			x			0	1
$\mathcal{B}_2$	$\Gamma \cdot b' = \Gamma \cdot (a \oplus b \oplus c \oplus d \oplus e)$			0			0	2
$\mathcal{B}_3$	$\Gamma \cdot b' = \Gamma \cdot (a \oplus b \oplus c \oplus d \oplus f)$		0	0			x	2
$\mathcal{D}_1$	$\Gamma \cdot d' = \Gamma \cdot (c \oplus d)$	x			0			1
$\mathcal{D}_2$	$\Gamma \cdot d' = \Gamma \cdot (a \oplus c \oplus d \oplus e \oplus f)$	0			0			2
$\mathcal{D}_3$	$\Gamma \cdot d' = \Gamma \cdot (b \oplus c \oplus d \oplus e \oplus f)$	0		0	x			2
$\mathcal{F}_1$	$\Gamma \cdot f' = \Gamma \cdot (e \oplus f)$		x			0		1
$\mathcal{F}_2$	$\Gamma \cdot f' = \Gamma \cdot (a \oplus b \oplus c \oplus e \oplus f)$		0			0		2
$\mathcal{F}_3$	$\Gamma \cdot f' = \Gamma \cdot (a \oplus b \oplus d \oplus e \oplus f)$	0	0			x		2

Table 7.2: Bias of equations.

	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$\mathcal{A}_4$	$\mathcal{E}_1$	$\mathcal{E}_2$	$\mathcal{E}_3$	$\mathcal{E}_4$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$	$\mathcal{C}_4$	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_3$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$
$\mathcal{A}_1$					•	•			•	•			•	•		•	•		•		
$\mathcal{A}_2$					•	•	•		•	•		•	•	•	•	•	•			•	
$\mathcal{A}_3$								•		•		•		•	•			•		•	
$\mathcal{A}_4$						•	•				•		•	•	•		•				•
$\mathcal{E}_1$	•	•							•	•			•	•		•			•	•	
$\mathcal{E}_2$	•	•		•					•	•	•		•	•			•		•	•	•
$\mathcal{E}_3$		•		•								•			•		•			•	•
$\mathcal{E}_4$			•							•	•			•				•	•	•	•
$\mathcal{C}_1$	•	•			•	•							•			•	•		•	•	
$\mathcal{C}_2$	•	•	•		•	•		•						•		•	•	•	•	•	
$\mathcal{C}_3$				•		•		•						•			•	•			•
$\mathcal{C}_4$		•	•				•								•	•	•	•		•	
$\mathcal{B}_1$	•	•		•	•	•			•							•	•		•	•	•
$\mathcal{B}_2$	•	•	•	•	•	•		•		•	•					•	•	•	•	•	•
$\mathcal{B}_3$		•	•	•			•					•				•	•	•		•	•
$\mathcal{D}_1$	•	•			•				•	•		•	•	•	•				•	•	
$\mathcal{D}_2$	•	•		•		•	•		•	•	•	•	•	•	•				•	•	•
$\mathcal{D}_3$			•					•		•	•	•		•	•				•	•	•
$\mathcal{F}_1$	•				•	•		•	•	•			•	•		•	•	•			
$\mathcal{F}_2$		•	•		•	•	•	•	•	•		•	•	•	•	•	•	•			
$\mathcal{F}_3$				•		•	•	•			•		•	•	•		•	•			

Table 7.3: Possible combinations.

would assume that $\Gamma \cdot (z_1 \oplus z_2) = \Gamma \cdot (y \oplus x)$ with a bias of $\varepsilon_g(0, \Gamma)\varepsilon_g(\Gamma, \Gamma)$. However, if x is uniform distributed, $\Gamma \cdot x$ cannot add any bias to the function.

Therefore, we made a table, which shows which equations can be combined. A • indicates that we can use the two equations at the same time.

7.3.4 Bias of Equations

We tried to find the optimal mask to maximize the bias of a simplified equation. We expanded the method proposed in [CP07] such that we can use any combination of $\varepsilon_H(0, \Gamma)$, $\varepsilon_G(0, \Gamma)$, $\varepsilon_{H_i}(\Gamma, \Gamma)$, $\varepsilon_{G_i}(\Gamma, \Gamma)$ and $\varepsilon_+(\Gamma, \Gamma)$, however, always with the same mask. The program runs in less than 3 minutes. The idea is that we first define which bias we want to optimize and then look for which mask Γ we achieve the biggest bias. An example of the biases for the mask $\Gamma = 0x600018d$ is given in Table 7.4.

Example 1

We tried to find an equation which does not depend on $M_L(t)$. For our first example we used the following approximations at time t: $\Gamma \cdot H_2(x) = 0$, $\Gamma \cdot G_2(x) = \Gamma \cdot x$ and $\Gamma \cdot (x+y) = \Gamma \cdot (x \oplus y)$ for a' and $\Gamma \cdot H_3(x) = 0$, $\Gamma \cdot G_1(x) = 0$ and twice $\Gamma \cdot (x+y) = \Gamma \cdot (x \oplus y)$

$\varepsilon_+(\Gamma,\Gamma)$	$\varepsilon_G(0,\Gamma)$	$\varepsilon_H(0,\Gamma)$
$2^{-3.00}$	$2^{-8.99}$	$2^{-8.58}$
$\varepsilon_{G_1}(\Gamma,\Gamma)$	$\varepsilon_{G_2}(\Gamma,\Gamma)$	$\varepsilon_{G_3}(\Gamma,\Gamma)$
$2^{-13.59}$	$2^{-15.91}$	$2^{-12.89}$
$\varepsilon_{H_1}(\Gamma,\Gamma)$	$\varepsilon_{H_2}(\Gamma,\Gamma)$	$\varepsilon_{H_3}(\Gamma,\Gamma)$
$2^{-14.96}$	$2^{-12.64}$	$2^{-15.66}$

Table 7.4: Biases for $\Gamma = 0x600018d$.

for f' . Then we get

$$\begin{aligned}\Gamma \cdot a'(t) &= \Gamma \cdot \Big[[a(t) + (e(t) \oplus f(t))] \oplus H_1(t) \oplus e(t) \oplus f(t)\Big] ,\\ \Gamma \cdot e'(t) &= \Gamma \cdot \Big[[a(t) + (e(t) \oplus f(t))] \oplus H_1(t) \oplus e(t)\Big] ,\end{aligned}$$

and finally

$$\Gamma \cdot (a'(t) \oplus e'(t)) = \Gamma \cdot f(t) \tag{7.7}$$

with a bias of $\varepsilon_H(0,\Gamma)^2\varepsilon_G(0,\Gamma)\varepsilon_{G_2}(\Gamma,\Gamma)\varepsilon_+(\Gamma,\Gamma)^3$. In a next step, we use Table 7.1 to write

$$\Gamma \cdot (a'(t) \oplus e'(t)) = \Gamma \cdot (c'(t-16) \oplus M_L(t)) .$$

We can approximate $\Gamma{\cdot}c'(t{-}16)$ by $\Gamma{\cdot}c(t{-}16)$, by adding a bias of $\varepsilon_H(0,\Gamma)^2\varepsilon_{G_3}(\Gamma,\Gamma)\varepsilon_+(\Gamma,\Gamma)^2$ (Table 7.2). Thus we have

$$\Gamma \cdot (a'(t) \oplus e'(t)) = \Gamma \cdot (c(t-16) \oplus M_L(t)) .$$

From Table 7.1 we know that we can write directly

$$\Gamma \cdot (a'(t) \oplus e'(t)) = \Gamma \cdot (a(t-24) \oplus M_L(t)) .$$

In a last step, we use the approximation $\Gamma \cdot a(t-24) = \Gamma \cdot a'(t-24)$ which adds a bias of $\varepsilon_H(0,\Gamma)^2\varepsilon_{G_2}(\Gamma,\Gamma)\varepsilon_+(\Gamma,\Gamma)^2$. Thus we obtain the final approximation

$$\Gamma \cdot (a(t)' \oplus e(t)') = \Gamma \cdot (a'(t-24) \oplus M_L(t))$$

for which we have to consider the total bias of:

- $\varepsilon_+(\Gamma,\Gamma)^7$
- $\varepsilon_G(0,\Gamma)$
- $\varepsilon_H(0,\Gamma)^6$
- $\varepsilon_{G_2}(\Gamma,\Gamma)^2$
- $\varepsilon_{G_3}(\Gamma,\Gamma)$

Best bias: $2^{-126.17}$ with mask $0x600018d$.

Example 2

For this example, we assume that we can apply different approximations of a' at different instances of time. We start from equation (7.7), thus we have

$$\Gamma \cdot (a'(t) \oplus e'(t)) = \Gamma \cdot f(t) \ , \tag{7.8}$$

with a bias of $\varepsilon_H(0,\Gamma)^2 \varepsilon_G(0,\Gamma) \varepsilon_{G_2}(\Gamma,\Gamma) \varepsilon_+(\Gamma,\Gamma)^3$. We replace $f(t)$ by $b(t-11) \oplus M_R(t)$ (Table 7.1) and obtain

$$\Gamma \cdot (a'(t) \oplus e'(t)) = \Gamma \cdot [b(t-11) \oplus M_R(t)] \ . \tag{7.9}$$

In a next step we use the approximation $\Gamma \cdot a'(t-11) = \Gamma \cdot [b(t-11) \oplus c(t-11)]$ which adds a bias of $\varepsilon_H(0,\Gamma)^2 \varepsilon_G(0,\Gamma) \varepsilon_+(\Gamma,\Gamma)^3$ (Table 7.2) to

$$\Gamma \cdot (a'(t) \oplus e'(t)) = \Gamma \cdot [a'(t-11) \oplus c(t-11) \oplus M_R(t)] \ . \tag{7.10}$$

We use Table 7.1 to obtain

$$\Gamma \cdot (a'(t) \oplus e'(t)) = \Gamma \cdot [a'(t-11) \oplus a(t-19) \oplus M_R(t)] \ . \tag{7.11}$$

In a final step, we apply the approximation $\Gamma \cdot a(t) = \Gamma \cdot a'(t)$ which adds a bias of $\varepsilon_H(0,\Gamma)^2 \varepsilon_{G_2}(\Gamma,\Gamma) \varepsilon_+(\Gamma,\Gamma)^2$. We obtain the approximation

$$\Gamma \cdot (a(t)' \oplus e(t)') = \Gamma \cdot (a'(t-11) \oplus a'(t-19) \oplus M_R(t))$$

for which we have to count the following biases:

- $\varepsilon_+(\Gamma,\Gamma)^8$
- $\varepsilon_G(0,\Gamma)^2$
- $\varepsilon_H(0,\Gamma)^6$
- $\varepsilon_{G_2}(\Gamma,\Gamma)^2$

Best bias: $2^{-125.27}$ with mask $0x600018d$.

Result of Cho and Pieprzyk

In the end, we apply our program on the equation used in [CP07] (see Section 7.2.2):

$$\Gamma \cdot a(t)' = \Gamma \cdot (e'(t+15) \oplus M_L(t+15)) \ .$$

In contrary to their final result, we use the same mask in each step. We have to consider:

- $\varepsilon_+(\Gamma,\Gamma)^4$
- $\varepsilon_H(0,\Gamma)^4$
- $\varepsilon_{G_1}(\Gamma,\Gamma)$
- $\varepsilon_{G_2}(\Gamma,\Gamma)$

Best bias: $2^{-75.8}$ with mask $0x600018d$.

7.4 Conclusion

The Dragon stream cipher consists of a non-linear shift register on 32-bit words. It applies an additional 64 counter to guarantee a period length of 2^{64} and restricts the number of keystream bits generated from one key to 2^{64}. The non-linear function F is constructed by means of S-boxes which map 8-bit words to 32-bit words. We can write relations between words at different moments of time by applying the shift property of the main register. However, we were not successful in improving the distinguisher presented in [CP07].

We do not think that we can find a better distinguisher based on the bias of the S-boxes. There might be a possibility to attack the key/IV setup but until now we did not find any efficient approach. During the four year of the eSTREAM project, no real attack on Dragon was found.

Part IV

Random Functions

Chapter 8

Characteristics of Random Functions

Random functions are a probabilistic method of studying the characteristics of mappings from a set of n elements into itself. For simplification reasons we assume that these mappings work on the set $\{1, 2, \dots, n\}$.

Definition 8.1 *Let* $\mathcal{F}_n = \left\{\varphi \,\middle|\, \varphi : \{1, 2, \dots, n\} \to \{1, 2, \dots, n\}\right\}$ *be the set of all functions which map the set of n elements onto itself. We say that Φ is a* random function *or a* random mapping *if it takes each value $\varphi \in \mathcal{F}_n$ with the same probability $Pr[\Phi = \varphi] = 1/n^n$.*

For an extended definition of a random function we refer to the book of Kolchin [Kol86].

In cryptography, random mappings are often used for theoretical models where the attacker has no information about the function used. Thus, we assume that we have a random mapping. For example in Time Memory Tradeoff attacks for block or stream ciphers, the attacker assumes that a random function is applied [Hel80, Bab95, Gol97, BS00]. In [TccSD08], the authors consider stream ciphers where the key is fixed and the IV is randomly chosen and fixed except for ℓ positions. They compare the behavior of the function from the ℓ bits of the IV to the first ℓ bits of the keystream with a random function in $\mathcal{F}_\ell$. In Chapter 9, we consider collision attacks on a stream cipher model which applies a random function to update its inner state.

Parameters of random mappings have been studied for a long time [Har60, Ste69, Mut88]. Flajolet and Odlyzko present in [FO89] a general framework to analyze the asymptotic behavior of random function parameters. We will discuss this method in detail in Section 8.1. More recent results [DS97] consider for example the limiting distribution of the image and the preimage of random functions.

In the next section, we will discuss the approach proposed by Flajolet and Odlyzko [FO89]. Subsequently, we give a short overview of some of the basic characteristics of random functions. In Chapter 9, we examine a stream cipher model using a random function to update its internal state. First, we introduce the model in Section 9.1. We derive, in Section 9.2, a new parameter of random functions which can be used to evaluate the entropy of the inner state of such a stream cipher. In Section 9.3, we discuss the complexities of some attacks which try to exploit the entropy loss in our model.

8.1 Approach to Analyze Random Functions

In this section, we will discuss in the approach of Flajolet and Odlyzko [FO89] to find asymptotic parameters of random mappings. Their idea is to analyze the structure of the functional graph of the functions $\varphi \in \mathcal{F}_n$.

Definition 8.2 *For a function $\varphi \in \mathcal{F}_n$, the* functional graph *is defined by the set of vertexes $V(\varphi) = \{1, 2, \ldots, n\}$ and the set of directed edges $E(\varphi) = \left\{ \left(i, \varphi(i)\right) \middle| 1 \leq i \leq n \right\}$. An example of a functional graph can be seen in Figure 8.1.*

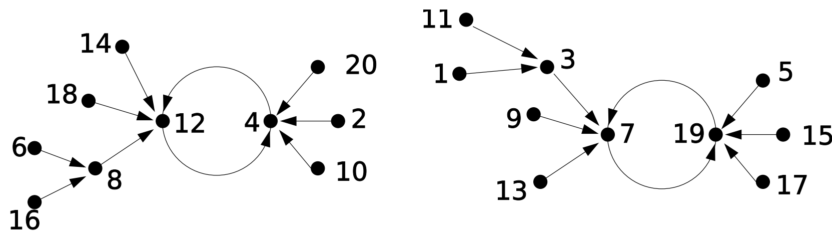


Figure 8.1: Example of a functional graph for $\varphi : x \mapsto ((x-1)^2 + 2) \pmod{20} + 1 \in \mathcal{F}_{20}$.

A functional graph of a finite function can be described as a set of cycles of trees, *i.e.* one or more components, where each component is a cycle of roots of trees. The exact composition can be displayed in the following way:

$$\begin{array}{lcl} \mathcal{T}ree & = & \mathcal{N}ode \star \text{SET}(\mathcal{T}ree) \;, \\ \mathcal{C}omponent & = & \text{CYC}(\mathcal{T}ree) \;, \\ \mathcal{F}ctgraph & = & \text{SET}(\mathcal{C}omponent) \;, \end{array} \tag{8.1}$$

where $\star$ denotes a product of two elements, and SET and CYC, respectively, the set and the cycle of elements from a given set. Flajolet and Odlyzko applied this structure on generating functions, as we will see in Section 8.1.1. To obtain the final parameters, they use singularity analysis on the generating functions. We briefly describe this final step in Section 8.1.2.

8.1.1 Use of Generating Function

Exponential generation functions can be used to describe labelled combinatorial structures, *i.e.* in these structures, each atom of an object is distinct from each other atoms by its label. In our case, the atoms of an object are nodes in a graph which are labelled from 1 to n.

Definition 8.3 *Let $\mathcal{A}$ denote a class of labeled combinatorial structures, and let a_n be the number of elements $\alpha \in \mathcal{A}$ of size $|\alpha| = n$. The* exponential generating function *(EGF) of*

the sequence $a_0, a_1, a_2, \ldots$ corresponding to $\mathcal{A}$ is defined by the function

$$A(z) = \sum_{n=0}^{\infty} a_n \frac{z^n}{n!} = \sum_{\alpha \in \mathcal{A}} \frac{z^{|\alpha|}}{|\alpha|!} \ . \tag{8.2}$$

Let $[z^n]A(z)$ denote the coefficient of z^n in the sum, then

$$a_n = n![z^n]A(z) \ . \tag{8.3}$$

The utilization of generating functions allows us to translate combinatorial constructions directly onto the corresponding generating functions. The next theorem gives a summary of some important structures and their corresponding manipulations of the generating functions.

Theorem 8.4 ([FS08]) *The associated operators on the exponential generating functions for the construction of combinatorial sum, labeled product, sequence, set, and cycle are as follows:*

$$\begin{array}{lll}
\textit{Sum}: & \mathcal{A} = \mathcal{B} + \mathcal{C} & A(z) = B(z) + C(z), \\
\textit{Product}: & \mathcal{A} = \mathcal{B} \star \mathcal{C} & A(z) = B(z) \times C(z), \\
\textit{Sequence}: & \mathcal{A} = \mathrm{SEQ}(\mathcal{B}) & A(z) = \frac{1}{1-B(z)}, \\
-\ k\ \textit{components}: & \mathcal{A} = \mathrm{SEQ_k}(\mathcal{B}) \equiv (\mathcal{B})^{\star k} & A(z) = B(z)^k, \\
\textit{Set}: & \mathcal{A} = \mathrm{SET}(\mathcal{B}) & A(z) = e^{B(z)}, \\
-\ k\ \textit{components}: & \mathcal{A} = \mathrm{SET_k}(\mathcal{B}) & A(z) = \frac{1}{k!}B(z)^k, \\
\textit{Cycle}: & \mathcal{A} = \mathrm{CYC}(\mathcal{B}) & A(z) = \log \frac{1}{1-B(z)}, \\
-\ k\ \textit{components}: & \mathcal{A} = \mathrm{CYC_k}(\mathcal{B}) & A(z) = \frac{1}{k}B(z)^k.
\end{array}$$

The notations $\mathrm{SEQ}_k(\mathcal{B})$, $\mathrm{SET}_k(\mathcal{B})$, and $\mathrm{CYC}_k(\mathcal{B})$ describe, respectively, a sequence, a set, and a cycle consisting of k elements of $\mathcal{B}$.

Applying this theorem to the structure in (8.1) leads to the following generating function for the set of functional graphs $\mathcal{F}ctgraph = \mathcal{F} = \bigcup_{n \geq 1} \mathcal{F}_n$. Here, $T(z)$, $C(z)$, and $F(z)$ are the generating functions for, respectively, the sets $\mathcal{T}ree$, $\mathcal{C}omponent$, and $\mathcal{F}ctgraph$:

$$\begin{array}{rcl}
T(z) & = & z e^{T(z)} \ , \\
C(z) & = & \log \frac{1}{1-T(z)} \ , \\
F(z) & = & \frac{1}{1-T(z)} \ .
\end{array} \tag{8.4}$$

These generating functions allows us to count the number of possible functional graphs of size n. However, if we want to consider an additional parameter of the graph we need bivariate generating functions.

Definition 8.5 ([FS08]) *Given a combinatorial class $\mathcal{A}$, a* parameter *is a function from $\mathcal{A}$ to $\mathbb{N}_0$ that associates to any object $\alpha \in \mathcal{A}$ an integer value $\xi(\alpha)$. The sequence*

$$a_{n,k} = \mathrm{card}\left(\{\alpha \in \mathcal{A} \big| |\alpha| = n, \xi(\alpha) = k\}\right) \ ,$$

is called the counting sequence *of pairs* $\mathcal{A}, \xi$*. The* (exponential) bivariate generating function (BGF) *of* $\mathcal{A}, \xi$ *is defined as*

$$A(z,u) := \sum_{n,k\geq 0} a_{n,k} \frac{z^n}{n!} u^k = \sum_{\alpha\in\mathcal{A}} \frac{z^{|\alpha|}}{|\alpha|!} u^{\xi(\alpha)} .$$

One says that the variable z *marks size and the variable* u *marks the parameter* ξ*.*

Let $\mathcal{A}_n$ define the subset of elements $\alpha \in \mathcal{A}$ of size $|\alpha| = n$. The probability of an element $\alpha \in \mathcal{A}_n$ to have $\xi(\alpha) = k$ is given by

$$Pr_{\mathcal{A}_n}[\xi = k] = \frac{a_{n,k}}{a_n} = \frac{[z^n][u^k]A(z,u)}{[z^n]A(z,1)} .$$

Then, we can give expectation and the variance of the parameter ξ in the following way:

Proposition 8.6 ([FS08]) *The first two moments of the parameter* ξ *can be determined from the BGF by:*

$$\mathbf{E}_{\mathcal{A}_n}(\xi) = \frac{[z^n]\partial_u A(z,u)|_{u=1}}{[z^n]A(z,1)} , \tag{8.5}$$

$$\mathbf{E}_{\mathcal{A}_n}\left(\xi^2\right) = \frac{[z^n]\partial_u^2 A(z,u)|_{u=1}}{[z^n]A(z,1)} + \frac{[z^n]\partial_u A(z,u)|_{u=1}}{[z^n]A(z,1)} , \tag{8.6}$$

and the variance by:

$$\mathbf{V}_{\mathcal{A}_n}(\xi) = \mathbf{E}_{\mathcal{A}_n}\left(\xi^2\right) - \mathbf{E}_{\mathcal{A}_n}(\xi)^2 . \tag{8.7}$$

Equation (8.5) is easy to see, if we differentiate the sum

$$\sum_k Pr_{\mathcal{A}_n}[\xi = k]u^k = \frac{[z^n]A(z,u)}{[z^n]A(z,1)}$$

with respect to u and if we consider the equation for the expectation

$$\mathbf{E}_{\mathcal{A}_n}(\xi) = \sum_k k Pr_{\mathcal{A}_n}[\xi = k] .$$

In the following, we consider parameters which count the number of specific components in an element $\alpha \in \mathcal{A}$. This can be done by marking the desired component with the variable u in the bivariate generating function. Let us consider the case of random mappings. The BGF counting the number of cycle nodes in the functional graph is

$$F_C(z,u) = \exp\left(\log \frac{1}{1-uT(z)}\right) = \frac{1}{1-uT(z)} .$$

If we want to count the number of cycles of size k, we can use the operator for CYC_k in Theorem 8.4 to get the corresponding BGF

$$F_{C_k}(z,u) = \exp\left(\log\frac{1}{1-T(z)} + (u-1)\frac{T(z)^k}{k}\right) .$$

The previous generating function can be used to compute the probability distribution of the size of the cycles in functional graph of a random mapping (*cf.* Theorem 8.11).

In this section, we consider only the simple counting of objects in the generating function, since this is the approach that we follow in Section 9.2. In their article, Flajolet and Odlyzko examine as well the generating functions of cumulative parameters, like the average tail length or the average cycle length, and extremal statistics, like the maximal tail length or the maximal cycle length. We refer the interested reader to their article or to the monograph of Flajolet and Sedgewick [FS08].

Now that we know how to find the desired generating functions, we want to extract their asymptotic coefficients. This can be done by means of singularity analysis, as we describe in the next section.

8.1.2 Singularity Analysis

Flajolet and Odlyzko developed in [FO90] the idea of analyzing the singularities of a generating function to obtain asymptotic values of the coefficients. Subsequently, they applied this method to study the parameters of random mappings [FO89]. The topic was further developed [Fla99, BFSS01, FFK05, FFG+06] to help in many other areas. A detailed discussion of this approach is given in the monograph [FS08] of Flajolet and Sedgewick.

Let us compare the generating functions and the asymptotic form of their coefficients in Table 8.1.2. We see that the place of the singularity has an influence on the coefficient.

$$\begin{array}{lcl}
[z^n]\frac{1}{1-3z} & = & 3^n \\
[z^n]\frac{1}{1-4z} & = & 4^n \\
[z^n]\frac{1}{\sqrt{1-4z}} & \sim & \frac{4^n}{\sqrt{\pi n}} \\
[z^n]\frac{1}{\sqrt{1-4z}}\log\frac{1}{1-4z} & \sim & \frac{4^n}{\sqrt{\pi n}}\log(n)
\end{array}$$

Table 8.1: Some examples of generating functions and their coefficients.

Likewise, the square root and the logarithm of the generating function appears again in the asymptotic coefficients. The next theorem shows the connection between the generating function and the coefficients.

Theorem 8.7 ([FO89]) *Let $f(z)$ be a function analytic in a domain*

$$\mathcal{D} = \left\{ z \;\middle|\; |z| \le s_1, |Arg(z-s)| > \frac{\pi}{2} - \eta \right\} ,$$

where $s, s_1 > s$ and η are three positive real numbers. Assume that, with $\sigma(u) = u^\alpha \log^\beta u$ and $\alpha \notin \{0, -1, -2, \ldots\}$, we have

$$f(z) \sim \sigma\left(\frac{1}{1 - z/s}\right) \qquad \text{as } z \to s \text{ in } \mathcal{D} .$$

Then, the Taylor coefficients of $f(z)$ satisfy

$$[z^n]f(z) \sim s^{-n}\frac{\sigma(n)}{n\Gamma(\alpha)} ,$$

where $\Gamma(\alpha)$ denotes the Gamma function.

Flajolet and Odlyzko showed that this theorem can be applied directly to the generating function of a tree due to the following property.

Proposition 8.8 ([FO89]) *The tree function $T(z)$ defined by $T(z) = z^{T(z)}$ is analytic in the domain $\mathcal{D}$ formed by the complex plane slit along $(e^{-1}, +\infty)$. For z tending to e^{-1} in $\mathcal{D}$, $T(z)$ admits the singular expansion*

$$T(z) = 1 - \sqrt{2}\sqrt{1 - ez} - \frac{1}{3}(1 - ez) + O((1 - ez)^{3/2}) . \tag{8.8}$$

If we apply Theorem 8.7 to a function $f(z) = \gamma g(z) + \eta h(z)$, then the corresponding Taylor coefficients have the form $f_n = \gamma g_n + \eta h_n$. Thus, for the asymptotic value of the coefficient we are interested only in the dominant part of the generating function. Let us consider the example of the generating function (8.8) of a tree. We have two different functions $f(z) \sim \left(\frac{1}{1-ze}\right)^{-1/2}$ and $g(z) \sim \left(\frac{1}{1-ze}\right)^{-3/2}$ on which we can apply the singularity analysis. The coefficients of those two functions grow, respectively, with $\mathcal{O}\left(e^{-n}n^{-3/2}\right)$ and $\mathcal{O}\left(e^{-n}n^{-5/2}\right)$. For $n \to \infty$, the dominant part corresponds to the function $f(z)$. Thus, to obtain the number of trees of size n we consider the function $\gamma f(z)$ with $\gamma = -\sqrt{2}$. Using $\Gamma(-1/2) = -2\sqrt{\pi}$ we get the asymptotic coefficients

$$[z^n]\gamma f(z) \sim \sqrt{2}e^n \frac{n^{-1/2}}{2n\sqrt{\pi}} .$$

By applying the Stirling approximation

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n ,$$

and $t_n = n![z^n]T(z)$ (8.3) we get

$$t_n \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \sqrt{2}e^n \frac{n^{-3/2}}{2\sqrt{\pi}} = n^{n-1} \tag{8.9}$$

for $n \to \infty$. The value n^{n-1} is not only the asymptotic, but the exact number of trees of size n. Thus, this proof can be used to show inversely the Stirling approximation.

We discussed the efficient approach of Flajolet and Odlyzko [FO89] to analyze parameters of random mappings. In the next section, we see the resulting asymptotic values for some interesting parameters.

8.2 Known Properties

In this section, we review some parameters which might be interesting for the use of random functions in cryptography. Most of the results are given by Flajolet and Odlyzko in [FO89], even if some of them were already known before [Har60, PW68, Ste69, AB82]. We also briefly discuss some results concerning the distribution properties of images and preimages [DS97].

The first theorem states the expectation of the number of occurrences of some objects in the functional graph. All results are asymptotic values for $n \to \infty$. For a given function $\varphi \in \mathcal{F}_n$, let a component denote a cycle of a trees, and a cycle node a node in a cycle. A terminal node x is a leave in a tree, which corresponds to a node with empty preimage $\varphi^{-1}(x) = \emptyset$. The image points are the nodes in the image $\varphi\big(\{1, 2, \ldots, n\}\big)$, *i.e.* all the nodes which are no terminal nodes. The k-th iterate image points are all the nodes in the image $\varphi^k\big(\{1, 2, \ldots, n\}\big)$ after k iterations of φ.

Theorem 8.9 ([FO89]) *The expectations of parameters number of components, number of cyclic points, number of terminal points, number of image points, and number of k-th iterate image points in a random mapping of size n have the asymptotic forms, as $n \to \infty$,*

1. *# Components:* $\frac{1}{2} \log n$,
2. *# Cyclic nodes:* $\sqrt{\pi n/2}$,
3. *# Terminal nodes:* $e^{-1} n$,
4. *# Image points:* $(1 - e^{-1}) n$,
5. *# k-th iterate image points:* $(1 - \tau_k) n$,

where τ_k satisfy the recurrence $\tau_0 = 0$ and $\tau_{k+1} = e^{-1+\tau_k}$.

Lemma 8.10 *The limit of the sequence τ_k is*

$$\lim_{k \to \infty} \tau_k = 1 \ .$$

Proof.
The exponential function is strictly increasing. Thus, we can write for all $k \geq 0$:

- The sequence τ_k is strictly increasing, *i.e.* $\tau_k < \tau_{k+1}$.
- If $0 \leq \tau_k < 1$ then $e^{-1} \leq \tau_{k+1} < 1$.

From this follows that the sequence τ_k has a limit L such that $e^{-1} \leq L \leq 1$. For any k we can write $\tau_k = L - \varepsilon$. Thus, for the limit L must hold that for every $0 < \varepsilon \leq L$:

$$e^{-1+L-\varepsilon} \leq L \ .$$

The only possible value L, with $e^{-1} \leq L \leq 1$, for which holds $e^{-\varepsilon} \leq Le^{1-L}$ for all $0 < \varepsilon \leq L$ is $L = 1$. ◇

Lemma 8.10 shows that the value $(1 - \tau_k)n$ decreases down to 0 when k goes to $+\infty$. However, the number of k-th iterate image points cannot decrease below the number of cycle points. Thus, the asymptotic value of parameter 5. is only valid as long as it is larger than $\sqrt{\pi n/2}$.

If we want to know the probability distribution of, for example, the size of cycles, we need to know the average number of cycle of size r for $1 \leq r \leq n$. The next theorem addresses this problems for some parameters. Let an r-node denote a node of indegree r. By an r-predecessor tree we mean an arbitrary tree in the functional graph of size r, whereas, by an r-cycle tree, we mean a tree of size r which is rooted in a cycle. By an r-cycle and an r-component we denote, respectively, a cycle and a component of size r.

Theorem 8.11 ([FO89]) *For any fixed integer r, the parameters number of r-nodes, number of predecessor trees of size r, number of cycle trees of size r and number of components of size r, have the following asymptotic mean values:*

1. *# r-nodes:* $ne^{-1}/r!$,
2. *# r-predecessor trees:* $nt_r e^{-r}/r!$,
3. *# r-cycle trees:* $\left(\sqrt{\pi n/2}\right) t_r e^{-r}/r!$,
4. *# r-cycles:* $1/r$,
5. *# r-components:* $c_r e^r/r!$,

where $t_r = r^{r-1}$ is the number of trees having r nodes, and $c_r = r![z^r]c(z)$ is the number of connected mappings of size r.

Until now, we considered parameters of the whole graph. In the next theorem, we consider parameters of the nodes in the functional graph. Each node $x \in \{1, 2, \ldots, n\}$ in the functional graph of $\varphi \in \mathcal{F}_n$ is part of a tree which is rooted in a cycle. We denote the distance of x to its root by the tail length $\lambda_\varphi(x)$ and the length of its cycle by $\mu_\varphi(x)$. The whole path from x until the first value k such that $\varphi^k(x) = \varphi^j(x)$ for an $j < k$ is defined by the rho length $\rho_\varphi(x) = \lambda_\varphi(x) + \mu_\varphi(x)$. The tree size denotes the size of the whole tree which contains x, whereas, the predecessors size denotes the size of the tree of which x is the root. The component size gives the size of the component which contains x.

Theorem 8.12 ([FO89]) *Seen from a random point in a random mapping of $\mathcal{F}_n$, the expectations of parameters tail length, cycle length, rho-length, tree size, component size, and predecessors size have the following asymptotic forms:*

1. *Tail length (λ):* $\sqrt{\pi n/8}$,

2. *Cycle length* (μ): $\sqrt{\pi n/8}$,

3. *Rho length* ($\rho = \lambda + \mu$): $\sqrt{\pi n/2}$,

4. *Tree size:* $n/3$,

5. *Component size:* $2n/3$,

6. *Predecessors size:* $\sqrt{\pi n/8}$.

The following three theorems consider maximal statistics, this means, they consider the maximal value of a parameter on average over all functions $\varphi \in \mathcal{F}_n$.

Theorem 8.13 ([FO89]) *The expectation of the* maximum cycle length ($\mu^{\max}$) *in a random mapping of* $\mathcal{F}_n$ *satisfies*

$$\mathbf{E}\left(\mu^{\max}\right) \sim c_1\sqrt{n} ,$$

where $c_1 \approx 0.78248$ *is given by*

$$c_1 = \sqrt{\frac{\pi}{2}} \int_0^\infty \left(1 - e^{-E_1(x)}\right) dx ,$$

and $E_1(x)$ *denotes the exponential integral*

$$E_1(x) = \int_x^\infty e^{-y} \frac{dy}{y} .$$

Theorem 8.14 ([FO89]) *The expectation of the* maximal tail length ($\lambda^{\max}$) *in a random mapping of* $\mathcal{F}_n$ *satisfies*

$$\mathbf{E}\left(\lambda^{\max}\right) \sim c_2\sqrt{n} ,$$

where $c_2 \approx 1.73746$ *is given by*

$$c_2 = \sqrt{2\pi}\log(2) .$$

Theorem 8.15 ([FO89]) *The expectation of the* maximal rho length ($\rho^{\max}$) *in a random mapping of* $\mathcal{F}_n$ *satisfies*

$$\mathbf{E}\left(\rho^{\max}\right) \sim c_3\sqrt{n} ,$$

where $c_3 \approx 2.4149$ *is given by*

$$c_3 = \sqrt{\frac{\pi}{2}} \int_0^\infty \left(1 - e^{-E_1(x)-I(x)}\right) dx ,$$

with $E_1(x)$ *denoting the exponential integral and*

$$I(x) = \int_0^x e^{-y} \left(1 - \exp\left(\frac{-2y}{e^{x-y}-1}\right)\right) \frac{dy}{y} .$$

In the end of this section, we consider some probability distribution of random function parameters. Drmota and Soria [DS97] study the generating functions of some parameters concerning the image and the preimage of random functions. Those generating functions have a specific structure, and the authors could give some general theorems on the distribution of these parameters. The exact theorems fill more than a page, thus in this section, we cite only some of the applications on random function. We denote by $\mathcal{N}(\mu, \sigma^2)$ the Gaussian distribution with mean μ and variance σ^2 and by $\mathcal{R}(s)$ the Rayleigh distribution with parameter s.

Theorem 8.16 ([Mut89, DS97]) *Let X_n denote the number of noncyclic points at a fixed distance $d > 0$ to a cycle in random mappings of size n. Then, X_n has the Rayleigh limiting distribution*

$$\frac{X_n}{\sqrt{n}} \xrightarrow{d} \mathcal{R}(1) . \tag{8.10}$$

Theorem 8.17 ([AB82, DS97]) *Let $r \geq 0$ be a fixed integer, and let X_n denote the number of points v with $|\varphi^{-1}(\{v\})| = r$ in mappings $\varphi \in \mathcal{F}_n$,* i.e. *$v$ is an r-node. Then*

$$\frac{X_n - \mu n}{\sqrt{\sigma^2 n}} \xrightarrow{d} \mathcal{N}(0,1) , \tag{8.11}$$

where $\mu = \frac{1}{er!}$ and $\sigma^2 = \mu + (1 + (r-1)^2)\mu^2$.

Theorem 8.18 ([DS97]) *Let $d \geq 0$ be a fixed integer, and let X_n denote the number of points v with*

$$\left| \bigcup_{d \geq 0} \varphi^{-d}(\{v\}) \right| = 0$$

i.e. *v can be reached by r different nodes, in mappings $\varphi \in \mathcal{F}_n$. Then,*

$$\frac{X_n - \mu n}{\sqrt{\sigma^2 n}} \xrightarrow{d} \mathcal{N}(0,1) , \tag{8.12}$$

where $\mu = \frac{r^{r-1}}{r!} e^{-r}$ and $\sigma^2 = \mu - 2r\mu^2$.

Theorem 8.19 ([DS97]) *Let $d \geq 0$ be a fixed integer, and let X_n denote the number of points $v \in \varphi^d(\{1, 2, \ldots, n\})$,* i.e. *the d-th iterate image size, of mappings $\varphi \in \mathcal{F}_n$. Then*

$$\frac{X_n - \mu n}{\sqrt{\sigma^2 n}} \xrightarrow{d} \mathcal{N}(0,1) , \tag{8.13}$$

where $\mu = h_d\left(\frac{1}{e}\right)$, $\sigma^2 = \frac{2}{e} h'_d\left(\frac{1}{e}\right)(1-\mu) - \mu$, and $h_d(x)$ is the exponential generating function of Cayley trees and satisfies the recursion $h_0(x) = 0$ and $h_{i+1}(x) = xe^{h_i(x)}$ for $i \geq 0$.

In this section, we saw some interesting parameters for random functions. In the next chapter, we present a new parameter which can be used to compute the expected state entropy of a stream cipher having a random update function.

Chapter 9

State Entropy of a Stream Cipher using a Random Function

Replacing random permutations by random functions for the update of a stream cipher introduces the problem of entropy loss. To assess the security of such a design, we need to evaluate the entropy of the inner state. We propose a new approximation of the entropy for a limited number of iterations. Subsequently, we discuss two collision attacks which are based on the entropy loss. We provide a detailed analysis of the complexity of those two attacks as well as of a variant using distinguished points.

The work from this chapter was presented at [Röc07b, Röc07a] and published in the proceedings of the international conference on cryptology in Africa (Africacypt 2008) in Casablanca, Morocco [Röc08b].

9.1 Introduction

Recently, several stream ciphers have been proposed with a non-bijective update function. Moreover, in some cases the update function seems to behave like a random function as for the Mickey stream cipher [BD05, BD08]. Using a random function instead of a random permutation induces an entropy loss in the state. An attacker might exploit this fact to mount an attack. Particularly, we will study some attacks which apply the approach of Time–Memory tradeoff [Hel80] and its variants [HS05]. At first we introduce the model with which we are going to work.

Stream Cipher Model The classical model of an additive synchronous stream cipher (Figure 9.1) is composed of an internal state updated by applying a function Φ. Then a filter function is used to extract the keystream bits from the internal state. To obtain the ciphertext we combine the keystream with the plaintext.

The particularity of our model is that Φ is a *random mapping* (Definition 8.1) which allows us to make some statistical statements about the properties of the stream cipher.

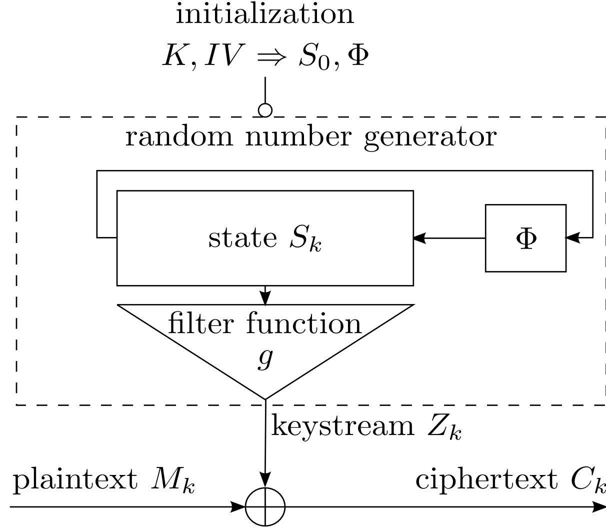


Figure 9.1: Model of a simple stream cipher.

Let S_k be the random variable denoting the value of the state after k iterations of Φ, for $k \geq 0$. From the model in Figure 9.1, we see that $S_k = \Phi(S_{k-1})$ where the value of Φ is the same for all iterations $k \geq 1$. The value of the keystream is given by $Z_k = g(S_k)$ and the ciphertext by $C_k = M_k \oplus Z_k$. The probability distribution of the initial state S_0 is $\{p_i\}_{i=1}^n$ such that

$$p_i = Pr[S_0 = i] \; .$$

If we do not state otherwise, we assume a uniform distribution. Thus $p_i = 1/n$ for all $1 \leq i \leq n$. We describe by

$$p_i^\Phi(k) = Pr[\Phi^k(S_0) = i]$$

the probability of the state being i after applying k times Φ on the initial state S_0. If we write only $\mathrm{p}_i^\varphi(k)$ we mean the same probability but for a specific function $\varphi \in \mathcal{F}_n$. The notation above allows us to define the entropy of the state after k iterations of Φ as:

$$H_k^\Phi = \sum_{i=1}^{n} p_i^\Phi(k) \log_2 \left(\frac{1}{p_i^\Phi(k)} \right) \; .$$

If $p_i^\Phi(k) = 0$, we use the classical convention in the computation of the entropy that $0 \log_2(\frac{1}{0}) = 0$. This can be done, since a zero probability has no influence on the computation of the entropy. In this chapter, we are interested in expectations where the average is taken over all functions $\varphi \in \mathcal{F}_n$. To differentiate between a value corresponding to a random mapping Φ, to a specific function φ, and the expectation of a value, taken over all functions $\varphi \in \mathcal{F}_n$, we will write in the following the first one italic (*e.g.* H_k^Φ), the second one upright (*e.g.* H_k^φ), and the last one bold (*e.g.* $\mathbf{H_k}$). For instance, the formula:

$$\mathbf{H_k} = \mathbf{E}(H_k^\Phi)$$

denotes the expected state entropy after k iterations.

The subsequent chapter is divided in two main sections. In Section 9.2, we discuss ways of estimating the state entropy of our model. We give a short overview of previous results from [FO89] and [HK05] in Section 9.2.1. Subsequently in Section 9.2.2, we present a new estimate which is, for small numbers of iterations, more precise than the previous one. In Section 9.3, we examine if it is possible to use the entropy loss in the state to launch an efficient attack on our model. We discuss two collision attacks against MICKEY version 1 [BD05] presented in [HK05]. We give a detailed evaluation of the costs of these attacks applied to our model. For this evaluation, we consider the space complexity, the query complexity and the number of different initial states needed. By the space complexity we mean the size of the memory needed, by the query complexity we mean the number of times we have to apply the update function during the attack. For the first attack, we show that we only gain a factor on the space complexity by increasing the query complexity by the same factor. For the second attack, we demonstrate that, contrary to what is expected from the results in [HK05], the complexities are equivalent to a direct collision search in the initial values. In the end, we present a new variant of these attacks which allows to reduce the space complexity; however the query complexity remains the same.

9.2 Estimation of Entropy

The entropy is a measure of the unpredictability. An entropy loss in the state facilitates the guessing of the state for an adversary. In this section, we therefore discuss different approaches to estimate the expected entropy of the inner state.

9.2.1 Previous Work

As we presented in the previous chapter, Flajolet and Odlyzko provide, in [FO89], a wide range of parameters of random functions by analyzing their functional graphs. To study the state entropy in our model, we are interested in the average value of the maximal tail length, the number of cyclic nodes, the number of r-nodes and the number of image points after some iterations of functions $\varphi \in \mathcal{F}_n$. The maximal tail length is, for each graph, the maximal number of steps before reaching a cycle. An r-node is a node in the graph with exactly r incoming nodes which is equivalent to a preimage of size r. By the image points we mean all points in the graph that are reachable after k iterations of the function. From Theorem 8.9, Theorem 8.11, and Theorem 8.14 we know the asymptotic values of these parameters for $n \to \infty$:

- the expected number of cycle point $\mathbf{cp}(n) \sim \sqrt{\pi n/2}$,
- the expected maximal tail length $\mathbf{mt}(n) \sim \sqrt{\pi n/8}$,
- the expected number of r-nodes $\mathbf{rn}(n, r) \sim \frac{n}{r!e}$, and

- the expected number of image points after k iterations $\mathbf{ip}(n,k) \sim n(1-\tau_k)$ where $\tau_0 = 0$ and $\tau_{k+1} = e^{-1+\tau_k}$.

For all these values, the expectation is taken over all functions in $\mathcal{F}_n$.

In [HK05], Hong and Kim use the expected number of image points to give an upper bound for the state entropy after k iterations of a random function. They utilized the fact that the entropy is always less than or equal to the logarithm of the number of points with probability larger than zero. After a finite number of steps, each point in the functional graph will reach a cycle, and thus the number of image points can never drop below the number of cycle points. Therefore, the upper bound for the estimated entropy of the internal state

$$\mathbf{H_k} \leq \log_2(n) + \log_2(1-\tau_k) \tag{9.1}$$

is valid only as long as $\mathbf{ip}(n,k) > \mathbf{cp}(n)$. We see that for this bound the loss of entropy only depends on k and not on n.

In Figure 9.2, we compare, for $n = 2^{16}$, the values of this bound with the empirically derived average of the state entropy. To compute this value we chose 10^4 functions, using

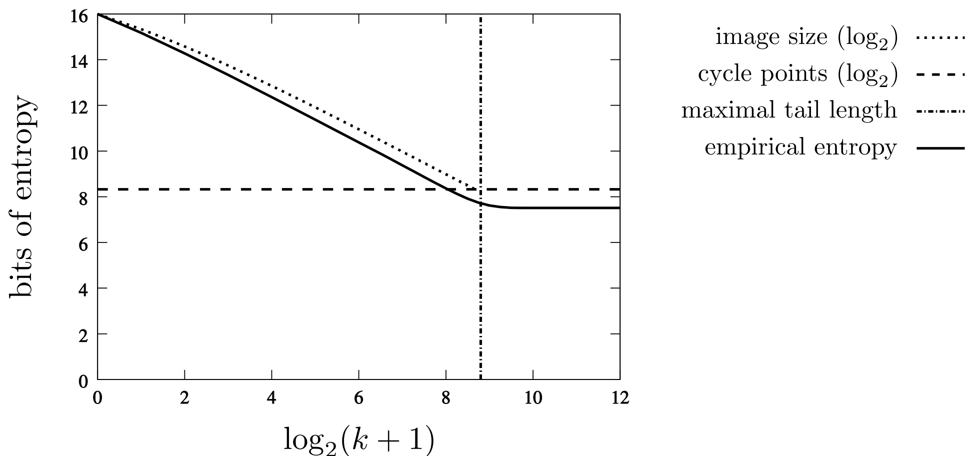


Figure 9.2: Upper bound and empirical average of the entropy for $n = 2^{16}$.

the HAVEGE random number generator [SS03], and computed the average entropy under the assumption of a uniform distribution of the initial state. Even if n is not very big, it is sufficient to understand the relation among the different factors. We can see in the graph that if k stays smaller than $\mathbf{mt}(n)$ this bound stays valid and does not drop under $\log_2(\mathbf{cp}(n))$.

9.2.2 New Entropy Estimation

The expected number of image points provides only an upper bound (9.1) of the expected entropy. We found a more precise estimation by employing the methods stated in [FO89] and discussed in Section 8.1.

For a given function $\varphi \in \mathcal{F}_n$, let i be a node with r incoming nodes (an r-node). The idea is that this is equivalent to the fact that i is produced by exactly r different starting values after one iteration. Thus, if the initial distribution of the state is uniform, this node has the probability $\mathrm{p}_i^{\varphi}(1) = r/n$. The same idea works also for more than one iteration.

Definition 9.1 *For a fixed n, we choose a function $\varphi \in \mathcal{F}_n$. Let $\varphi^{-k}(i) = \{j|\varphi^k(j) = i\}$ define the* preimage *of i after k iterations of φ. We denote by*

$$\mathrm{rn}_k^{\varphi}(r) = \# \left\{ i \middle| \left|\varphi^{-k}(i)\right| = r \right\}$$

the number of points in the functional graph of φ which are reached by exactly r nodes after k iterations.

For a random function Φ on a set of n elements, we define by $\mathbf{rn_k}(n, r)$ the expected value of $\mathrm{rn}_k^{\varphi}(r)$, thus

$$\mathbf{rn_k}(n, r) = \frac{1}{n^n} \sum_{\varphi \in \mathcal{F}_n} \mathrm{rn}_k^{\varphi}(r) \ .$$

A small example might help to better understand these definitions. For $n = 13$ we consider

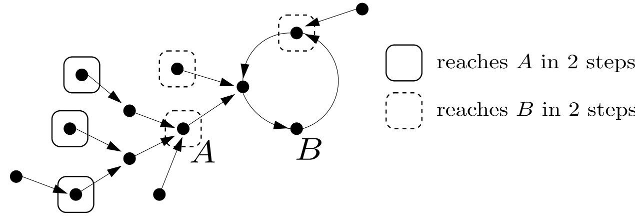


Figure 9.3: Example of a functional graph to illustrate $\mathrm{rn}_k^{\varphi}(r)$.

a function φ with a functional graph as displayed in Figure 9.3. The only points that are reached by $r = 3$ points after $k = 2$ iterations are A and B. Thus, in this case we have $\mathrm{rn}_2^{\varphi}(13, 3) = 2$. The value $\mathbf{rn_2}(13, 3)$ is then the average taken over all functions $\varphi \in \mathcal{F}_{13}$.

Using Definition 9.1 we can state the following theorem.

Theorem 9.2 *In the case of a uniform initial distribution, the expected entropy of the inner state after k iterations is*

$$\mathbf{H_k} = \log_2(n) - \sum_{r=1}^{n} \mathbf{rn_k}(n, r) \ \frac{r}{n} \ \log_2(r) \ . \tag{9.2}$$

Proof.
Let us fix a function φ. We use the idea that after k iterations of φ we have $\mathrm{rn}_k^{\varphi}(r)$ states

with probability $\frac{r}{n}$. Thus, the entropy after k iterations for this specific function is

$$\begin{aligned}
\mathrm{H}_k^{\varphi} &= \sum_{r=1}^{n} \mathrm{rn}_k^{\varphi}(r) \frac{r}{n} \log_2 \left(\frac{n}{r}\right) \\
&= \log_2(n) \frac{1}{n} \sum_{r=1}^{n} r \, \mathrm{rn}_k^{\varphi}(r) - \sum_{r=1}^{n} \mathrm{rn}_k^{\varphi}(r) \frac{r}{n} \log_2(r) \ .
\end{aligned}$$

We ignore the case $r = 0$ since it corresponds to a probability zero, which is not important for the computation of the entropy. Each $1 \leq j \leq n$ appears exactly in one preimage of φ after k iterations. We see directly from the definition of $\mathrm{rn}_k^{\varphi}(r)$ that $\sum_{r=1}^{n} r \, \mathrm{rn}_k^{\varphi}(r) = n$. Therefore, we can write

$$\mathrm{H}_k^{\varphi} = \log_2(n) - \sum_{r=1}^{n} \mathrm{rn}_k^{\varphi}(r) \frac{r}{n} \log_2(r) \ .$$

By using this equation, we can give the expected entropy after k iterations as

$$\begin{aligned}
\mathbf{H_k} &= \frac{1}{n^n} \sum_{\varphi \in \mathcal{F}_n} \mathrm{H}_k^{\varphi} \\
&= \frac{1}{n^n} \sum_{\varphi \in \mathcal{F}_n} \left[\log_2(n) - \sum_{r=1}^{n} \mathrm{rn}_k^{\varphi}(r) \frac{r}{n} \log_2(r) \right] \\
&= \log_2(n) - \frac{1}{n^n} \sum_{\varphi \in \mathcal{F}_n} \left[\sum_{r=1}^{n} \mathrm{rn}_k^{\varphi}(r) \frac{r}{n} \log_2(r) \right] \ .
\end{aligned}$$

Since we only have finite sums we can change the order:

$$\mathbf{H_k} = \log_2(n) - \sum_{r=1}^{n} \left[\frac{1}{n^n} \sum_{\varphi \in \mathcal{F}_n} \mathrm{rn}_k^{\varphi}(r) \right] \frac{r}{n} \log_2(r) \ .$$

We conclude our proof by applying Definition 9.1. $\diamond$

In the same way we can compute the entropy for any arbitrary initial distribution.

Theorem 9.3 *For a given n, let $P = \{p_1, p_2, \dots, p_n\}$ define the distribution of the initial state. Then, the expected entropy of the state after k iterations is given by*

$$\mathbf{H_k^P} = \sum_{r=1}^{n} \mathbf{rn_k}(n, r) \frac{1}{\binom{n}{r}} \sum_{1 \leq j_1 < \cdots < j_r \leq n} (p_{j_1} + \cdots + p_{j_r}) \log_2 \frac{1}{p_{j_1} + \cdots + p_{j_r}} \ . \tag{9.3}$$

Proof.
Let us choose a specific φ and a state i. After k iterations of φ, the state i has the probability $\sum_{j\in\varphi^{-k}(i)} p_j$. Therefore, the expected entropy after k iterations is given by

$$\mathbf{H}_\mathbf{k}^\mathbf{P} = \frac{1}{n^n} \sum_{\varphi\in\mathcal{F}_n} \sum_{i=1}^{n} \left(\sum_{j\in\varphi^{-k}(i)} p_j \right) \log_2 \left(\frac{1}{\sum\limits_{j\in\varphi^{-k}(i)} p_j} \right) . \tag{9.4}$$

For a given r we fix a set of indices $\{j_1, \ldots, j_r\}$. Without loss of generality we assume that they are ordered, *i.e.* $1 \leq j_1 < \cdots < j_r \leq n$. We now want to know how many times we have to count $(p_{j_1} + \cdots + p_{j_r}) \log_2 \frac{1}{p_{j_1}+\cdots+p_{j_r}}$ in (9.4). This is equivalent to the number of pairs (i, φ) where $\varphi^{-k}(i) = \{j_1, \ldots, j_r\}$.

From Definition 9.1, we know that $n^n\mathbf{rn_k}(n, r)$ is the number of pairs (i, φ) such that $|\varphi^{-k}(k)| = r$. Due to symmetry, each set of indices of size r is counted the same number of times in (9.4). There are $\binom{n}{r}$ such sets. Thus, $(p_{j_1} + \cdots + p_{j_r}) \log_2 \frac{1}{p_{j_1}+\cdots+p_{j_r}}$ is counted exactly $\frac{n^n\mathbf{rn_k}(n,r)}{\binom{n}{r}}$ times and we can write

$$\mathbf{H}_\mathbf{k}^\mathbf{P} = \frac{1}{n^n} \sum_{r=1}^{n} \frac{n^n\mathbf{rn_k}(n, r)}{\binom{n}{r}} \sum_{1\leq j_1<\cdots<j_r\leq n} (p_{j_1} + \cdots + p_{j_r}) \log_2 \frac{1}{p_{j_1} + \cdots + p_{j_r}} ,$$

which is equivalent to (9.3). ◇

Theorem 9.2 can also be shown by using Theorem 9.3; however the first proof is easier to follow. Finally, we want to consider a further special case.

Corollary 9.4 *For a given n, let the distribution of the initial state be $P_m = \{p_1, p_2, \ldots, p_n\}$. From the n possible initial values only m occur with probability exactly $\frac{1}{m}$. Without loss of generality we define*

$$p_i = \begin{cases} \frac{1}{m} & 1 \leq i \leq m \\ 0 & m < i \leq n . \end{cases}$$

In this case, we get

$$\mathbf{H}_\mathbf{k}^{\mathbf{P_m}} = \sum_{r=1}^{n} \mathbf{rn_k}(n, r) \frac{1}{\binom{n}{r}} \sum_{\ell=0}^{r} \binom{m}{\ell} \binom{n-m}{r-\ell} \frac{\ell}{m} \log_2 \frac{m}{\ell} . \tag{9.5}$$

Proof.
For a given r, let us consider the sum $(p_{j_1} + \cdots + p_{j_r})$ for all possible index tuples $1 \leq j_1 < \cdots < j_r \leq n$. In $\binom{m}{\ell}\binom{n-m}{r-\ell}$ cases we will have $(p_{j_1} + \cdots + p_{j_r}) = \frac{\ell}{m}$ for $0 \leq \ell \leq r$. Thus, (9.5) follows directly from Theorem 9.3. ◇

To approximate $\mathbf{rn_k}(n, r)$ for one iteration, we use directly the results for the expected number of r-nodes, given in [FO89], since $\mathbf{rn_1}(n, r) = \mathbf{rn}(n, k) \sim \frac{n}{r!e}$. We can see that

already for $k = 1$, a uniform initial distribution, and n large enough, there is a non negligible difference between our estimate (9.2)

$$\mathbf{H_1} \sim \log_2(n) - e^{-1} \sum_{r=1}^{n} \frac{1}{(r-1)!} \log_2(r) \approx \log_2(n) - 0.8272$$

and the upper bound (9.1)

$$\mathbf{H_1} \leq \log_2(n) + \log_2(1 - e^{-1}) \approx \log_2(n) - 0.6617 \; .$$

For more than one iteration we need to define a new parameter.

Theorem 9.5 *For $n \to \infty$ we can give the following asymptotic value*

$$\mathbf{rn_k}(n, r) \sim n \; c_k(r) \tag{9.6}$$

of the expected number of points in the functional graph which are reached by r points after k iterations, where

$$c_k(r) = \begin{cases} \frac{1}{r!e} & \text{for } k = 1 \\ D(k, r, 1) f_1(k) \frac{1}{e} & \text{for } k > 1 \end{cases}$$

$$D(k, r, m) = \begin{cases} 1 & \text{for } r = 0 \\ 0 & \text{for } 0 < r < m \\ \sum_{u=0}^{\lfloor r/m \rfloor} \frac{c_{k-1}(m)^u}{u!} D(k, r - mu, m + 1) & \text{otherwise} \end{cases}$$

and

$$f_1(k) = \begin{cases} 1 & \text{for } k = 1 \\ e^{e^{-1} f_1(k-1)} & \text{for } k > 1 \; . \end{cases}$$

Proof.

The concept of this proof is the same as in Section 8.1. We see the functional graph as a combinatorial structure. We are going to build the generating function corresponding to this structure where we mark the desired parameter. By means of the singularity analysis of the generating function we obtain the asymptotic value of this parameter. The difficulty is to mark the right property in the generation function.

Let $\mathcal{F} = \bigcup_{n \geq 1} \mathcal{F}_n$ be the set of all functions which map a finite set into itself. For a specific $\varphi \in \mathcal{F}$, we denote by $|\varphi|$ the size n of the finite set. We are interested in the parameter

$$\xi_{r,k}(\varphi) = \mathrm{rn}_k^{\varphi}(r)$$

which gives the number of nodes in the functional graph of φ which have a k-th iterate preimage of size r. We now have to define the bivariate generating function $F_{r,k}(z, u)$ of the pair $\mathcal{F}, \xi_{r,k}$. We will then apply Theorem 8.6 and singularity analysis to obtain the asymptotic mean of $\xi_{r,k}$.

To define the function $F_{r,k}(z, u)$, we start by a tree. A node in a tree which is reached by r nodes after k iterations can be described by the concatenation of three elements:

1. A *node.*
2. A *SET* of *trees* where each tree has a depth smaller than $k-1$.
3. A *concatenation* of j *trees* where the order of the concatenation is not important and where $1 \leq j \leq r$. Each of these trees has a depth larger or equal to $k-1$ and their roots are reached by respectively $i_1, \ldots, i_j$ nodes after $k-1$ iterations such that $i_1 + \cdots + i_j = r$.

In Figure 9.4, these three elements are marked for the node A. To write the corresponding

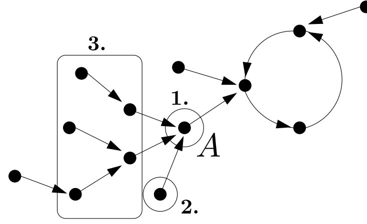


Figure 9.4: Example of the structure explained in 1.-3. for the node A.

bivariate generating function we need three more notations:

- The generating function of a set of trees of depth smaller than $k-1$, as described in 2., is given by $f_1(z,k)$ where

$$f_1(z,k) = \begin{cases} 1 & \text{for } k = 1 \\ e^{z\, f_1(z,k-1)} & \text{for } k > 1 \end{cases} .$$

- We mean by $Par(r)$ the integer partition of r, *i.e.* the set of all possible sequences $[i_1, \ldots, i_j]$ for $1 \leq j \leq r$ such that $1 \leq i_1 \leq \cdots \leq i_j \leq r$ and $i_1 + \cdots + i_j = r$. For example, for $r = 4$ we have $Par(4) = \{[1,1,1,1],[1,1,2],[2,2],[1,3],[4]\}$.

- Since the order of the concatenation in 3. is not important, we need a correction term $f_2([i_1, \ldots, i_j])$. If there are some $i_{x_1}, \ldots, i_{x_\ell}$ with $1 \leq x_1 < \cdots < x_\ell \leq j$ and $i_{x_1} = \cdots = i_{x_\ell}$ we have to multiply by a factor $1/\ell!$ to compensate this repeated appearance, e.g. $f_2([1,1,1,1,2,2,3]) = \frac{1}{4!2!1!}$.

Let $T_{r,k}(z,u)$ be the bivariate generation function of an arbitrary tree. We define by $c_k(z,r)$ a variable such that

$$c_k(z,r) T_{r,k}(z,u)^r$$

is the generating function of a tree where the root is reached by r nodes after k iterations. For $k = 1$, such a tree has r children, where each child is again a tree. In terms of generating functions, this structure can be represented by $z\frac{T_{r,k}(z,u)^r}{r!}$. Thus, we get

$$c_1(r,z) = \frac{z}{r!} .$$

For $k > 1$ we can use the structure given in 1.-3. and our notations to write:

$$\begin{aligned} & c_k(z,r) \\ = \;& \tfrac{1}{T_{r,k}(z,u)^r} \overbrace{z}^{1.} \overbrace{f_1(z,k)}^{2.} \\ & \overbrace{\sum_{[i_1,\dots,i_j]\in Par(r)} \left[c_{k-1}(i_1,z)T_{r,k}(z,u)^{i_1}\right] \cdots \left[c_{k-1}(i_j,z)T_{r,k}(z,u)^{i_j}\right] f_2([i_1,\dots,i_j])}^{3.} \\ = \;& z f_1(z,k) \textstyle\sum_{[i_1,\dots,i_j]\in Par(r)} c_{k-1}(i_1,z)\cdots c_{k-1}(i_j,z) f_2([i_1,\dots,i_j]) \;. \end{aligned}$$

In total we get

$$c_k(z,r) = \begin{cases} \text{for } k = 1: \; z/r! \\ \\ \text{for } k > 1: \\ z\, f_1(z,k) \displaystyle\sum_{[i_1,\dots,i_j]\in Par(r)} c_{k-1}(i_1,z) \;\cdots\; c_{k-1}(i_j,z)\; f_2([i_1,\dots,i_j]) \end{cases} \tag{9.7}$$

We can now write the generation function of a tree where we mark with the variable u the nodes which are reached by r other nodes after k iterations:

$$T_{r,k}(z,u) = z e^{T_{r,k}(z,u)} + (u-1) T_{r,k}(z,u)^r c_k(z,r) \;.$$

The first part describes a tree as a node concatenated with a set of trees. The second part correspond to our desired parameter. By applying the properties the structure in (8.1) we get the bivariate generating function for a general functional graph

$$F_{r,k}(z,u) = \frac{1}{1 - T_{r,k}(z,u)} \;.$$

In the next step, we apply Theorem 8.6. For this, we will use that for $u = 1$ we get the general generating function of a tree $T_{r,k}(1,z) = T(z) = ze^{T(z)}$.

$$\begin{aligned} \partial_u F_{r,k}(z,u)\big|_{u=1} &= \frac{1}{(1-T_{r,k}(z,u))^2}\, \partial_u T_{r,k}(z,u)\Big|_{u=1} \\ &= \frac{1}{(1-T_{r,k}(z,u))^2}\, \frac{T_{r,k}(z,u)^r c_k(z,r)}{1 - ze^{T_{r,k}(z,u)} - (u-1) r T_{r,k}(z,u)^{r-1} c_k(z,r)}\Bigg|_{u=1} \\ &= \frac{1}{(1-T(z))^2}\, \frac{T(z)^r c_k(z,r)}{1 - ze^{T(z)}} \\ &= \frac{T(z)^r c_k(z,r)}{(1-T(z))^3} \;. \end{aligned} \tag{9.8}$$

We cannot get directly the coefficients of the generating function (9.8), thus, we apply a singularity analysis. For this, we need the singular expansion (8.8) for the tree function $T(z)$. The dominant factor for the singularity analysis is $T(z) \sim 1 - \sqrt{2}\sqrt{1-ez}$ for $z \to e^{-1}$. We use that $F_{r,k}(z,1) = F(z)$ is the generating function for the set $\mathcal{F}$, thus $[z^n]F_{r,k}(z,1) = \frac{n^n}{n!}$. When applying (8.5), (9.8), and $\Gamma(3/2) = \sqrt{\pi}/2$ we can write:

$$\begin{aligned}
\mathbf{E}_{\mathcal{F}_n}(\xi) &= \frac{n!}{n^n}[z^n]\frac{T(z)^r c_k(z,r)}{(1-T(z))^3} , \\
&\overset{z\to e^{-1}}{\sim} \frac{n!}{n^n}[z^n]\frac{\left(1-\sqrt{2}\sqrt{1-ez}\right)^r c_k(e^{-1},r)}{(\sqrt{2}\sqrt{1-ez})^3} , \\
&\overset{n\to\infty}{\sim} \frac{n!}{n^n}\frac{c_k(e^{-1},r)}{2^{3/2}}e^n\frac{n^{3/2}}{n\sqrt{\pi}/2} , \\
&\overset{n\to\infty}{\sim} n\, c_k(e^{-1},r) .
\end{aligned}$$

Remark 9.6 *In the construction of our generating function we only count the nodes in the* tree *which are reached by r points after k iterations (* e.g. *node A in Figure 9.3). We ignore the nodes on the cycle (* e.g. *node B in Figure 9.3). However, the average proportion of the number of cycle points in comparison to the image size after k iterations is*

$$\frac{\mathbf{cp}(n)}{\mathbf{ip}(n,k)} \sim \frac{\sqrt{\pi n/2}}{n(1-\tau_k)} .$$

For a fixed k and $n \to \infty$ it is clear that this proportion gets negligible.

Thus, we can write

$$\mathbf{rn_k}(n,r) \sim n\, c_k(e^{-1},r) .$$

The computation of $c_k(e^{-1},r)$ as defined in (9.7) is not very practical. In this paragraph, we will show that we can do it more efficiently using dynamic programming. For simplicity we write in the following $c_k(e^{-1},r) = c_k(r)$ and $f_1(e^{-1},k) = f_1(k)$. We define the new value $D(k,r,m)$ by

$$D(k,r,m) = \sum_{[i_1,\ldots,i_j]\in Par_{\geq m}(r)} c_{k-1}(i_1)\cdots c_{k-1}(i_j) f_2([i_1,\ldots,i_j])$$

where $Par_{\geq m}(r)$ is the set of all partitions of the integer r such that for each $[i_1,\ldots,i_j] \in Par_{\geq m}(r)$ must hold that $i_\ell \geq m$ for all $1 \leq \ell \leq j$. Using this, we can give the recursive definition of $D(k,r,m)$ and $c_k(r)$ as described in this theorem. ◇

Most of the time we will compute $c_k(r)$ in a floating point arithmetic with fixed precision. The following proposition gives the number of operations needed. It says nothing about the precision of the final result.

Proposition 9.7 *Let as fix R and K. In a floating point arithmetic we can compute of $c_k(r)$, as described in Theorem 9.5, for all $r \leq R$ and $k \leq K$ in a time complexity of $\mathcal{O}(KR^2 \ln(R))$.*

Proof.
We use dynamic programming to compute $c_k(r)$.

The computation of $f_1(k)$ can be done once for all $k \leq K$ and then be stored. Thus, it has a time and space complexity of $\mathcal{O}(K)$. For $k = 1$, if we start with $r = 1$ we can compute $c_1(r)$ for all $r \leq R$ in R steps. The same is true for $1 < k \leq K$ if we already know $D(k, r, 1)$ and $f_1(k)$.

The most time consuming factor is the computation of $D(k, r, m)$. For a given k', let us assume that we have already computed all $c_{k'-1}(r)$ for $1 \leq r \leq R$. In the computation of $D(k, r, m)$ we will go for r from 1 to R, and for m from r to 1. This means that

- For a given r' we already know all $D(k', r, m)$ with $r < r'$.
- For a fixed r' and m' we already know all $D(k', r', m)$ with $m > m'$.
- To compute

  $$\sum_{u=0}^{\lfloor r/m \rfloor} \frac{c_{k-1}(m)^u}{u!} D(k, r - mu, m + 1)$$

  we need $\lfloor r/m \rfloor$ steps.

Thus in total, for each $1 < k \leq K$ we need

$$\sum_{r=1}^{R} \sum_{m=1}^{r} \left\lfloor \frac{r}{m} \right\rfloor$$

steps to compute all $D(k, r, m)$. By using that

$$\sum_{m=1}^{r} \frac{1}{m} = \ln(r) + C + \mathcal{O}\left(\frac{1}{r}\right)$$

where $C = 0.5772\ldots$ is the Euler constant, and

$$\begin{aligned} \sum_{r=1}^{R} r \ln(r) &\leq \ln(R) \sum_{r=1}^{R} r \\ &= \ln(R) \frac{R(R+1)}{2} \end{aligned}$$

we obtain the final time complexity of $\mathcal{O}(KR^2 \ln(R))$. $\diamond$

Let us go back to the expected entropy in (9.2). By using (9.6) we can write for $n \to \infty$

$$\mathbf{H_k} \sim \log_2(n) - \underbrace{\sum_{r=1}^{R} c_k(r)\, r\, \log_2(r)}_{(a)} - \underbrace{\sum_{r=R+1}^{n} c_k(r)\, r\, \log_2(r)}_{(b)}\,, \tag{9.9}$$

where (a) represents an estimation of the entropy loss which does not depend on n and (b) is an error term. In Figure 9.5, we see that the value $c_k(r)\ r\ \log_2(r)$ decreases fast

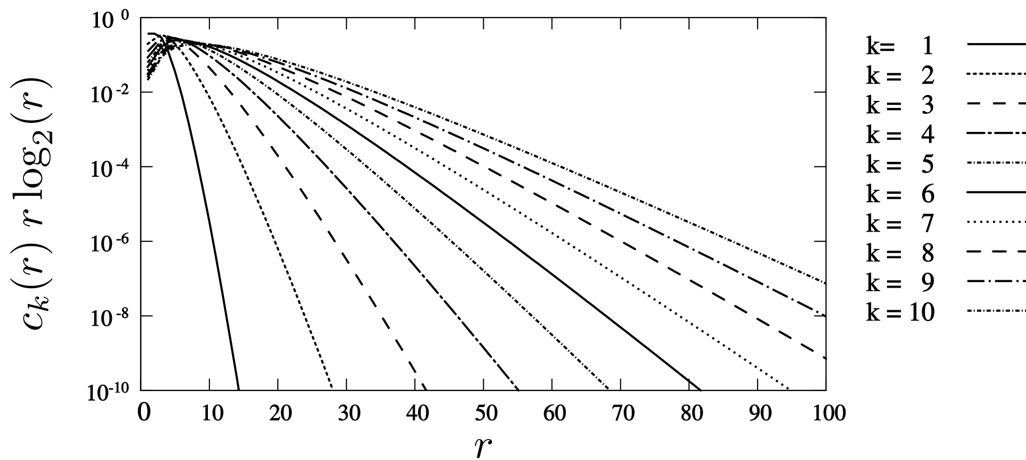


Figure 9.5: The course of $c_k(r)\ r\ \log_2(r)$ for different values of k and r.

with growing r. However, for larger k this decrease becomes slower. If we want (b) to be negligible for larger k we also need a larger value for R. Unfortunately, we could not find a closed formula of the function $c_k(r)\ r\ \log_2(r)$, so we restrict ourselves to the study of the graph. In Table 9.1, we compare our entropy estimator

$$H_k(R) = \log_2(n) - \sum_{r=1}^{R} c_k(r)\ r\ \log_2(r) \tag{9.10}$$

with the estimated lower bound of the loss given by the expected number of image points (9.1) and the empirical results from the experiment presented in Figure 9.2. From (9.6) and (9.9) we know that

$$\mathbf{H_k} \sim H_k(R)$$

for $n \to \infty$ and $R \to n$. We can see that for small k, in the order of a few hundred, we reach a much better approximation than the upper bound (9.1). For example, for most of the modern stream ciphers, the number of iterations for a key/IV–setup is in this order of magnitude. However, for increasing values of k we also need bigger values of R and, thus, this method gets computationally expensive. For $k = 100$ and $R = 1000$ the result of our estimate is about 0.02 larger than the empirical data. The fact that our estimate is larger shows that it is not due to the choice of R (it does not change a lot if we take $R = 2000$) but to the fact that our $n = 2^{16}$ is relatively small and, thus, the proportion of cycle points which is about

$$\frac{\sqrt{\pi n/2}}{n(1-\tau_k)} \approx 0.253$$

is not negligible.

k		1	2	3	⋯	10	⋯	50	⋯	100
empirical data, $n = 2^{16}$		0.8273	1.3458	1.7254	⋯	3.1130	⋯	5.2937	⋯	6.2529
image points (9.1)		0.6617	1.0938	1.4186	⋯	2.6599	⋯	4.7312	⋯	5.6913
$H_k(R)$, **(9.10)**	$R = 50$	0.8272	1.3457	1.7254	⋯	3.1084	⋯	2.6894	⋯	1.2524
	$R = 200$	0.8272	1.3457	1.7254	⋯	3.1129	⋯	5.2661	⋯	5.5172
	$R = 1000$	0.8272	1.3457	1.7254	⋯	3.1129	⋯	5.2918	⋯	6.2729

Table 9.1: Comparison of different methods to estimate the entropy loss.

In this section, we presented a new entropy estimator. We could show that if the number of iterations is not too big, it is much more precise than the upper bound given by the image size. In addition, the same method can be used for any arbitrary initial distribution.

9.3 Collision Attacks

In the previous section, we studied the loss of entropy in the inner state of our stream cipher model. In this section, we examine if it is possible to exploit this loss for a generic attack on our model. Hong and Kim present in [HK05] two attacks on the Mickey stream cipher [BD05, BD08], based on the entropy loss in the state. This stream cipher has a fixed update function; however Hong and Kim state, due to empirical results, that the update function behaves almost like a random function with regard to the expected entropy loss and the expected number of image points. Thus, these attacks are directly applicable on our model. We will give a detailed complexity analysis of these attacks and will show that in the case of a real random function they are less efficient than what one might assume from the argumentation of Hong and Kim.

Let us take two different initial states S_0 and S'_0 and apply the same function iteratively onto both of them. We speak about a collision if there exist k and k' such that $S_k = S_{k'}$, for $k \neq k'$, or $S_k = S'_{k'}$ for any arbitrary pair k, k'. The idea of Hong and Kim was that a reduced entropy leads to an increased probability of a collision. Once we have found a collision, we know that the subsequent output streams are identical. Due to the birthday paradox, we assume that with an entropy of m-bits we reach a collision, with high probability, by choosing $2^{\frac{m}{2}}$ different states.

The principle of the attacks is that we start from m different, randomly chosen, initial states $X_1, X_2, \ldots, X_m$ and that we apply iteratively the same update function k times on each of them. In the end, we search for a collision and hope that our costs are less than for a search directly in the initial states. We will study the two attacks proposed in [HK05] as well as a variant using distinguished points. For each of these attacks we provide a detailed complexity analysis where we examine the query and the space complexity as well as the number of necessary initial states to achieve a successful attack with high probability. We mean by the *query complexity* the number of all states produced by the cipher during the attack which is equivalent to the number of times the updated function is applied. We

denote by the *space complexity* the number of states we have to store, such that we can search for a collision within them. Each time we compare the results to the attempt of finding a collision directly within the initial states which has a space and query complexity of $\sim \sqrt{n}$.

All these attacks consider only the probability of finding a collision in a set of states. This is not equivalent to an attack where we have $m-1$ initial states prepared and we want the probability that if we take a new initial state, it will be one of the already stored. In such a scenario, the birthday paradox does not apply. We also never consider how many output bits we would really need to store and to recognize a collision, since this value depends on the specific filter function used. In the example of MICKEY, Hong and Kim states that they need about 2^8 bits to store a state.

9.3.1 States after k Iterations

The first attack of Hong and Kim takes randomly m different initial states $X_1, X_2, \ldots, X_m$ and applies k times the same instance of Φ on each of them. Then, we search for a collision in the m resulting states (Figure 9.6). Using the upper bound (9.1) we know that

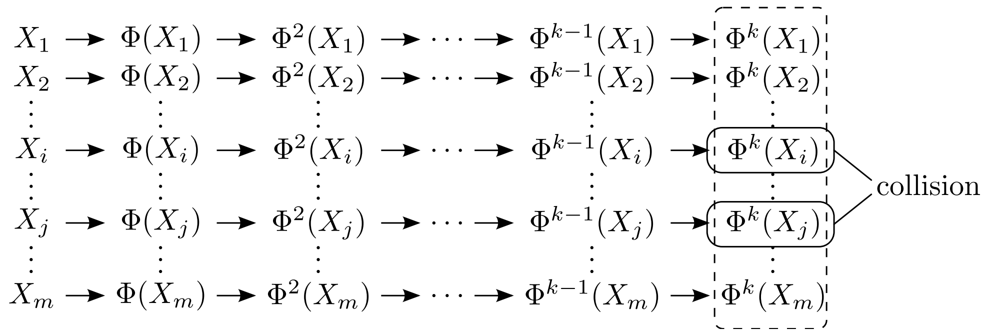


Figure 9.6: Collision search in the last column.

the average entropy after k iterations is less than $\log_2(n) + \log_2(1-\tau_k)$. Hong and Kim conjecture, based on experimental results, that this is about the same as $\log_2(n) - \log_2(k) + 1$. Thus, with high probability we find a collision if $m > 2^{(\log_2(n)-\log_2(k)+1)/2} = \sqrt{2n/k}$.

This attack stores only the last value of the iterations and searches for a collision within this set. This leads to a space complexity of $m \sim \sqrt{2n/k}$ for large enough k. However, Hong and Kim did not mention that we have to apply k times Φ on each of the chosen initial states, which results in a query complexity of $mk \sim \sqrt{2kn}$. This means that we increase the query complexity by the same factor as we decrease the space complexity and the number of initial states.

The question remains, if there exist any circumstances under which we can use this approach without increasing the query complexity. The answer is yes, if the stream cipher uses a family of update functions which lose more than $2\log_2(k)$ bits of entropy after k iterations. Such a characteristic would imply that we do not use random functions to

update our state, since they have different properties as we have seen before. However, the principle of the attack stays the same.

9.3.2 Including Intermediate States

The second attack in [HK05] is equivalent to applying $2k-1$ times the same instance of Φ on m different initial states and searching for a collision in all intermediate states from the k-th up to the $(2k-1)$-th iteration (Figure 9.7). Since after $k-1$ iterations we have

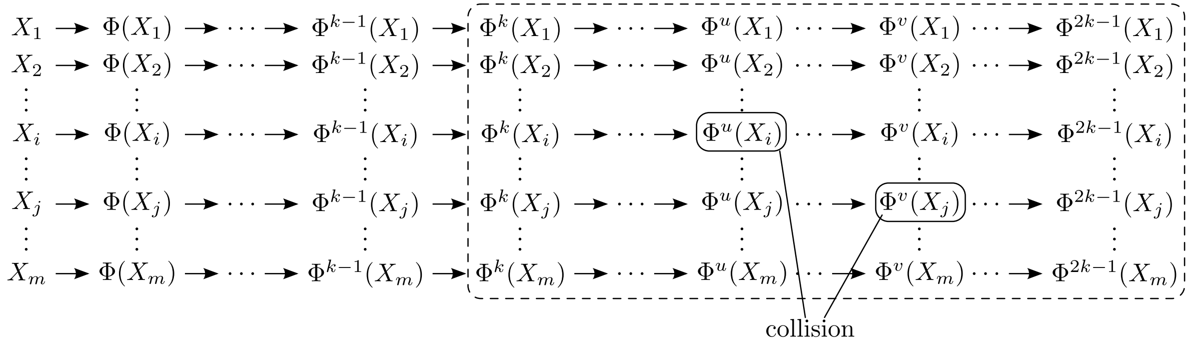


Figure 9.7: Collision search in the second half of the iterations.

about $\log_2(n) - \log_2(k) + 1$ bits of entropy, Hong and Kim assume that we need a m such that $mk \sim \sqrt{n/k}$. They state that this result would be a bit too optimistic since collisions within a row normally do not appear in a practical stream cipher. However, they claim that this approach still represents a realistic threat for the MICKEY stream cipher. We will show that for a random function, contrary to their conjecture, this attack has about the same complexities as a direct collision search in the initial states.

Let us take all the $2km$ intermediate states for the $2k-1$ iterations. Let $Pr[A]$ define the probability that there is no collision in the all the $2km$ intermediate states. If there is no collision in this set then there is also no collision in the km states considered by the attack. Thus, the probability of a successful attack is even smaller than $1 - Pr[A]$. By means of only counting arguments we can show the following proposition.

Proposition 9.8 *The probability of no collision in the $2km$ intermediate states is*

$$Pr[A] = \frac{n(n-1)\cdots(n-2km+1)}{n^{2km}} , \tag{9.11}$$

where the probability is taken over all functions $\varphi \in \mathcal{F}_n$ and all possible choices of m initial states $X_1, X_2, \ldots, X_m$.

Proof.
Let $Pr[I]$ be the probability of no collision in the m initial states. We can see directly that

$$\begin{aligned} Pr[A] &= Pr[A \cap I] \\ &= Pr[A|I]\, Pr[I] . \end{aligned}$$

Let us assume that we have chosen m different initial states. This happens with a probability of

$$Pr[I] = \frac{\binom{n}{m} m!}{n^m} . \tag{9.12}$$

In this case we have:

- n^n different instances $\varphi \in \mathcal{F}_n$ of our random functions, where each of them creates
- $\binom{n}{m} m!$ different tables. Each table can be produced more than once. There exist
- $n\ (n-1) \ldots (n-2km+1)$ different tables that contain no collisions. Each of them can be generated by
- $n^{n-(2k-1)m}$ different functions, since a table determines already $(2k-1)m$ positions of φ.

Thus, we get the probability

$$Pr[A|I] = \frac{n\ (n-1) \ldots (n-2km+1)\ n^{n-(2k-1)m}}{n^n \binom{n}{m} m!} \tag{9.13}$$

for $m > 0$ and $2km \leq n$. By combining (9.12) and (9.13) we can conclude our proof. $\diamond$

The probability of $Pr[A]$ given in (9.11) is exactly the probability of no collision in $2km$ random points, which means that we need at least an m such that $2km \sim \sqrt{n}$. This leads to a query complexity of $\sim \sqrt{n}$ and a space complexity of $\sim \sqrt{n}/2$.

9.3.3 Improvement with Distinguished Points

Distinguished points are a well known technique in cryptography to reduce the number of time expensive memory accesses. The idea was first mention by Rivest (see [Den82, p. 100]) and was applied in several practical attacks, like [QD88, QD89, BS00]. Using this technique we can reduce the space complexity in the second attack; however the query complexity stays the same.

We mean by *distinguished points* (DPs) a subset of the set $\{1, 2, \ldots, n\}$ which is distinguished by a certain property, *e.g.* by a specific number of 0's in the most significant bits. The idea is to compare only the DPs and not all intermediate values. In our new variant of the second attack, we iterate Φ in each row up to the moment where we reach a DP. In this case we stop and store the DP. If we do not reach a DP after k_{MAX} iterations we stop as well but we store nothing. If there was a collision in any of the states in the rows where we reached a DP, the subsequent states would be the same and we would stop with the same DP. Thus, it is sufficient to search for a collision in the final DPs (see Figure 9.8).

Let d be the number of distinguished points in the set $\{1, 2, \ldots, n\}$. We assume that the ratio $c = \frac{d}{n} \leq 1$ is large enough that with a very high probability we reach a DP before the end of the cycle in the functional graph. This means that the average number

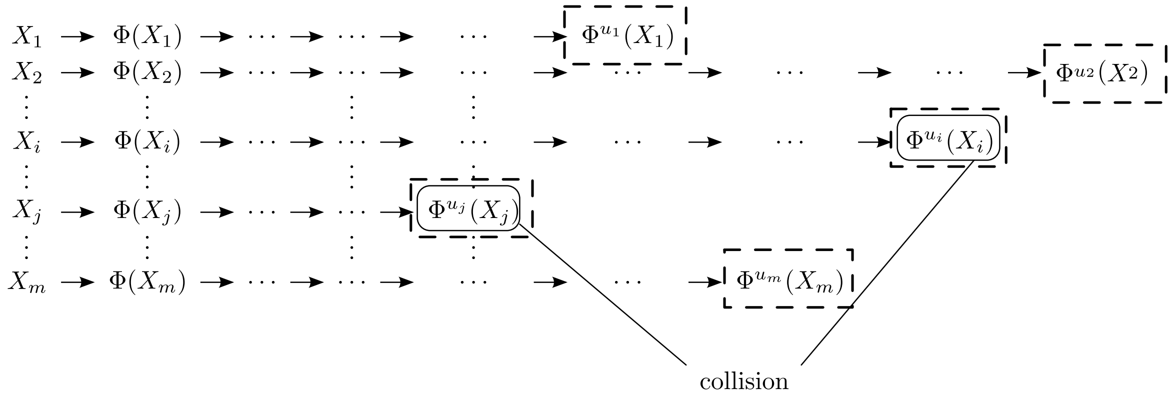


Figure 9.8: The attack using distinguished points.

of iterations before arriving at a DP is much smaller than the expected length of the tail and the cycle together (which would be about $\sqrt{\frac{\pi n}{2}}$ due to [FO89]). We assume that in this case, the average length of each row would be in the range of $1/c$ as in the case of random points. We also suppose that that we need about $m/c \sim \sqrt{n}$ query points to find a collision, as in the previous case. This leads to a query complexity of $\sim \sqrt{n}$ and a space complexity of only $\sim c\sqrt{n}$. Empirical results for example for $n = 2^{20}$, $-6 \leq \log_2(c) \leq -1$ and $k_{MAX} = \sqrt{n}$ confirm our assumptions, as we can see in Figure 9.9.

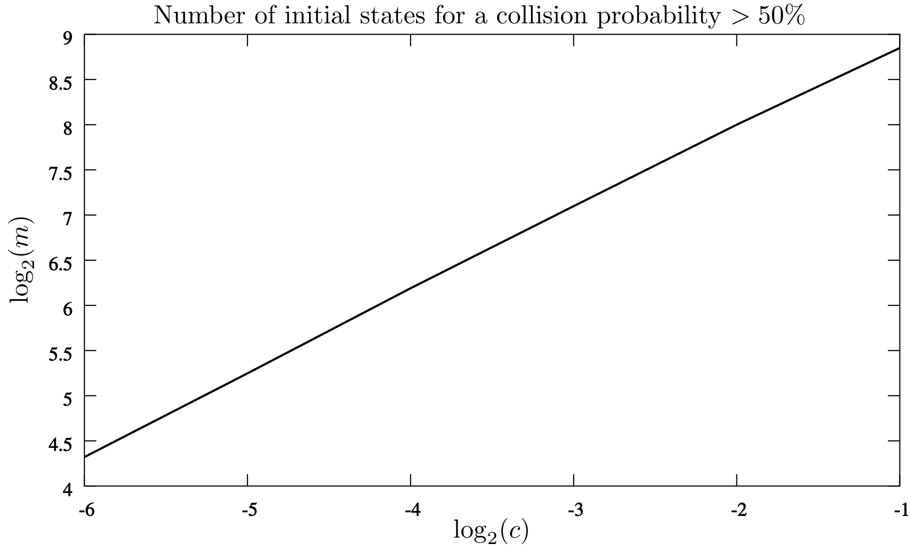


Figure 9.9: Empirical results for an attack with DP and $n = 2^{20}$.

A summary of the complexities of all attacks can be found in Table 9.2, where we marked by *(new)* the results that where not yet mentioned by Hong and Kim. In the case where

we consider only the states after k iterations, we have to substantially increase the query complexity to gain in the space complexity and the number of initial states. We were able to show that even when we consider all intermediate states, the query complexity has a magnitude of $\sqrt{n}$. The variant using the distinguished points allows to reduce the space complexity by leaving the other complexities constant.

attack	**# initial states**	**space complexity**	**query complexity**
after k iterations, 9.3.1	$\sim \sqrt{2n/k}$	$\sim \sqrt{2n/k}$	$\sim \sqrt{2kn}$ *(new)*
with interm. states, 9.3.2	$\sim \sqrt{n}/2k$ *(new)*	$\sim \sqrt{n}/2$ *(new)*	$\sim \sqrt{n}$ *(new)*
with DPs, 9.3.3	$\sim c\sqrt{n}$ *(new)*	$\sim c\sqrt{n}$ *(new)*	$\sim \sqrt{n}$ *(new)*

Table 9.2: Complexities of attacks.

9.4 Conclusion

In this chapter, we studied a stream cipher model which uses a random update function. We have introduced a new method of estimating the state entropy in this model. This estimator is based on the number of values that produce the same value after k iterations. Its computation is expensive for large numbers of iterations; however, for a value of k up to a few hundred, it is much more precise than the upper bound given by the number of image points.

In this model, we have also examined the two collision attacks proposed in [HK05] which are based on the entropy loss in the state. We pointed out that the first attack improves the space complexity at the cost of significantly increasing the query complexity. We proved that the complexity of the second attack is of the same magnitude as a collision search directly in the starting values. In addition, we discussed a new variant of this attack, using distinguished points, which reduces the space complexity but leaves the query complexity constant.

The use of a random function in a stream cipher introduces the problem of entropy loss. However, the studied attacks based on this weakness are less effective than expected. Thus, the argument alone that a stream cipher uses a random function is not enough to threaten it due to a collision attack based on the entropy loss.

Part V

FCSRs

Chapter 10

Introduction to FCSRs

Feedback with Carry Shift Registers (FCSRs) are non linear shift registers. They are constructed similarly to Linear Feedback Shift Registers (LFSRs), however instead of XOR, *i.e.* an addition modulo 2, they use integer addition. FCSRs were first presented by Klapper an Goresky in [KG93]. The two authors developed their idea further in several conferences [KG93, GK94, KG95b, KG95a] and gave a detailed summary and introduction into the area of FCSR in their article [KG97].

Independently, Marsaglia [Mar94] and Couture and L'Ecuyer [CL97] developed the multiple-with-carry (MWC) random number generator as an enhancement of the add-with-carry (AWC) generator [MZ91, CL94]. The MWC generator is based on the same principles as the FCSR.

The original FCSR automaton corresponds to the Fibonacci architecture of an LFSR. In 2002, Klapper and Goresky [GK02] developed as well a Galois architecture for FCSRs. An example for a Fibonacci and Galois architecture can be seen in Figure 10.3 and Figure 10.4, respectively.

The big advantages of FCSRs are that they are non-linear, structured enough to allow a detailed analysis and fast enough to be interesting in modern implementations. The properties of LFSRs are well known [Gol81, McE87]. However, their linear update function makes them vulnerable to algebraic attacks [CM03]. For this reason, Nonlinear Feedback Shift Registers (NLFSRs) [Gol81] seem to be an interesting alternative, yet, in the general case of NLFSRs, it is hard to give a detailed analysis. In the case of FCSRs, we can build on the strong theory of 2-adic numbers. In addition, the simple structure allows an efficient implementation in hardware and software [GB07, LR08].

The FCSRs found a concrete application in the F-FCSR stream cipher [AB05b, ABL08]. This cipher was a candidate of the European eSTREAM project [eST] in the Hardware Profile and was first chosen as one of the finalists. However, after an attack from Hell and Johansson [HJ08] the F-FCSR was withdrawn from the final eSTREAM portfolio. Arnault, Berger, Minier and Pousse working on a new architecture for FCSRs which combines the Galois and the Fibonacci setup. This allows to resist attacks which exploit the

specific weaknesses of the particular architectures[1]. The FCSRs remain an interesting tool in the area of cryptography.

FCSRs are a young research area and every year several new articles are published on this topic. In addition to the original FCSRs, there exist several extensions like FCSRs over finite fields [Kla94], the *d*-FCSRS [Kla04, GK04] or the Algebraic Feedback Shift Registers (AFSRs) [KX99, KX04]. In Section 10.4 we will address these extensions shortly. However, if not otherwise mentioned, we will consider binary FCSRs in Fibonacci or Galois setup.

FCSRs are analogous to LFSRs. Thus, we will first give an overview of the most important properties of LFSRs in Section 10.1 to show afterwards, in Section 10.2, the parallels and differences to FCSRs. In both cases, we will restrict our discussion to the binary case, *e.g.* we consider an output sequence $\mathcal{S} = (s_0, s_1, s_2, \ldots)$ where every $s_i \in \{0, 1\}$ for $i \geq 0$. Subsequently, we will discuss some applications of FCSRs in Section 10.3 and the most interesting extension in Section 10.4. My contributions concerning FCSRs will be described in detail in Chapters 11 and 12.

10.1 Characteristics of LFSRs

All the properties of LFSRs that we are going to state are well known and explained in nearly every book concerning cryptography [CDR98, MvOV96, Sch95]. A detailed description can be found in the monographs of Golomb and McEliece [Gol81, McE87].

Definition 10.1 *An* LFSR *is an automaton which generates* linear recursive sequences*, which means that the output sequence* $\mathcal{S} = (s_0, s_1, s_2, \ldots)$ *can be described in the following way in the finite field* $\mathbb{F}_2$*:*

$$s_i = q_1 s_{i-1} + q_2 s_{i-2} + \cdots + q_n s_{i-n} \in \mathbb{F}_2 \ , \tag{10.1}$$

for $i \geq n$ *and given values* $n \in \mathbb{N}$ *and* $q_1, q_2, \ldots, q_n \in \mathbb{F}_2$*.*

The addition in $\mathbb{F}_2$ can be realized by an XOR, which we will denote by $\oplus$. Thus, the relation in (10.1) is equivalent to the implementation in Figure 10.1 where the output is taken from cell a_0. Such an automata is called a *Fibonacci LFSR*. It is characterized by *a multiple inputs and single output feedback function*. We will see later that the same sequence can be produced by an automaton with *multiple feedback functions with a common input* as described in Figure 10.2. This setup is called *Galois LFSR*. The following definition is valid for the both variants.

Definition 10.2 *The* connection polynomial $q(X)$ *of an LFSR is defined by its feedback positions:*

$$q(X) = q_n X^n + q_{n-1} X^{n-1} + \cdots + q_1 X + 1 \in \mathbb{F}_2[X] \ . \tag{10.2}$$

The characteristic polynomial $Q(X) = X^n q(1/X)$ *is the reciprocal polynomial of* $q(X)$*:*

$$Q(X) = q_n + q_{n-1} X + \cdots + q_1 X^{n-1} + X^n \in \mathbb{F}_2[X] \ . \tag{10.3}$$

[1]Personnel communications with François Arnault, Thierry Berger, Marine Minier, and Benjamin Pousse.

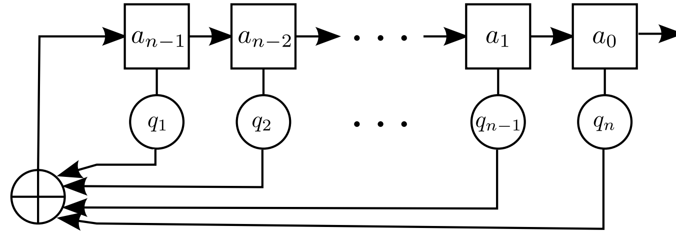


Figure 10.1: Fibonacci LFSR.

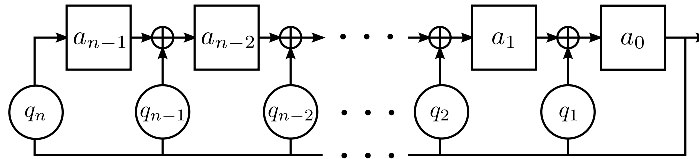


Figure 10.2: Galois LFSR.

The *generating function* or *formal power series* of the sequence $\mathcal{S}$ is defined by the element $s(X)$ of the ring $\mathbb{F}_2[[X]]$ of formal power series:

$$s(X) = \sum_{i=0}^{\infty} s_i X^i \ .$$

Theorem 10.3 ([Gol81]) *Let $\mathcal{S} = (s_0, s_2, s_2, \ldots)$ be a sequence satisfying the linear recurrence $s_i = q_1 s_{i-1} + q_2 s_{i-2} + \cdots + q_n s_{i-n}$ for $i \geq n$. Then, its generating function is equal to a quotient of two polynomials*

$$s(X) = \frac{h(X)}{q(X)} \in \mathbb{F}_2[[X]] \ , \tag{10.4}$$

where the denominator is the connection polynomial $q(X) = 1 + \sum_{i=i}^{n} q_i X^i$ of an LFSR generating the sequence. If $q_n = 1$, i.e. *$\mathcal{S}$ is strictly periodic, the degree of the nominator $h(X)$ is smaller than the degree n of $q(X)$.*

Let us define $q_0 = 1$ and assume an initial loading of $a_0, \ldots, a_{n-1}$. In the case of a Fibonacci LFSR, the polynomial $h(X)$ is given by

$$h(X) = \sum_{j=0}^{n-1} \sum_{i=0}^{j} q_i a_{j-i} X^j \ ,$$

whereas in the case of a Galois LFSR we have

$$h(X) = h(X) = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1} \ .$$

The same sequence can be generated by different automata and different connection polynomials. We say that $q(X)$ is *minimal* if $\gcd(h(X), q(X)) = 1$.

Definition 10.4 *The* linear complexity *or* linear span *of a sequence $\mathcal{S} = (s_0, s_1, s_2, \ldots)$ is the smallest LFSR, which generates the sequence,* i.e. *the smallest n such that there exists $q_1, q_2, \ldots q_n$ which fulfill the relation in (10.1). If $\mathcal{S}$ is strictly periodic,* i.e. $q_n = 1$*, this is equivalent to the* degree *of the minimal polynomial $q(X)$ which correspond $s(X)$.*

We can use the Berlekamp-Massey algorithm [Mas69] to find the smallest LFSR, *i.e.* the minimal $q(X)$ which generates a given sequence $\mathcal{S}$. This algorithm is is based on Berlekamp's algorithm for BCH decoding [Ber68b, Ber68a] and was first applied on LFSRs by Massey in [Mas69]. The algorithm is optimal in two senses, it finds the smallest LFSR and needs the smallest number of output bits to do so, namely two times the linear complexity of $\mathcal{S}$.

If we bitwise combine two binary linear recurrent sequences $\mathcal{S} = (s_0, s_1, s_2, \ldots)$ and $U = (u_0, u_1, u_2, \ldots)$, the generating function of result has the form $v(X) = s(X) + u(X) \in \mathbb{F}_2[[X]]$. Then, the linear complexity of the combination V can not be bigger than the sum of the linear complexities of $\mathcal{S}$ and U.

In the general case, we can give the following two statements about the period of a linear recurrent sequence:

Theorem 10.5 ([Gol81])

1. *The succession of states in the LFSR is eventually periodic, with period $T \leq 2^n$, where n is the size of the LFSR.*
2. *Let $\mathcal{S} = (s_0, s_1, s_2, \ldots)$ be a sequence generated by an LFSR. The* period *of $\mathcal{S}$ is the smallest positive integer T for which $1 - X^T$ is divisible by the connection polynomial $q(X)$.*

In cryptography, we are often interested in sequences with high periods.

Definition 10.6 *An output sequence of an LFSR is called an* m–sequence *if it has the maximal period of $2^n - 1$. This is the case if and only if $q(X)$ is a* primitive polynomial.

A last property that we want to mention, is that the ith output of an LFSR can be calculated not only by the recurrence equation, but also directly by means of the trace function.

Theorem 10.7 ([GK02]) *Let $q(X)$ be the connection polynomial of degree n of our LFSR which produces the sequence $\mathcal{S} = (s_0, s_1, s_2, \ldots)$. If $q(X)$ is* irreducible *and $\lambda \in \mathbb{F}_{2^n}$ is a root of $q(X)$, then*

$$s_i = Tr\left(A\lambda^i\right) \tag{10.5}$$

for all $i \geq 0$ and for some $A \in \mathbb{F}_{2^n}$, where A corresponds to the choice of the initial state of the LFSR and $Tr : \mathbb{F}_{2^n} \to \mathbb{F}_2$ denotes the trace function.

10.2 Characteristics of FCSRs

The first version of an FCSR presented by Klapper and Goresky [KG97] was in Fibonacci architecture, as presented in Figure 10.3. The difference to an LFSR is that the sum $\sum_{i=1}^{n} a_{n-i}q_i$ is now calculated over $\mathbb{Z}$ and not $\mathbb{F}_2$, and that we have to consider the additional memory m.

Definition 10.8 (Fibonacci FCSR) *A* Fibonacci FCSR *is an automaton which realizes the following state-update function:*

Let n be the length of the main register, $a_0, a_1, \ldots, a_{n-1} \in \{0,1\}$ represent the value of its cells and $q_1, q_2, \ldots, q_n \in \{0,1\}$ the feedback positions. The value $m \in \mathbb{Z}$ corresponds to the auxiliary memory. Then, the transition of the state from time t to time $t+1$ is given by:

$$\begin{array}{lcl} \sigma & = & m(t) + \sum_{i=1}^{n} a_{n-i}(t)q_i \;, \\ a_i(t+1) & = & a_{i+1} \quad \textit{for } 0 \leq i \leq n-2 \;, \\ a_{n-1}(t+1) & = & \sigma \bmod 2 \;, \\ m(t+1) & = & \sigma \operatorname{div} 2 \;, \end{array} \tag{10.6}$$

where div *denotes the integer division.*

The equations in (10.6) for $a_{n-1}(t)$ and $m(t)$ can be written together as:

$$2m(t+1) + a_{n-1}(t+1) = m(t) + \sum_{i=1}^{n} a_{n-i}(t)q_i \tag{10.7}$$

since it must hold that $a_{n-1} \in \{0,1\}$.

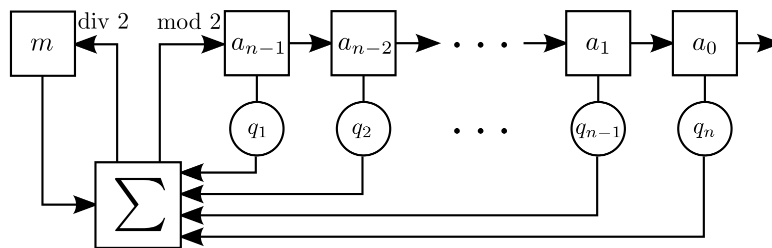


Figure 10.3: Fibonacci FCSR.

Remark 10.9 *In the case of a Fibonacci FCSR, we mean by the* state (A, m) *the content of the cells of the main register and the memory,* i.e. $A = (a_0, a_1, \ldots, a_{n-1})$*, and m. We will write respectively $A(t) = (a_0(t), a_1(t), \ldots, a_{n-1}(t))$, and $m(t)$ if we refer to the values at a specific time t. The* initial state *denotes the state at time $t = 0$.*

We say that the state is periodic *if the FCSR will eventually return to it, after some iterations of the transition function (10.6).*

In the following, we will often refer to the number of feedback positions. In the case of a Fibonacci FCSR, we will denote this number by:

$$\ell = \#\{q_i = 1 | 1 \leq i \leq n\} \; . \tag{10.8}$$

The initial value of m can be any integer, however, the size of m is bounded as soon as the FCSR reaches a *periodic state*.

Theorem 10.10 ([KG97]) *Let ℓ be the number of feedback positions,* i.e. *the number of i's such that $q_i = 1$. If an FCSR is in a periodic state, then the memory is in the range of $0 \leq m \leq \ell$, thus, we need no more than $\lfloor \log_2(\ell - 1) \rfloor + 1$ bits to store m.*

If the memory $m(t) \geq \ell$, then it will monotonically decrease and will arrive in the range $0 \leq m < \ell$ within $\lfloor \log_2 (m(t) - \ell) \rfloor + n$ steps. If the initial memory $m(t) < 0$, then it will monotonically increase and will arrive in the range $0 \leq m < \ell$ within $\lceil \log_2 (|m(t)|\rceil + n$ steps.

The Galois FCSR was introduced by Klapper and Goresky some years later in [GK02]. It has not one integer summation, but one at each feedback path. This yields to an advantage in the implementation, since the additions can be done in parallel. An example for a Galois FCSR can be seen in Figure 10.4. Each symbol $\boxplus$ represents an integer addition

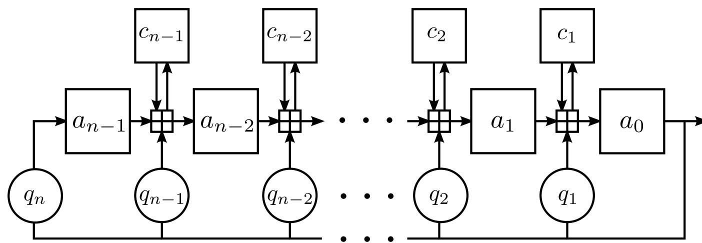


Figure 10.4: Galois FCSR.

for up to three bits, where the carry is stored in c_i for $1 \leq i \leq n-1$.

Definition 10.11 (Galois FCSR) *A Galois FCSR is an automaton which consists of a main register, and an integer addition with carry bit at each feedback position. Let n be the size of the main register with cells $a_0, a_1, \ldots, a_{n-1}$ and $a_i(t) \in \{0,1\}$ for all $0 \leq i \leq n-1$. The multiplication factors $q_1, q_2, \ldots q_n \in \{0,1\}$ determine the feedback positions, where $q_n = 1$, and the memory cells $c_1, c_2, \ldots c_{n-1}$ store the carry bits. The state transition from time t to time $t+1$ is done in the following way:*

for all $1 \leq i \leq n-1$

$$\begin{array}{rcl} \sigma_i & = & a_i(t) + c_i(t) + q_i a_0(t) \; , \\ a_{i-1}(t+1) & = & \sigma_i \bmod 2 \; , \\ c_i(t+1) & = & \sigma_i \operatorname{div} 2 \; , \end{array} \tag{10.9}$$

and

$$a_{n-1}(t+1) = a_0(t) \; .$$

For $1 \leq i \leq n-1$, we can sum up the equation (10.9) for $a_{i-1}(t+1)$ and $c_i(t+1)$, in the following way:

$$2c_i(t+1) + a_{i-1}(t+1) = a_i(t) + c_i(t) + q_i a_0(t) \ . \tag{10.10}$$

Remark 10.12 *In Figure 10.4, we display all c_i. However, in the following we will assume that $c_i = 0$ if $q_i = 0$. Thus, in practice, we need only $\ell = \#\{q_i = 1 | 1 \leq i \leq n-1\}$ carry bits. In the following, we will denote by ℓ the number of carry cells used in a Galois FCSR.*

The state *of a Galois FCSR consists of the cells of the* main register $A = (a_0, a_1, \ldots, a_{n-1})$ *and the cells of the* carry register $C = (c_1, c_2, \ldots, c_{n-1})$. *Let us define the following two integers*

$$a = \sum_{i=0}^{n-1} a_i 2^i \in \mathbb{N} \ , \tag{10.11}$$

and

$$c = \sum_{i=0}^{n-2} c_{i+1} 2^i \in \mathbb{N} \ , \tag{10.12}$$

which are equivalent representations of respectively the main and the carry register. Thus, the state can be represented by the integer pair (a, c) or directly by the values of the cell (A, C). The specific values at time t are given respectively by $A(t) = (a_0(t), a_1(t), \ldots a_{n-1}(t))$, $C(t) = (c_1(t), c_2(t), \ldots, c_{n-1}(t))$, and $(a(t), c(t))$.

Where the LFSRs are build on the theory of polynomials, the FCSRs are based on the theory of 2-adic numbers.

Definition 10.13 (2-adic numbers $\mathbb{Z}_2$) *Let $\mathbb{Z}_2$ denote the set of 2-adic numbers. An element $s \in \mathbb{Z}_2$ is given by its formal power series*

$$s = \sum_{i=0}^{\infty} s_i 2^i \ , \tag{10.13}$$

with $s_i \in \{0, 1\}$. This sum always converges in the 2-adic topology, where addition and multiplication are realized by transferring the carries to higher order terms, i.e. $2^i + 2^i = 2 \ 2^i = 2^{i+1}$. *Together with addition and multiplication as defined above, $\mathbb{Z}_2$ is a ring with additive identity 0 and multiplicative identity $1 = 1 \ 2^0$.*

Remark 10.14 *A number $s \in \mathbb{Z}_2$ is* positive *if there exists an N such that $s_i = 0$ for all $i > N$. The number -1 is given by $-1 = \sum_{i=0}^{\infty} 1 \ 2^i$. It is easy to verify that $1 + (-1) = 0$ in the ring of $\mathbb{Z}_2$. For a given element $s = 2^n \left(1 + \sum_{i=0}^{\infty} s_i 2^i\right)$ we can write its* negative *value in the following way*

$$-s = 2^n \left(1 + \sum_{i=0}^{\infty} \overline{s}_i\right) \ ,$$

where $\overline{s}_i$ defines the complement of s_i.

Every odd element $s \in \mathbb{Z}_2$, i.e. *where $s_0 = 1$, has a unique multiplicative inverse.*

The last remark can be used to give the following theorem.

Theorem 10.15 ([KG97]) *There is a one-to-one correspondence between rational numbers $s = h/q$ (where q is odd) and* eventually periodic binary *sequences $\mathcal{S} = (s_0, s_1, \ldots)$, which associates to each such rational number s the bit sequence $s_0, s_1, \ldots$ of its 2-adic expansion. The sequence $\mathcal{S}$ is* strictly periodic *if and only if $s \leq 0$ and $|s| < 1$.*

Let $\mathrm{ord}_q(2)$ define the order of 2 modulo q, *i.e.* the smallest k such that $1 = 2^k \bmod q$. An old result of Gauss follows from the previous theorem which is interesting for the case of FCSRs.

Corollary 10.16 ([Gau01, KG97]) *If h and q are relatively prime, $-q < h \leq 0$, and q is odd, then the period of the bit sequence for the 2-adic expansion of $s = h/q$ is $T = \mathrm{ord}_q(2)$.*

Remark 10.17 *Let us consider the rational number $s = h/q$ corresponding to an eventually periodic sequence. We want that $h/q \leq 0$ is negative, however, whether we choose q positive or negative is a matter of taste. In the following, we will always assume that q is positive.*

Now that we know the basic facts about 2-adic numbers, we will consider the case of Fibonacci and Galois FCSRs.

Definition 10.18 *For a given FCSR (Fibonacci or Galois), as described in Figures 10.3 and 10.4, let us define the* connection integer *as*

$$q = -1 + \sum_{i=1}^{n} q_i 2^i \in \mathbb{Z} , \tag{10.14}$$

where $q_n = 1$. We call n the degree *of the q.*

Theorem 10.19 ([KG97]) *Suppose an FCSR with connection integer q of degree n has initial loading $a_0, a_1, \ldots, a_{n-1}$ and initial memory m. Set*

$$h = m2^n - \sum_{j=0}^{n-1} \sum_{i=0}^{j} q_i a_{j-i} 2^j \in \mathbb{Z} \tag{10.15}$$

where $q_0 = -1$. Then the output sequence is the coefficient sequence of the 2-adic number $s = -h/q$. Conversely, if $\mathcal{S} = (s_0, s_1, \ldots)$ is an eventually periodic binary sequence, *let $s = -h/q$ be the corresponding 2-adic number. Then q is the connection integer of an FCSR which generates the sequence, and h determines the initial loading by (10.15). The sequence $\mathcal{S}$ is* strictly periodic *if and only if $h \geq 0$ and $h < q$. In this case, the memory must lie in the range $0 \leq m < wt(q+1)$, where wt denotes the Hamming weight of the 2-adic representation of $q+1$,* i.e. *the number of $1 \leq i \leq n$ such that $q_i = 1$.*

Theorem 10.20 ([GK02]) *Suppose an n-stage Galois-FCSR with connection integer q has initial loading with register and memory contents $a_0, a_1, \dots, a_{n-1}$ and $c_1, c_2, \dots, c_{n-1}$, respectively. Set*

$$h = a_0 + (a_1 + c_1)2 + \cdots + (a_{n-1} + c_{n-1})2^{n-1} . \tag{10.16}$$

Then the output sequence $s_0, s_1, s_2, \dots$ of the FCSR is the coefficient sequence for the 2-adic expansion of the rational number $s = -h/q$. Conversely, if $\mathcal{S} = (s_0, s_1, \dots)$ is any strictly periodic sequence, let $s = -h/q$ be the corresponding 2-adic number. Then q is the connection integer of a Galois FCSR which generates the sequence, and h determines the initial loading by (10.16).

If we compare (10.11) and (10.12) we see that (10.16) is equivalent to

$$h = a + 2c . \tag{10.17}$$

For a Galois FCSR we assume that $c_i = 0$ if $q_i = 0$, thus, the maximal value of h is bounded by $h \leq \sum_{i=0}^{n-1} 2^i + \sum_{i=1}^{n-1} q_i 2^i = q$ and reaches the bound exactly if all the bits in the FCSR are 1. Therefore, from Theorems 10.15 and 10.20 we know that the sequence $\mathcal{S} = (s_0, s_1, ...)$ generated by a Galois FCSR is *always strictly periodic*. However, this is not true for the state itself. There is often more than one possibility for a and c to create one h. This problem is discussed in more detail in Chapter 11. In Figure 10.5 we see the example of a functional graph of a Galois FCSR. In this graph, a vertex with the pair (a, c) represents a specific state and an arrow from state (a, c) to (a', c') the transition from one state to another. We can see that there are only a subset of all states which are periodic, *i.e.* which are rereached after a finite number of state transitions. In [ABM08] Arnault, Berger and Minier discuss in detail the sequences in each cell of a Galois FCSR and not only of the output sequence $\mathcal{S}$. They also show that a Galois FCSR reaches a *periodic state* after at most $n + 4$ iterations.

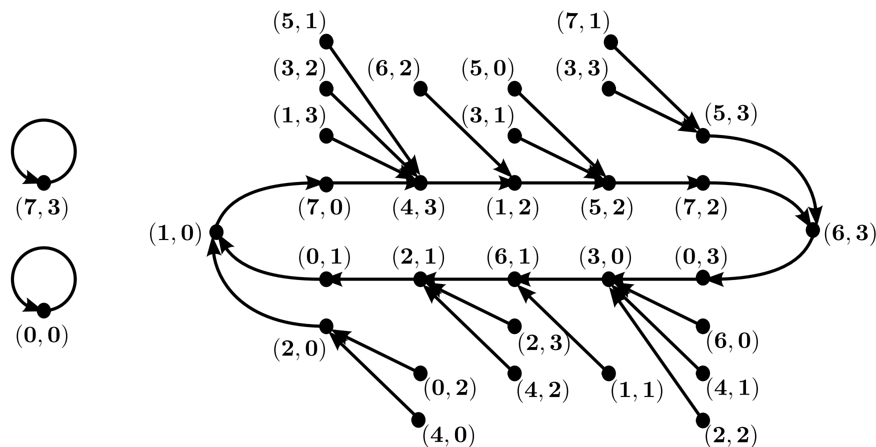


Figure 10.5: Functional graph of a Galois FCSR with $n = 3$ and $q = 13$.

In the case of Fibonacci FCSRs, there is a *one-to-one* mapping between the state (A, m) and h. This is easy to verify if we consider the equation (10.15) and the 2-adic components of $h = \sum_{i=0}^{r} h_i 2^i$:

$$\begin{aligned} h_0 &= a_0 \\ h_1 &= (a_1 - a_0 q_1) \bmod 2 \\ h_2 &= \Big(a_2 - a_1 q_1 - a_0 q_2 + (a_1 - a_0 q_1) \operatorname{div} 2\Big) \bmod 2 \\ &\dots \end{aligned}$$

Due to this one-to-one mapping, the output of a Fibonacci FCSR is *strictly periodic* if and only if its state is strictly periodic as well.

An important method to classify binary sequences is the linear complexity (Definition 10.4), *i.e.* the minimal length of an LFSR generating the sequence. The analogue for FCSRs is the 2-adic complexity.

Definition 10.21 (2-adic complexity [KG97]) *Let $\mathcal{S} = (s_0, s_1, \dots)$ be a sequence represented by the rational number $s = h/q < 0$, such that $\gcd(h, q) = 1$. Then, the 2-adic complexity is the real number $\Lambda_2(\mathcal{S}) = log_2(\max(|p|, |q|))$.*

This number does not correspond exactly to the number of cells in the FCSR. For example, for a strictly periodic sequence the 2-adic complexity will always be $log_2(|q|)$, and, thus, determine the size of the main register without counting the bits of the memory or the carry bits. However, in this case, the necessary number of bits for the auxiliary memory is bounded. Klapper and Goresky propose for Fibonacci FCSRs a second term which really determines the size in terms of memory cells of the smallest FCSR producing the sequence.

Definition 10.22 (2-adic span for Fibonacci FCSRs) *For a given Fibonacci FCSR with connection integer q and initial memory m, let $\ell = wt(q + 1)$ denote the number of $q_i = 1$ for $1 \leq i \leq n$ and*

$$\lambda = n + max\Big(\lfloor \log_2(\ell) \rfloor + 1, \lfloor \log_2(|m|) \rfloor + 1\Big) + 1$$

the number of bits in the FCSR. The 2-adic span $\lambda_2(\mathcal{S})$ of a binary, eventually periodic sequence $\mathcal{S} = (s_0, s_1, \dots)$, is the smallest value of λ which occurs among all Fibonacci FCSRs whose output is the sequence $\mathcal{S}$.

As we will see in the following, there is a direct connection between the 2-adic complexity and the 2-adic span for FCSRs.

Proposition 10.23 ([KG97]) *If $s = \sum_{i=0}^{\infty} s_i 2^i = h/q$ is the rational number, reduced to lowest terms, corresponding to an eventually periodic sequence $\mathcal{S}$, then the 2-adic span and complexity are related by*

$$|(\lambda_2(\mathcal{S}) - 2) - \Lambda_2(\mathcal{S})| \leq \log_2(\Lambda_2(\mathcal{S})) .$$

In the case of random periodic binary sequences, empirical evidences suggest that the 2-adic complexity is about half of the length of the sequence [KG95a]. However, if we combine by integer addition two sequences with small 2-adic complexity, the resulting sequence has small 2-adic complexity as well.

Theorem 10.24 ([KG97]) *Suppose $\mathcal{S}$ and $\mathcal{T}$ are periodic binary sequences with corresponding 2-adic numbers s and t. Let $\mathcal{U}$ denote the binary sequence obtained by adding the sequences $\mathcal{S}$ and $\mathcal{T}$ with carry,* i.e. *the 2-number $u = \sum_{i=0}^{\infty} u_i 2^i$ is obtained by adding $u = s + t$. Then the 2-adic complexity of $\mathcal{U}$ is bounded as follows,*

$$\Lambda_2(\mathcal{U}) \leq \Lambda_2(\mathcal{S}) + \Lambda_2(\mathcal{T}) + 1.$$

This property allows Klapper and Goresky [KG95a] to attack the Rueppel's and Massey's summation combiner [MR89, Rue86]. The principle of this generator is to combine, by addition with carry, the output of several LFSRs with maximal periods, where the periods of the LFSRs are pairwise coprime. The linear complexity of the output tends to be near the product of all periods. However, the 2-adic complexity grows only by addition of the complexities.

Now that we know the connection between a rational number $h/q < 0$ and an FCSR we want to find the smallest pair (h, q) which represents a given sequence $\mathcal{S}$, *i.e.* we search for an equivalent to the Berlekamp-Massey algorithm for FCSRs. There are two algorithms which fulfill this task. The first one was introduced by Klapper and Goresky in [KG97] and is based on De Weger's algorithm [dW79]. Klapper's and Goresky's *Rational Approximation Algorithm* is optimal in two senses:

1. Given the first T bits of $\mathcal{S}$ it always finds the smallest FCSR, in terms of the 2-adic complexity, which produces these T bits (*cf.* Theorem 10.25).

2. It finds the pair (h, p) with the minimal number of bits of $\mathcal{S}$, namely for $T \geq 2\lceil \Lambda_2(\mathcal{S}) \rceil + 2$ (*cf.* Theorem 10.26).

Theorem 10.25 ([KG97]) *Let Φ be defined for any pair $g = (g_1, g_2) \in \mathbb{Z} \times \mathbb{Z}$ as $\Phi(g) = \max(|g_1|, |g_2|)$.*

Let $g = (g_1, g_2)$ denote the output of the Rational Approximation Algorithm when the first T bits of $\mathcal{S} = (s_0, s_1, \ldots)$ are used. Then g_2 is odd,

$$s g_2 - g_1 \equiv 0 \pmod{2^T},$$

and any other pair $g' = (g'_1, g'_2)$ which satisfies these two conditions has $\Phi(g') \geq \Phi(g)$.

Theorem 10.26 ([KG97]) *Suppose $\mathcal{S} = (s_0, s_1, \ldots)$ is an eventually periodic sequence with associated 2-adic number $s = \sum_{i=0}^{\infty} s_i 2^i = h/q$. With $h, q \in \mathbb{Z}$ and $\gcd(h, q) = 1$. If $T \geq \lceil 2\Lambda_2(\mathcal{S}) \rceil + 2$ bits s_i are used, then the 2-adic Rational Approximation Algorithm outputs $g = (h, q)$.*

Let $T = \lceil 2\Lambda_2(\mathcal{S}) \rceil + 2$, then this algorithm has a complexity of $\mathcal{O}(T^2 \log T \log\log T)$.

The second algorithm was proposed by Arnault, Berger and Necer in [ABN04, AB08] and is based on Euclid's Algorithm. It has the advantage that it is easy to implement and that is has a complexity of only $\mathcal{O}(T^2)$, where $T = 2\Lambda_2(\mathcal{S}) + 3$ is the number of necessary bits to recover h and q such that $s = \sum_{i=0}^{\infty} s_i 2^i = h/q < 0$.

As in the case of LFSRs, we are interested in sequences with maximal period. We will call them ℓ-sequences. From Corollary 10.16 we know that if $s = h/q$ with h and q coprime and q odd, the period of T is $\text{ord}_q(2)$. Let φ denote Euler's Phi function. It is a widely known fact that if q is odd, $ord_q(2)|\varphi(q)$. The maximal value $\varphi(q)$ can only be reached if q is prime or a prime factor $q = p^e$ with p prime. In those two cases $\varphi(q)$ is respectively $q-1$ and $p^{e-1}(p-1)$. Klapper and Goresky gave two definitions for ℓ-sequences. The first considers the case where q is prime, whereas the second one deals with an arbitrary q.

Definition 10.27 (ℓ-sequence for q prime [KG97]) *An* ℓ-sequence *is a periodic sequence (of period $T = q - 1$) which is obtained from an FCSR with prime connection integer q for which 2 is a primitive root.*

In the same year they published a more general definition.

Definition 10.28 (ℓ-sequence (general case) [GK97]) *The period of an FCSR can be made large by choosing q so that 2 is a primitive root modulo q, i.e. $\text{ord}_q(2) = \varphi(q)$. For this to be true it is necessary that q be a power of a prime $q = p^e$, but no precise condition for 2 to be a primitive root modulo q is known. However, it is known that if 2 is a primitive root modulo p and modulo p^2, then it is a primitive root modulo p^e for every e. We call such sequences* ℓ-sequences.

Remark 10.29 *In cryptography, we are more interested in sequences corresponding to Definition 10.27, since they allow a maximal period for all $-q < h < 0$. If $q = p^e$ for $e > 1$, then all h with $\gcd(h,q) > 1$ have the period $\text{ord}_{q/gcd(h,q)}(2) < \text{ord}_q(2)$. Another advantage of q prime is that it produces a sequence of period T which the smallest q possible,* i.e. *$q = T + 1$. This leads to an FCSR of size $\log_2(T+1)$. For example, an ℓ-sequence of period $T = 18$ can be produced by $q = 19$ and $q = 27$. The preference for q prime in cryptographic applications is reflected in the following definition.*

Definition 10.30 ([ABM08]) *We say that a FCSR automaton with connection integer q is* optimal *if the order of 2 modulo q is exactly $T = q - 1$.*

Sometimes the output of multiple FCSRs is combined by bitwise XOR. In this case, the period of the generated sequence is given by the next theorem.

Theorem 10.31 ([GK06]) *Let $\mathcal{S} = (s_0, s_1, \ldots)$ be a binary periodic sequence of (minimal) period T_s with each $s_i \in \{0,1\}$, and let $\mathcal{U} = (u_0, u_1, \ldots)$ be a binary periodic sequence of (minimal) period T_u with each $u_i \in \{0,1\}$. Let $\mathcal{W} = (w_0, w_1, \ldots)$ be the sequence with $w_i = s_i \oplus u_i$ for each i. Suppose that for every prime p, the largest power of p that divides T_s is not equal to the largest power of p that divides T_u. Then $\mathcal{W}$ is periodic and the period of $\mathcal{W}$ is the least common multiple of T_s and T_u.*

Thus, the maximal period which we can reach by combining two ℓ-sequences is the following:

Corollary 10.32 ([GK06]) *Let $\mathcal{S} = (s_0, s_1, \ldots)$, $\mathcal{U} = (u_0, u_1, \ldots)$ be binary ℓ-sequences with prime connection integers q and q' respectively. Suppose that 4 divides $q-1$ but does not divide $q'-1$ and that no odd prime divides both $q-1$ and $q'-1$ (so that* $\gcd(q-1, q'-1) = 2$*). Then the sequence $\mathcal{W} = \mathcal{S} \oplus \mathcal{U}$ obtained by taking the termwise sum, modulo 2, of $\mathcal{S}$ and $\mathcal{U}$ has period $(q-1)(q'-1)/2$.*

In cryptography, we are interested in sequence which have have a uniform looking behaviour. The ideal case would be a 2*-ary de Bruijn* sequence, *i.e.* a sequence of period 2^n which contains every n-tuple $(x_0, x_1, \ldots, x_{n-1}) \in \{0,1\}^n$ exactly once in each period [vT05]. This cannot be reached directly by an FCSR. However, ℓ-sequences have almost de Bruijn properties as we will see in the following.

Theorem 10.33 ([KG97]) *Let q be a power of a prime p, say $q = p^e$, and suppose that 2 is primitive modulo q. Let r be any non-negative Integer, and let α and β be r-bit subsequences. Let $\mathcal{S}$ be any maximal period, purely periodic FCSR sequence, generated by an FCSR with connection integer q. Then the number of occurrences of α and β in $\mathcal{S}$ with their staring positions in a fixed period of $\mathcal{S}$ differ by at most 2.*

Corollary 10.34 ([ABM08]) *If q is prime and the order of 2 modulo q is maximal (*i.e. *equals to $q-1$), then the 2^n sequences with n consecutive bits appear at least one time and at most two times in a period of the binary expansion of any p/q where $-q < p < 0$.*

In addition, the period behaves symmetrically.

Proposition 10.35 ([GK97]) *If $\mathcal{S}$ is an ℓ-sequence, then the second half of one period of $\mathcal{S}$ is the bitwise complement of the first halft.*

In the case of a Galois FCSR, we know even more:

Every Galois FCSR has two fixpoints, $(0,0)$ and $(2^n - 1, \sum_{i=0}^{n-2} q_{i+1} 2^i)$. They correspond to the cases where all the cells of the FCSR contains respectively only 0's or 1's.

Proposition 10.36 ([ABLM07]) *Let q be the connection integer of a Galois FCSR of degree n and ℓ the number of carry cells. If the order of 2 modulo q is exactly $T = q-1$, then the graph contains the two fixpoints $(0,0)$ and $(2^n - 1, \sum_{i=0}^{n-2} q_{i+1} 2^i)$ and only one more component of $2^{n+\ell} - 2$ points, considered as a main cycle of length $q-1$ to which many tails converges. (*cf. *Figure 10.5).*

We even know the upper bound of steps for each point before reaching the main cycle.

Theorem 10.37 ([ABM08]) *Suppose that a Galois FCSR with connection integer q and degree n starts from the state (a, c). Then the FCSR will be in a periodic state,* i.e. *on a cycle in the functional graph, after at most $n+4$ iterations. More generally, the length of the tails of the functional graph of the FCSR are at most $n+4$.*

Until now we always considered the output sequence $\mathcal{S}$ of the FCSR, *i.e.* the sequence produced in cell a_0. However, in the case of a Galois FCSR, we can also determine the sequence in each cell of the main register. Let $\mathcal{A}_i = (a_i(t))_{t\in\mathbb{N}}$ denote the sequence generated in cell a_i for $0 \leq i \leq n-1$, thus, $\mathcal{A}_0 = \mathcal{S}$.

Theorem 10.38 ([AB05a]) *Consider an n-stage Galois FCSR with connection integer q. Then for all i such that $0 \leq i \leq n-1$, there exists an integer h_i such that $\mathcal{A}_i$ is the 2-adic expansion of h_i/q. These integers are related by the following formulas:*

$$\begin{aligned} h_i &= q a_i(0) + 2h_{i+1} && \text{if } q_{i+1} = 0 \\ h_i &= q\big(a_i(0) + 2c_{i+1}(0)\big) + 2(h_{i+1} + h_0) && \text{if } q_{i+1} = 1 \end{aligned}$$

The exact formula for each h_i can be found in [ABM08].

As in the case of LFSRs, we can directly compute the i'th output of an FCSR.

Theorem 10.39 ([KG97]) *Suppose a periodic sequence $\mathcal{S} = (s_0, s_1, \ldots)$ is generated by an FCSR with connection integer q. Let $\gamma = 2^{-1} \in \mathbb{Z}/(q)$ be the (multiplicative) inverse of 2 in the ring $\mathbb{Z}/(q)$ of integers modulo q. Then there exists $\alpha \in \mathbb{Z}/(q)$ such that for all $i = 0, 1, 2, \ldots$ we have*

$$a_i = \alpha\gamma^i \pmod q \pmod 2 . \tag{10.18}$$

Here the notation $\pmod q \pmod 2$ means that first the number $\alpha\gamma^i$ should be reduced modulo q to give a number between 0 and $q-1$, and then that number should be reduced modulo 2 to give an element of $\mathbb{F}_2$.

10.3 Applications of FCSRs

FCSRs are easy to implement, nonlinear and based on the well known theory of 2-adic numbers. They can provide large periods (*cf.* Definition 10.27) and almost de Bruijn properties(*cf.* Theorem 10.33 and Corollary 10.34). This makes them interesting for the creation of pseudorandom numbers and particularly in the use of stream ciphers. In this section, we will thus present some stream cipher proposals using FCSRs as their main component.

The first example of such a stream cipher was presented in 2005 at the Fast Software Encryption conference [AB05b]. If we would use directly the output of an FCSR as keystream for a stream cipher, it would be easy to compute the content of the register [KG95a]. This can be done by the two algorithms [KG97, ABN04] mentioned in the previous section, by using only $2\Lambda_2 + 3$ output bits. Here Λ_2 is the 2-adic complexity of the sequence, which is about the size of the main register. Therefore, Arnault and Berger propose a filtered FCSR (F-FCSR). Their main idea is to use a Galois FCSR, which is initialized by the key and the IV. They use a q prime with order $\mathrm{ord}_q(2) = q-1$ to achieve a maximal period. In addition, they request that the period $q-1 = 2p$ is twice a prime. This implies that the combination of the output sequence has periods of p or $2p$, except of degenerated cases. The output is taken by a linear filter from the main register. Since the FCSR possesses

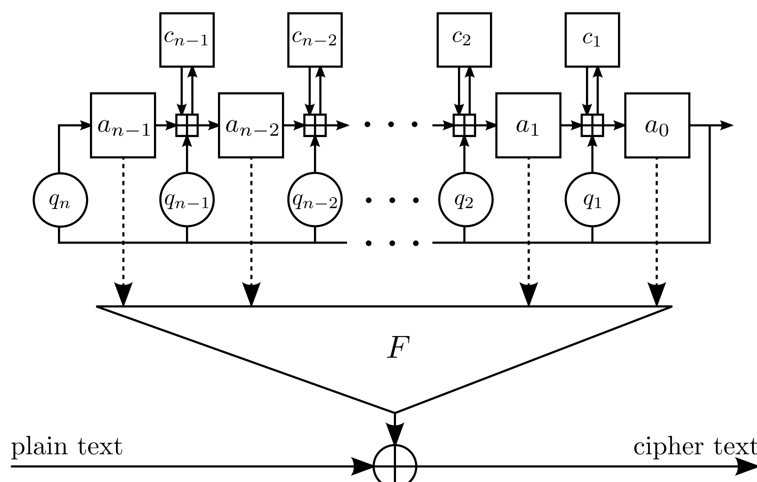


Figure 10.6: F-FCSR.

already a non-linear behavior, the authors chose linear filtering, which guarantees optimal resilience. In addition, they demand that between every cell used as inputs for the filter, there must be at least one feed-back path. Otherwise the dependence between two such cells would only be a shift. Arnault and Berger propose one FCSR with four different filters, two static and two dynamic ones. The Galois FCSR has a size of the main register of $n = 128$ bits and $\ell = 68$ carry cells. Thus, the size of the inner state is 196, thus the authors hoped that this stream cipher would have a complexity of a time/memory/data tradeoff attack [BS00] of 2^{98}. To augment the size of the state, they propose to choose the filter dynamically depending on the key. The stream cipher has a key of 128 bits and an IV of up to 64 bits. The key/IV setup is done by inserting the key into the main register and the IV into the carry cells. After six initial iterations of the FCSR, the filter produces one or eight bits from the main register at each clock cycles. The four variants of the stream cipher are the following:

F-FCSR-SF1: Uses a static filter which outputs one bit per clock cycle.

F-FCSR-SF8: Uses a static filter which outputs eight bits per clock cycle.

F-FCSR-DF1: Uses a dynamic filter which outputs one bit per clock cycle.

F-FCSR-DF8: Uses a dynamic filter which outputs eight bits per clock cycle.

The short number of only *six initial iterations* was definitely too short and allowed algebraic attacks in the IV mode [BM05]. Therefore, Arnault, Berger and Lauradoux increase the number of initial iterations to 64 for their proposals [BAL05] to the ECRYPT Stream Cipher Project [eST, RB08]. They presented two stream ciphers which are based on the principles explained in [AB05b].

F-FCSR-8:

- Key size of 128 bits and an IV of up to 128 bits.
- Size of main register $n = 128$ bits and $\ell = 68$ carry cells.
- Dynamic filter depending on the key which outputs 8 bits at each clock cycle.
- Key/IV setup:
  - Put the key K directly into the main register and put all carry cells to 0.
  - Clock 128 times the FCSR.
  - Put the first 64 bit of the IV into the carry register and clock 64 times.
  - If the IV is larger than 64 bits, put the remaining bits into the carry register and clock again 64 times.

F-FCSR-H:

- Key size of 80 bits and IV of size v bits with $32 \leq v \leq 80$.
- Size of main register $n = 160$ bits and $\ell = 82$ carry cells.
- Static filter which outputs 8 bits per clock cycle.
- Key/IV setup:
  - The main register is initialized by concatenating the key and the IV. All carry bits are set to 0.
  - Clock 160 times the FCSR.

The authors hoped that increasing the number of initial iterations would prevent further attacks. However, Jaulmes and Muller showed some general weaknesses of the design in [JM05b] and some specific attacks on the F-FCSR-H and F-FCSR-8 stream cipher in [JM05a]. They showed that a time/memory/data tradeoff attack on the F-FCSR-8 is possible with a complexity of 2^{80} in time and memory, by using only 2^{67} data bits. This is due to the fact that the main cycle in the functional graph has only a length of $q-1 \simeq 2^{128}$. As soon as this cycle is reached, there are only about 2^{128} possible states left. This corresponds to a large loss of entropy as we will see in Chapter 11. The dynamic filter is not a big problem either, since each output bit depends on only 16 secret taps, which can be guessed for the attack. The authors also pointed out that due to the slow diffusion, there still remains some statistical imbalances, even after the increased number of initial iteration. This can be used for resynchronization attacks.

As a result of these attacks Arnault, Berger and Lauradoux [ABL05] changed the key and IV setup and the size of the main register of their eSTREAM proposals. They also abandoned the idea of a dynamic filter. The final versions works at follows [ABL08]:

F-FCSR-H:

- Key size of 80 bits and IV of size of up to 80 bits
- Size of main register $n = 160$ bits and $\ell = 82$ carry cells.

- Uses a static filter which outputs 8 bits per clock cycle. The j'th bit of the output is given by:

$$\bigoplus_{i=0}^{n/8-1} q_{8i+j+1} a_{8i+j}.$$

  To use the values q_i in the filter implies that the filter tap is always at a cell directly after a feedback position.
- Key/IV setup:
  - The main register is initialized by concatenating the key and the IV. All carry bits are set to 0.
  - The FCSR is clocked 20 times. The main register is reinitialized with the generated $8 \times 20 = 160$ bits. The carry bits are set to 0.
  - The FCSR is clocked another 162 times while discarding the output. This guarantees the necessary diffusion.

F-FCSR-16:

- Key size of 128 bits and an IV of up to 128 bits.
- Size of main register $n = 256$ bits and $\ell = 130$ carry cells.
- Uses a static filter which outputs 16 bits per clock cycle. The j'th bit of the output it given by:

$$\bigoplus_{i=0}^{n/16-1} q_{16i+j+1} a_{16i+j}.$$

- Key/IV setup:
  - The main register is initialized by concatenating the key and the IV. All carry bits are set to 0.
  - The FCSR is clocked 16 times. The main register is reinitialized with the generated $16 \times 16 = 256$ bits. The carry bits are set to 0.
  - The FCSR is clocked another 258 times while discarding the output. This guarantees the necessary diffusion.

The hardware performance of this version was studied in [GB07]. The authors compared the throughput, power consumption, area-time product, throughput per area and power-area-time product of the F-FCSR-H cipher with other eSTREAM Profile II (Hardware) candidates. The performance of the F-FCSR-H lies in the average of the candidates. Finally, the authors suggest the F-FCSR-H as second choice for WLAN applications with key-length of 80 bits, *i.e.* for applications with about 10Mbps, in contrary to low-end wireless applications in RFID/WSN tags which work at a rate of about 100kHz.

Arnault, Berger and Minier showed in [ABM08] that by setting the carry bits to 0 during the initialization, it is not possible anymore for two different initial states to collide

to the same state after some iterations. This implies that the number of possible initial states is only $q - 1 \simeq 2^n$ and that the entropy does not decrease anymore. This change, together with the augmentation of the state size, makes the time/memory/data tradeoff attack unfeasible for the new version of the F-FCSR. They authors proved as well that a Galois FCSR cannot produce a sequence of $n+1$ consecutive 0's or 1's and that after only $n+4$ iterations it is sure that the FCSR reached a periodic state. They state [ABL05] that due to the $n+2$ iterations it is not possible anymore to find any statistical imbalance that can be used for a distinguishing attack. Arnault, Berger and Minier [ABM08] demonstrated that the state transition function F-FCSR has a linear behavior if all the carry bits and the feedback bit a_0 are 0 for several iterations. However, they could show that for the F-FCSR-H and F-FCSR-16 such a situation cannot appear as soon as we are on the main cycle, thus after $n+4$ iterations they should be safe. Unfortunately this property was not strong enough as was shown by Hell and Johansson [HJ08]. There are states which have all zeros in the carry bits except one in the least significant carry bit. By compensating for this single one, they obtain linear equations connecting the output to the inner state. If the carry bits stay in this way for only 18 iterations and then twice in a state of all zeros, it is possible to retrieve the whole state. Since in each iteration the eight output bits are generated independently of each other, we can write eight independent sets of equations. This simplifies the solving of the linear system. Hell and Johansson showed empirically that the probability of the F-FCSR being in a state which is favorable to the attack, is about $2^{-25.3}$. Thus, they could completely recover the internal state of the F-FCSR with a practical number keystream bits. The authors even sketch out how to retrieve the secret key, once the state is known.

The F-FCSR stream ciphers reached the third phase of the eSTREAM project. At first, they where even chosen as one of the four final candidate in the hardware profile. However, after the attack of Hell and Johansson, they were withdrawn from the final portfolio. Arnault, Berger, Minier and Pousse are working on a new version of the stream cipher which combines the Fibonacci and Galois architecture and is able to withstand the latest attack[2].

The stream ciphers mentioned above were adapted for hardware implementation, however, for software implementations they are to slow. With the X-FCSR [ABLM07], Arnault, Berger, Lauradoux and Minier introduce an FCSR based stream cipher family, which was designed for efficient software implementations. These stream ciphers use keys of 128 bits and IVs of length 64 to 128 bits. The principle is based on two FCSRs with a main register size of 256 bits each. Their corresponding connection integers q and q' are distinct, but both optimal, *i.e.* 2 is a primitive root modulo q and q', respectively. They are clocked in opposite directions and the states A and A' of the two main registers are combined by bit-wise XOR after each iteration. The connection integers where chosen in a way such that at least one of two combined bits is influenced by a feedback.

To avoid the time-intensive $n+2$ iterations after each IV setup, the X-FCSR uses a

[2]Personnel communications with François Arnault, Thierry Berger, Marine Minier, and Benjamin Pousse.

different approach. Like for the SOSEMANUK stream cipher [BBC+08], the key and IV setup are separated. For each new key, the X-FCSR extracts a set of subkeys once. This set is then used in the IV setup as roundkeys for an AES like blockcipher. The IV is encrypted and the output after rounds $12, 16, 20$ and 24 are used to fill the main registers of the two FCSRs. The carry bits are set to 0.

There are two different versions of the X-FCSR, the X-FCSR-128 and the X-FCSR-256, which generates respectively 128 and 265 bits at each clock cycle. A scheme of the X-FCSR-256 can be found in Figure 10.7. The main different between these two versions

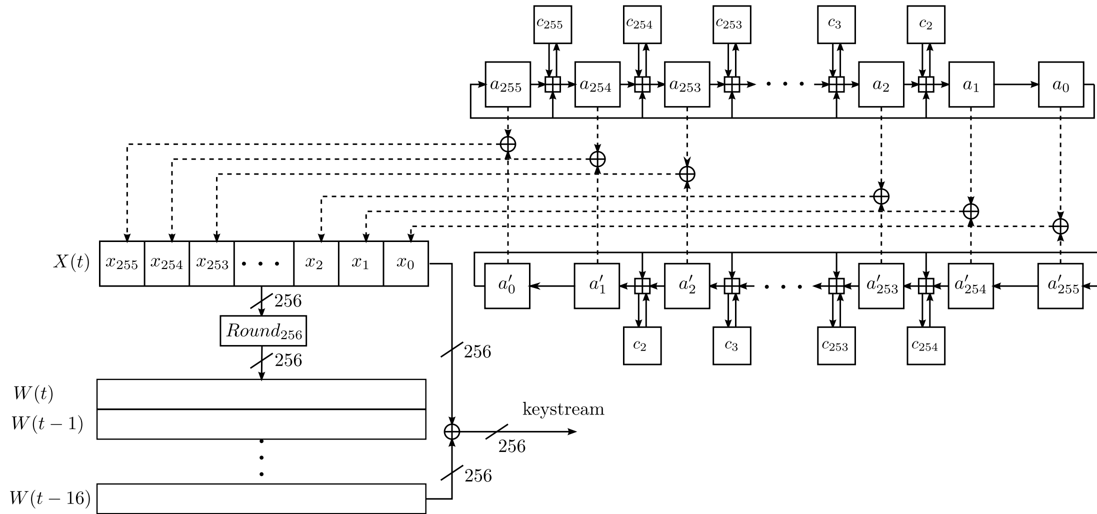


Figure 10.7: X-FCSR-256.

is that the X-FCSR-128 folds the 256 bits from the XOR of the main registers and works subsequently on 128 bits, whereas the X-FCSR-256 directly works on the 256 bits. Let $X(t)$ denote the 128 or 256 bit output from the FCSRs at time t. Then the stream cipher applies an AES like round function, respectively, $Round_{128}$ and $Round_{256}$, at $X(t)$ to obtain $W(t)$ which is stored in an auxiliary memory. This memory contains all previous values of W up to $W(t-16)$. Finally, we combine $X(t)$ and $W(t-16)$ to produce the keystream. Storing the output for 16 iteration was done to make algebraic attacks infeasible.

Unfortunately, there was also an attack found on the X-FCSR stream cipher family. Stankovski, Hell and Johansson announced an efficient key recovery attack on X-FCSR-254 at the Fast Software Encryption conference 2009 in Leuven, Belgium.

In this section, we saw different proposals of stream ciphers using FCSRs. Even when the first versions were flawed, the idea of using the nonlinear property of an FCSR in a stream cipher remains interesting and there is still space for new developments in this area.

10.4 Extensions of FCSRs

In the previous sections, we considered exclusively binary FCSRs. However, there exists a big range of extension to the binary case. We will mention some of them in this section.

The most obvious extension consists in exchanging 2 by a prime number p. This implies that all cells in the FCSR contain numbers in $\mathbb{F}_p$. Most of the properties mentioned in Section 10.2 can be applied directly to this case [KG97]. The addition-with-carry (AWC) generator [MZ91] can be seen as a Fibonacci FCSR over $\mathbb{F}_p$ for which the connection integer has the special form $q = p^a + p^b - 1$ for two different integers a and b.

The next step is to use numbers in $\mathbb{Z}/(N)$ for any arbitrary integer N. Again, most of the previous theorems can be extended to this case, as the interested reader can see in [KX89], [ABN04], and [GK08].

The first extension which does not work directly on $\mathbb{Z}/(N)$ and which has a slightly different structure is the d-FCSR.

10.4.1 d-FCSRs

Binary FCSRs are based on 2-adic numbers. The carry bit in addition and multiplication is shifted one position, for example:

$$1 + 1 = 2 = 0 \times 2^0 + 1 \times 2^1 .$$

A d-FCSRs shifts the carry not by one but by d positions. This idea was presented by Goresky and Klapper in [GK94, KG97]. We will first consider the case of binary d-FCSRS. To analyze these automata we work with the ring $\mathbb{Z}[\pi]$ where $\pi^d = 2$, *i.e.* π is a d-root of 2. Any element $z \in \mathbb{Z}[\pi]$ of the ring can be described as a polynomial $z = z_0 + z_1\pi + z_2\pi^2 + \cdots + z_{d-1}\pi^{d-1}$ with $z_i \in \mathbb{Z}$. We say that z is *nonnegative* if all $z_i \geq 0$. A nonnegative z can be uniquely described by a polynomial

$$z = \sum_{i=0}^{k} z'_i \pi^i$$

where all $z'_i \in \{0, 1\}$. This is done by applying the binary representations of the z_i's and the fact that $2 = \pi^d$. In the following, we assume that all z are nonnegative. Multiplication and addition preserve this property and use:

$$1 + 1 = 2 = 0 + 0\pi + 0\pi^2 + \cdots + 0\pi^{d-1} + 1\pi^d .$$

This means a shift of the carry bit by d positions.

In the case of a Fibonacci d-FCSR, the implementation is done in two steps, as we can see in Figure 10.8. At first, we add the values of the main register by a normal integer addition:

$$\sigma = \sum_{i=1}^{n} q_i a_{n-i} .$$

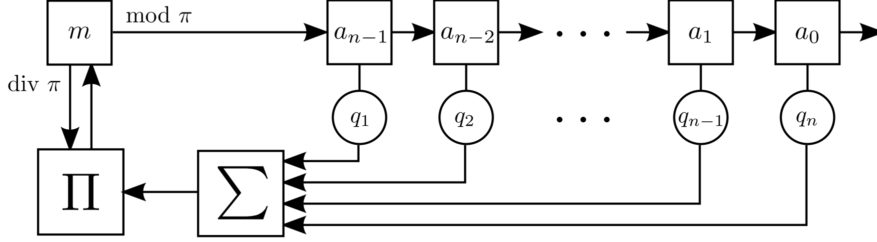


Figure 10.8: Fibonacci d-FCSR.

Subsequently, we add σ to $m = m_0 + m_1\pi + \cdots + m_j\pi^j$ by an addition in $\mathbb{Z}[\pi]$. This addition is represented by the symbol Π in Figure 10.8 . Let $\sigma = \sum_{i=0}^{k} \sigma_i 2^i$ be the binary representation of σ. Then, its corresponding polynomial is given by

$$\sigma = \sigma_0 + \sigma_1\pi^d + \sigma_2\pi^{2d} + \cdots + \sigma_k\pi^{kd} \in \mathbb{Z}[\pi] .$$

An implementation of the $\mathbb{Z}[\pi]$ adder for $d = 2$ and $\sigma \leq 7$ is shown in Figure 10.9. The carry bit is reinjected d positions later. The main register is shifted one position to the right and the cell a_{n-1} is updated with the value $\sigma + m \bmod \pi$.

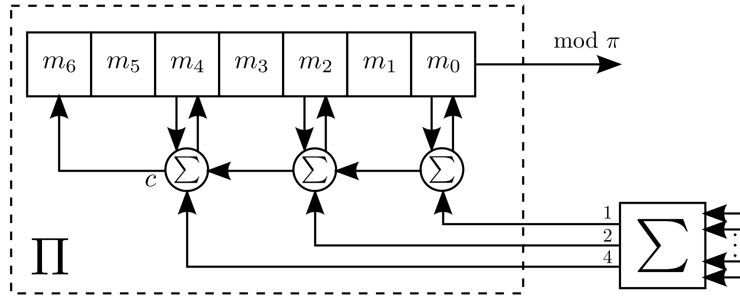


Figure 10.9: The $\mathbb{Z}[\pi]$ adder in detail for $d = 2$ and $\sum_{i=1}^{n} q_i a_{n-i} \leq 7$.

The Galois setup for a d-FCSR was presented in [GK02]. The idea is again to inject the carry of the addition $q_i a_0 + a_i$ at position $i + d$. In the case of $d = 2$, we need additional cells a_n and c_n, as we can see in Figure 10.10. For $d > 2$ we need $t + 1$ additional carry cells and $d + t - 1$ additional cells in the main register. The necessary size of t such that there will not be a memory overflow, is bounded. For more details about the size of t we refer to [GK02].

The theory of d-FCSRs is not limited to the binary case. For any integer N and any fixed $d \geq 1$ such that the polynomial $X^d - N$ is irreducible, we can define d-FCSRs over $\mathbb{Z}[\pi]$ where $\pi^d = N$.

The theory of π-adic numbers allows us to nicely describe the properties of d-FCSR. More details of their distribution and correlation properties can be found in, respectively, [Kla04] and [GK04].

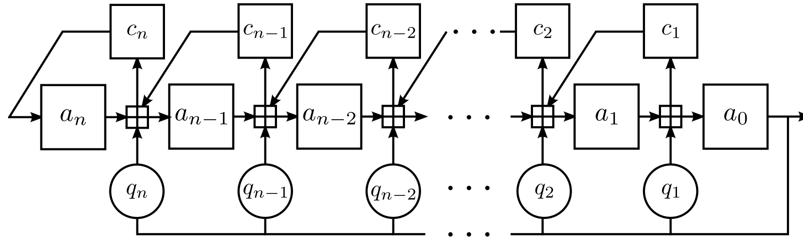


Figure 10.10: Galois d-FCSR for $d = 2$.

10.4.2 AFSR

Algebraic Feedback Shift Registers (AFSRs) are a generalization of all the previous cases. They were introduced by Klapper and Xu in [KX99].

Definition 10.40 ([KX99]) *Let R be a domain with fraction field F, principal maximal prime ideal I generated by an element π, and finite residue field $K = R/I$. Let $S, T \subseteq R$ be a pair of complete sets of residues. For any $\alpha = \sum_{i=0}^{\infty} a_i \pi^i$ the reduction of α modulo π is a_0 and the integral quotient of α by π is given by* $\mathrm{quo}(\alpha, \pi) = \alpha = \sum_{i=0}^{\infty} a_{i+1} \pi^i$.

An algebraic feedback shift register *(AFSR) over (R, π, S, T) of length n is specified by $n + 1$ elements $q_0, q_1, \ldots, q_n \in T$ called the* taps, *with $q_0 \not\equiv 0 \bmod \pi$ and q_0 invertible modulo π. It is an automaton each of whose states consist of n elements $a_0, a_1, \ldots a_{n-1} \in S$ and an element $m \in R$. The state is updated by the following steps.*

1. *Compute*

$$\tau = \sum_{i=1}^{n} q_i a_{n-i} + m$$

2. *Find $a_n \in S$ such that $q_0 a_n \equiv \tau \bmod \pi$.*

3. *Replace $(a_0, \ldots, a_{n-1})$ by $(a_1, \ldots, a_n)$ and replace m by* $\mathrm{quo}(\tau - q_0 a_n, \pi)$.

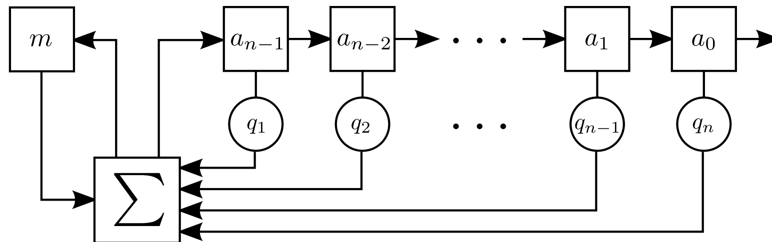


Figure 10.11: AFSR.

This definition implies the following equation:

$$q_0 a_n(t+1) + \pi m(t+1) = \sum_{i=1}^{n} q_i a_{n-i}(t) + m(t) \; . \tag{10.19}$$

It differs from the equation of a general FCSR mainly in q_0. The case for $q_0 = 1$ corresponds to an FCSR over finite fields as discussed in [Kla94]. If we set $R = \mathbb{Z}$, $\pi = 2$, $S = T = \{0, 1\}$ we have the case of the original FCSR. An LFSR over a field K is realized by $(R, \pi, S, T) = (K[X], X, K, K)$.

Klapper and Xu [KX04] studied several cases of AFSRs which allow an efficient synthesis algorithm, *i.e.* an algorithm to find the smallest AFSR generating a sequence. Klapper [Kla08] discusses the average π-adic complexity of sequences and gives the result for the case $\pi^2 = 2$. The connection between the multiply-with-carry random number generator [Mar94, CL97] and AFSRs is shown in [GK03].

The theory of AFSRs has consequences for cryptographic applications. If we want that a sequence is not easy to reproduce it should not only provide high linear complexity, but also high π-adic complexity for all cases for which efficient synthesis algorithm exists.

Chapter 11

Entropy of the Inner State of an FCSR in Galois Setup

When we use FCSRs in areas like cryptography or generation of pseudorandom sequences, we do not want an attacker to be able to easily guess the content of the register. This requires a high entropy of the inner state. In the case of a Galois FCSR, the same output sequence can be produced by different initial states. This leads to a loss of entropy in the inner state of the FCSR. In this chapter, we show that we already lose after one iteration a lot of entropy. The entropy reduces until the moment where the FCSR reaches a periodic behavior. We present an algorithm which computes the final entropy of an FCSR and which also allows us to show that the entropy never decreases under the size of the main register.

The results of this chapter were presented at the Fast Software Encryption 2008 in Lausanne, Switzerland [Röc08a].

11.1 Introduction

Let $\mathcal{S} = (s_0, s_1, \ldots)$ be the output sequence of a binary Galois FCSR (*cf.* Figure 10.4) with connection integer q of degree n. We use the integer tuple $(a(t), c(t))$ to describe the inner state of the FCSR at time t, where a and c are defined as in, respectively, (10.11) and (10.12). By ℓ we mean the number of carry bits, *i.e.* the number of $q_i = 1$ for $1 \leq i \leq n-2$. We use T to denote the state transition function $T(a(t), c(t)) = (a(t+1), c(t+1))$.

From Theorem 10.20 we know that the 2-adic number s corresponding to the output sequence $\mathcal{S}$ can be described as rational number $-h/q$. The integer $h = a(0) + 2c(0)$ is determined by the initial state and must lie in the range $0 \leq h \leq q < 2^{n+1}$. We can write $h = \sum_{i=0}^{n} h_i 2^i$.

In total, we have $n + \ell$ cells in our FCSR, thus, there are $2^{n+\ell}$ possible states. Let us enumerate the $2^{n+\ell}$ values of the state. We then denote the probability that the FCSR is in the i'th value of the state at time t by $p_i(t)$, for $1 \leq i \leq 2^{n+\ell}$. Thus, the state entropy

of the FCSR at time t is given by

$$H(t) = \sum_{i=1}^{2^{n+\ell}} p_i(t) \log_2 \left(\frac{1}{p_i(t)} \right) . \qquad (11.1)$$

We assume a uniform initial distribution, thus $p_i(0) = 1/2^{n+\ell}$ for all i. This implies that the initial entropy $H(0) = n + \ell$. If the i'th state is produced by exactly v other states at time t, then its probability is $p_i(t) = v/2^{n+\ell}$.

Remark 11.1 *In the case of a stream cipher, the assumption of a uniform initial distribution corresponds to a key/IV setup which influences the main register and the carry bits in the initial state. This was done in the* F-FCSR *described in [ABN02] and the first version of the* F-FCSR-8 *[BAL05] proposed at the eSTREAM workshop. However, this is not always the case,* e.g. *for later versions of the* F-FCSR *[ABL05], the carry bits of the initial value are always set to 0. This implies that at the beginning this stream cipher has only n bits of entropy, however, it will not lose any more entropy, as we will see later.*

We lose entropy if two states collide, *i.e* there are $(a(t), c(t)) \neq (a'(t), c'(t))$ such that $T(a(t), c(t)) = T(a'(t), c'(t))$. Such a behaviour represents a tree node with more than one child in the functional graph (*cf.* Figure 10.5). As soon as we reach a point on the cycle from any possible starting point, the FCSR behaves like a permutation and the entropy stays constant. We will denote this value by the final entropy H_f.

In this chapter, we often use the following two finite sums:

$$S_1(k) = \sum_{x=2^{k-1}+1}^{2^k} x \log_2(x) ,$$

$$S_2(k) = \sum_{x=1}^{2^k-1} x \log_2(x) .$$

For simplification reasons we will write only $S_1(k)$ and $S_2(k)$. Their exact values are hard to compute for large values of k, however, we give close upper and lower bounds in Section 11.5.

This chapter is structured in the following way: In Section 11.2, we compute the state entropy after one iteration. Subsequently, in Section 11.3, we present an algorithm which computes the final entropy for any arbitrary FCSR. This algorithm uses S_1 and S_2, which makes it difficult to evaluate for large ℓ. However, we give a method to compute very close upper and lower bounds of the entropy. The same algorithm is used in Section 11.4 to prove that the final entropy is always larger than n. In Section 11.5, we give upper and lower bounds for the sums $S_1(k)$ and $S_2(k)$ which we use to approximate the final entropy.

11.2 Entropy after One Iteration

If the initial value of the state is chosen uniformly, we have an initial state entropy of $n + \ell$ bits. We are interested in how many bits of entropy we lose already after one iteration.

Let us take an arbitrary initial state $(a(0), c(0))$ which produces the state $(a(1), c(1))$ after one iteration. To compute the probability of $(a(1), c(1))$, we want to know the number

$$v = \#\Big\{(a'(0), c'(0)) \neq (a(0), c(0)) \;\Big|\; T(a'(0), c'(0)) = (a(1), c(1))\Big\}$$

of other initial states, which produces the same state $(a(1), c(1))$ after one iteration. The probability of this state is then $p_{(a(1),c(1))} = (v + 1)/2^{n+\ell}$.

Let us consider the state transition function T as described in (10.9). We want to know which bits of $(a'(0), c'(0))$ must be the same as in $(a(0), c(0))$ and which can be different. From the last line in (10.9) we see that $a'_0(0) = a_{n-1}(1) = a_0(0)$ is fixed. For a given $1 \leq i \leq n-1$, if $q_i = 0$ we know that $c_i = 0$ as well and we can write the equation in (10.10) as

$$a_{i-1}(t + 1) = a_i(t)\ .$$

Thus for any i with $q_i = 0$ the values $a'_{i+1}(0) = a_i(1) = a_{i+1}(0)$ are already determined. By using (10.10) and the previous remarks, we can write for $q_i = 1$:

$$\begin{aligned} 2c_i(1) + a_{i-1}(1) &= a'_i(0) + c'_i(0) + a_0(0) \\ 2c_i(1) + a_{i-1}(1) &= a_i(0) + c_i(0) + a_0(0) \end{aligned}$$

and thus

$$c'_i(0) + a'_i(0) = c_i(0) + a_i(0)\ .$$

If $a_i(0) = c_i(0)$ it must hold that $a_i(0) = c'_i(0) = a'_i(0)$ since each value can only be either 0 or 1. Therefore, the only possibility for $(a'(0), c'(0))$ to differ from $(a(0), c(0))$ is if there is a position $1 \leq i \leq n-1$ with $q_i = 1$ and $a_i(0) \neq c_i(0)$.

Let j be the number of positions $1 \leq i \leq n-1$ in the initial state where $q_i = 1$ and $c_i(0) + a_i(0) = 1$. Then there are exactly $v = 2^j - 1$ other initial states which produce the same state after one iteration. Thus, $(a(1), c(1))$ has a probability of $2^j/2^{n+\ell}$.

In a next step, we want to know how many states $(a(1), c(1))$ have this probability $2^j/2^{n+\ell}$. Such a state must be created by an initial state $(a(0), c(0))$ which has j positions $1 \leq i \leq n-1$ with $q_i = 1$ and $c_i(0) + m_i(0) = 1$. There are $\binom{\ell}{j}$ possibilities to choose these positions. At the remaining $\ell - j$ positions $1 \leq i \leq n-1$ with $q_i = 1$ we have $a_i(0) = c_i(0) \in \{0, 1\}$. In the same way we can choose between 0 and 1 for the remaining $n - \ell$ positions. Thus, there exist exactly $2^{n-j}\binom{\ell}{j}$ different states $(a(1), c(1))$ with a probability of $2^j/2^{n+\ell}$.

Using (11.1), $\sum_{j=0}^{\ell}\binom{\ell}{j} = 2^\ell$ and $\sum_{j=0}^{\ell} j\binom{\ell}{j} = \ell 2^{\ell-1}$ we can write the entropy after one iterations as

$$\begin{aligned} H(1) &= \sum_{j=0}^{\ell} 2^{n-j}\binom{\ell}{j} 2^{j-n-\ell}(n + \ell - j) \\ &= n + \frac{\ell}{2}. \end{aligned}$$

We have shown that the entropy after one iteration is

$$H(1) = n + \frac{\ell}{2} \tag{11.2}$$

which is already $\frac{\ell}{2}$ bits smaller than the initial entropy.

11.3 Final State Entropy

We have shown that after one iteration the entropy has already decreased by $\ell/2$ bits. We are now interested in down to which value the entropy decreases after several iterations, *i.e.* the final entropy H_f. For the computation of H_f, we need to know how many initial states arrive at the same cycle point after the same number of iterations. We are going to use the following proposition.

Proposition 11.2 *[ABM08, Prop. 5] Two states (a, c) and (a', c') are equivalent,* i.e. *$a + 2c = a' + 2c' = h$, if and only if they eventually converge to the same state after the same number of iterations.*

Let us assume that we have iterated the FCSR sufficiently many times that we are on the cycle of the functional graph. In this case, we do not have any more collisions. If a state in the cycle is reached by v other states, it has a probability of $v/2^{n+\ell}$. After one iteration, all probabilities shift one position in the direction of the cycle, which corresponds to a permutation of the probabilities. However, the definition of the entropy is invariant to such a permutation. Let $v(h)$ denote the number of states which produce the same h. From Proposition 11.2, we know that we find a corresponding state in the cycle, which is reached by $v(h)$ states and has a probability of $v(h)/2^{n+\ell}$. We can write the final entropy by means of equation (11.1):

$$H_f = \sum_{h=0}^{q} \frac{v(h)}{2^{n+\ell}} \log_2 \left(\frac{2^{n+\ell}}{v(h)} \right). \tag{11.3}$$

Remark 11.3 *Let us have a look at an FCSR as mentioned in Remark 11.1, which always sets $c(0) = 0$. For each $0 \leq h < 2^n$ there is only one possibility to form $h = a$. This means that there are no collision and the final entropy is the same as the initial entropy, namely n bits.*

The numerator h can take any value between 0 and $2^n - 1 + \sum_{i=1}^{n-1} q_i 2^i = q$. We look at the binary representations of $a, 2c$ and h to find all possible pairs (a, c) which correspond to a given h. We study bit per bit which values are possible. This idea is presented in Figure 11.1, where each box represents a possible position of a bit. We mean by $2c$ the carry bits of the value c all shifted one position to the left.

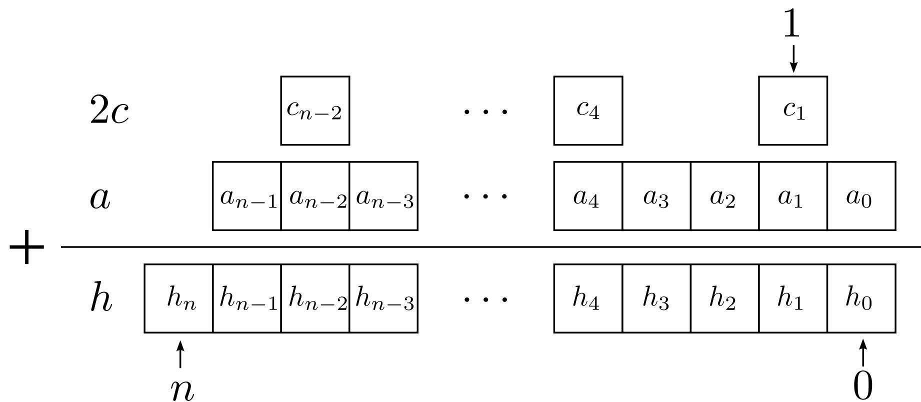


Figure 11.1: Evaluation of $h = a + 2c$ bit per bit.

11.3.1 Some Technical Terms

Before continuing, we give some technical terms which we need for our computation:

In the following, we consider only the integer addition $a+2c$, thus, if we talk of a carry we mean the carry of this addition. We mean by

$$ca(j) = [a_j + c_j + ca(j-1)] \text{ div } 2$$

the carry which is produced by adding the bits a_j, c_j and the carry $ca(j-1)$ of the previous position. *E.g.* if we have $a = 13$ and $c = 2$ we have $ca(1) = 0$ and $ca(2) = ca(3) = 1$ (*c.f* Figure 11.2). The value $ca(0)$ is always 0 since we only have a_0 for the sum.

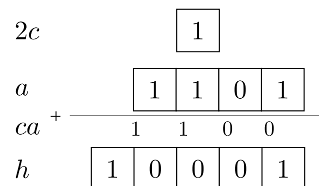


Figure 11.2: Example for $a = 13$ and $c = 2$.

In our computation, we often consider a specific value of h and want to know the position of its most significant bit which is not 0. We denote this position by

$$i(h) := \lfloor \log_2(h) \rfloor \; .$$

Most of the time, we will use directly $i = i(h)$. We refer by

$$\ell'(i) = \#\{j \leq i | q_j = 1\}$$

to the number of indices smaller or equal to i for which $q_j = 1$.

For a given h let

$$r(h) = \max\{j < i | q_j = 0, h_j = 1\}$$

be the highest index smaller than i such that $q_j = 0$ and $h_j = 1$. In this case, the carry of the integer addition $a + 2c$ cannot be forwarded over the index $r(h)$, *i.e.* we always have $ca(r(h)) = 0$, independently of $ca(r(h) - 1)$. If there is no index j with $q_j = 0$ and $h_j = 1$, we set $r(h) = -1$. For the case $i < n$ and $q_i = 0$, we get a range of $-1 \leq r(h) < i$. Let

$$r^* = \max\{1 \leq j \leq n - 1 | q_j = 1\}$$

denote the maximal index $j < n$ such that $q_j = 1$. In the case of $2^n \leq h \leq q$, the value $r(h)$ is only possible for $-1 \leq r(h) < r^*$. For simplicity reasons, we sometimes write only r if it is clear which h we are meaning or if there are multiple h's with the same value for $r(h)$.

In the further computation we will use the following helping function:

$$f_1(r) = \begin{cases} 2^r & \text{for } r \geq 0 \\ 1 & \text{for } r = -1 \end{cases} .$$

It is used to combine the cases where $r = -1$ and $r \geq 0$.

For a given r such that $r = r(h)$, we define by

$$\ell''(r) = \#\{j < r | q_j = 1\}$$

the number of indices strictly smaller than r for which $q_j = 1$. Again, we use sometimes only ℓ'' if it is clear to which r we refer.

As mentioned above, we will denote by

$$v(h) = \#\{(a, c) | a + 2c = h\}$$

the number of pairs (a, c) which create $h = a + 2c$.

In Case 2 in Section 11.3.2, we will see that it is sufficient to consider the indices j with $r(h) < j < i$ and $q_i = 1$ to know if there is a carry at position $i - 1$. To facilitate the computation, we use the following notations:

We mean with h', a' *and* c' the bit strings $h, a, 2c$ reduced to the positions j with $r < j < i$ and $q_j = 1$. An example can be seen in Figure 11.3.

Remark 11.4 *The value c' corresponds to $2c$ and not to c. Thus, we consider a' and c' (and not $2c'$) such that $h' = a' + c'$.*

In contrast to c, the value of c' is a continuous bit string, since we are only interested in positions where there is a feedback register. The length of h', a' and c' is $\ell' - \ell'' - 1$ bits.

We define by $0h'$ and $1h'$ the integers which we obtain by concatenating, respectively, 0 and 1 to the left of the bit string h'.

Let $k = \ell' - \ell'' - 1$ be the size of h'. We denote by $X(h')$ the number of possible bit strings a' and c' such that $1h' = a' + c'$, *i.e.* $2^k + h' = a' + c'$. This corresponds to the case where we have a carry $ca(i - 1)$ in the addition $h = a + 2c$. We use the following lemma to determine the value of $X(h')$.

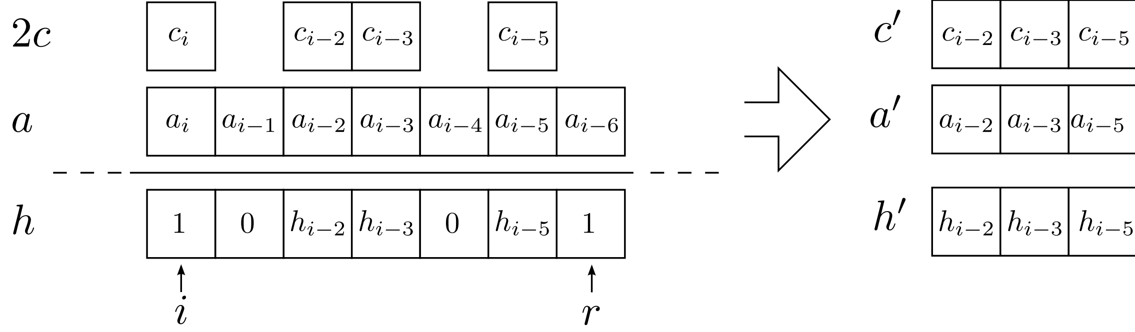


Figure 11.3: Reduction of $h, a, 2c$ to h', a', c'.

Lemma 11.5 *Let h', a' and c' be three bit strings of length k. For all $0 \leq x \leq 2^k - 1$, there exists* exactly one *h' with $X(h') = x$,* i.e. *there exists exactly one h' such that for x different pairs (a', c') we can write $1h' = a' + c'$.*

Proof.
Let h', a', c' be three bit strings of size k and let $X(h')$ be the number of possible pairs (a', c') such that $1h' = a' + c'$. We are going to use the following properties:

- For a given h', let (a', c') be such that $1h' = a' + c'$. We have two possibilities, $10h' = 0a' + 1c' = 1a' + 0c'$, to create $10h'$. If (a', c') is such that $0h' = a' + c'$, we only have one possibility, $10h' = 1a' + 1c'$, to build $10h'$. Thus, we can write:

$$X(0h') = 2X(h') + \left(2^k - X(h')\right) \ . \tag{11.4}$$

- The only possibility to create $11h'$ is $11h' = 1a' + 1c'$ for a (a', c') such that $1h' = a' + c'$. Therefore, it holds that

$$X(1h') = X(h') \ . \tag{11.5}$$

- It is easy to see that

$$X(1) \ = \ 0 \ , \tag{11.6}$$
$$X(0) \ = \ 1 \ . \tag{11.7}$$

For any $0 \leq j \leq k-1$, let h'_j denote the j'th bit of h'. From Equations (11.4),(11.5),(11.6), and (11.7), we see that for any position $0 \leq j \leq k-1$ with $h'_j = 0$ we have to count 2^j in $X(h')$. If $h'_j = 1$, we count nothing. Thus $X(h')$ is the integer value corresponding to the complement of h' and we can write:

$$X(h') = 2^k - 1 - h' \ . \tag{11.8}$$

This one-to-one mapping between h' and $X(h')$ shows that for any $0 \leq x \leq 2^k - 1$, there is exactly one value $h' = 2^k - 1 - x$ such that $X(h') = x$.

◇

11.3.2 Final Entropy Case by Case

We cannot write the sum (11.3) for the final entropy directly in a simple equation. However, we partition the set of all possible $0 \leq h \leq q$ in four cases. For each case, we will evaluate the value $\frac{v(h)}{2^{n+\ell}} \log_2 \left(\frac{2^{n+\ell}}{v(h)} \right)$ for all its h's. We obtain the final sum of the entropy by summing all these values.

Case 1: $1 \leq i < n$ and $q_i = 0$

To create $h_i = 1$ at position i, we have to add $a_i + ca(i-1)$. For each value of $ca(i-1)$ there exists exactly one possibility for a_i. For each position j with a feedback bit, $q_j = 1$, we have two possibilities to create the value of h_j. In this case, we can write for each h:

$$v(h) = 2^{\ell'}.$$

For each i, all h's within the range $[2^i, 2^{i+1}[$ are possible. So we must add:

$$H_1(n, i, \ell, \ell') = 2^i \, 2^{\ell' - n - \ell}(n + \ell - \ell') \tag{11.9}$$

to the entropy for each $1 \leq i \leq n$ with $q_i = 0$.

Case 2: $1 \leq i < n$ and $q_i = 1$:

For a given h we know from the definition of $r(h)$ that for all j's with $r(h) < j < i$, if $q_j = 0$, then $h_j = 0$. In the case of $(q_j = 0, h_j = 0)$, a carry is always forwarded. This means that for $a + 2c$, if we have $ca(j-1) = 1$, a_j must be 1 and we have $ca(j) = 1$. However, with $ca(j-1) = 0$ we have $a_j = 0$, and so we have $ca(j) = 0$ as well. It is sufficient to consider the $\ell' - \ell'' - 1$ positions j with $i > j > r(h)$ for which $q_j = 1$, to know if we have a carry at index $i - 1$.
From the definition of this case, we know that $h_i = 1$ and $q_i = 1$. If we have $ca(i-1) = 0$ we have two possibilities for (a_i, c_i), namely $(1, 0)$ and $(0, 1)$, to generate the $h_i = 1$. Otherwise, we only have one possibility $(0, 0)$. For a given h' we have:

$$X(h') + 2(2^{\ell' - \ell'' - 1} - X(h')) = 2^{\ell' - \ell''} - X(h') \tag{11.10}$$

possibilities to choose a',c' and (a_i, c_{i-1}). We use Lemma 11.5 to compute $X(h')$. In our case: $k = \ell' - \ell'' - 1$. The next question is, how many h's have the same reduction h'. If $0 \leq r < i$ we have 2^r possibilities, in the case $r = -1$ we have only one. We consider this behavior by using the helping function $f_1(r)$.

By combining Lemma 11.5 and (11.10), we obtain that for each $2^{\ell' - \ell'' - 1} + 1 \leq y \leq 2^{\ell' - \ell''}$ there is exactly one h' which is generated by y different combinations of a' and c'. Each h which corresponds to such a h', is generated by $y 2^{\ell''}$ different pairs (a, c), since at each position $j < r$ with $q_j = 1$ we have two possibilities to create h. This means that this h has a probability of $\frac{y 2^{\ell''}}{2^{n+\ell}}$.

For fixed values of i, r, ℓ' and ℓ'' we have to add the following value to the entropy:

$$\begin{aligned} & H_2(n, r, \ell, \ell', \ell'') \\ = \; & f_1(r) \sum_{y=2^{\ell'-\ell''-1}+1}^{2^{\ell'-\ell''}} \frac{y 2^{\ell''}}{2^{n+\ell}} \log_2 \left(\frac{2^{n+\ell}}{y 2^{\ell''}} \right) \\ = \; & f_1(r) 2^{-n-\ell} \left[2^{\ell'-2} \left(3 \, 2^{\ell'-\ell''-1} + 1 \right) (n + \ell - \ell'') - 2^{\ell''} S_1(\ell' - \ell'') \right]. \end{aligned}$$

Thus, in this case, we have to add for every $1 \leq i \leq n-1$ with $q_i = 1$, and every $-1 \leq r < i$ with $q_r = 0$ the value:

$$H_2(n, r, \ell, \ell', \ell'') = f_1(r) 2^{-n-\ell} \left[2^{\ell'-2} \left(3 \, 2^{\ell'-\ell''-1} + 1 \right) (n + \ell - \ell'') - 2^{\ell''} S_1(\ell' - \ell'') \right] \tag{11.11}$$

where $\ell' = \ell'(i)$ and $\ell'' = \ell''(r)$.

Case 3**:** $i = n$, $2^n \leq h \leq q$:

In this case, we always need a carry $ca(n-1)$ to create $h_n = 1$. Like in the previous case, we are going to use $r(h)$, ℓ'', h', (a', c') and $X(h')$. However, this time we have $i = n$ and $\ell = \ell'$.

For $h = q$, which means that (a, c) consists of only 1's, it holds that for all $n > j > r^*$, we have $h_j = 0$. If we would have a $r(h) \geq r^*$, then h would be greater than q which is not allowed. Therefore, r must be in the range of $-1 \leq r < r^*$.

From Lemma 11.5, we know that for all $1 \leq x \leq 2^{\ell-\ell''} - 1$ there exists exactly one h' with $X(h') = x$, *i.e.* there are x pairs of (a', c') with $1h' = a' + c'$. We exclude the case $X(h') = 0$, because we are only interested in h's that are able to create a carry. For each h', there are $f_1(r)$ possible values of h which are reduced to h'. If h' is created by x different pairs of (a', c'), then each of its corresponding values of h is created by $x 2^{\ell''}$ pairs of (a, c) and has a probability of $\frac{x 2^{\ell''}}{2^{n+\ell}}$.

For a given r and ℓ'' the corresponding summand of the entropy is:

$$\begin{aligned} H_3(n, r, \ell, \ell'') &= f_1(r) \sum_{x=1}^{2^{\ell-\ell''}-1} \frac{x 2^{\ell''}}{2^{n+\ell}} \log_2 \left(\frac{2^{n+\ell}}{x 2^{\ell''}} \right) \\ &= f_1(r) 2^{-n} \left[2^{-1} \left(2^{\ell-\ell''} - 1 \right) (n + \ell - \ell'') - 2^{\ell''-\ell} S_2(\ell - \ell'') \right]. \end{aligned}$$

In this case, we have to add for each value of $-1 \leq r < r^*$ with $q_r = 0$ and $\ell'' = \ell''(r)$:

$$H_3(n, r, \ell, \ell'') = f_1(r) 2^{-n} \left[2^{-1} \left(2^{\ell-\ell''} - 1 \right) (n + \ell - \ell'') - 2^{\ell''-\ell} S_2(\ell - \ell'') \right]. \tag{11.12}$$

Case 4**:** $0 \leq h \leq 1$

If $h = 0$ or $h = 1$, there exists only one pair (a, c) which can produce the corresponding h. Thus, for each of these h's we have to add:

$$H_4(n, \ell) = 2^{-n-\ell}(n + \ell) \tag{11.13}$$

to the sum of the entropy.

The Algorithm 2 shows how we can compute the final entropy for an FCSR defined by n and q using the summands of the individual cases.

Algorithm 2 Final entropy.

```
1: H_f ← 0
2: ℓ' ← 0
3: ℓ ← Σ_{i=1}^{n−1} q_i
4: H_f ← H_f + 2H_4(n, ℓ) {h = 0 and h = 1}
5: for i = 1 to n − 1 do
6:    if q_i = 0 then
7:       H_f ← H_f + H_1(n, i, ℓ, ℓ')
8:    else {q_i = 1}
9:       ℓ' ← ℓ' + 1
10:      ℓ'' ← 0
11:      for r = −1 to i − 1 do
12:         if q_r = 0 then
13:            H_f ← H_f + H_2(n, r, ℓ, ℓ', ℓ'')
14:         else {q_r = 1}
15:            ℓ'' ← ℓ'' + 1
16:         end if
17:      end for
18:   end if
19: end for
20: ℓ'' ← 0
21: for r = −1 to r* do {2^n ≤ h ≤ q}
22:    if q_r = 0 then
23:       H_f ← H_f + H_3(n, r, ℓ, ℓ'')
24:    else {q_r = 1}
25:       ℓ'' ← ℓ'' + 1
26:    end if
27: end for
```

11.3.3 Complexity of the Computation

In our algorithm we use the sums $S_1(k) = \sum_{x=1}^{2^k-1} x \log_2(x)$ and $S_2(k) = \sum_{x=2^{k-1}+1}^{2^k} x \log_2(x)$. If we have stored the values $S_1(k)$ and $S_1(k)$ for $1 \leq k \leq \ell$, we are able to compute the final entropy in $\mathcal{O}(n^2)$. We need $\mathcal{O}\left(2^\ell\right)$ steps to evaluate both sums, which is impractical for large ℓ. However, by using the bounds (11.19)-(11.20) for larger k's, we can easily compute a lower and upper bound of those sums. In Table 11.1, we compare for different FCSRs the exact computation with those using upper and lower bounds. The values of q

are arbitrarily chosen. However, the accuracy of the estimation of the sums can be shown anyway.

We mean by H_f the exact computation of the final entropy. The values lb/ub H_f mean the lower and the upper bound obtained by using the approximation of S_1 and S_2. The last two columns, lb/ub H_f, $k > 5$, we gain by using the approximations only for $k > 5$. This last approximation is as close that we do not see any difference between the lower and the upper bound in the first 8 decimal places.

n	q	ℓ	H_f	lb H_f	ub H_f	lb H_f , $k > 5$	ub H_f, $k > 5$
8	0x15B	4	8.3039849	8.283642	8.3146356	8.3039849	8.3039849
16	0x148BB	7	16.270332	16.237686	16.287598	16.270332	16.270332
24	0x14B369B	12	24.273305	24.241851	24.289814	24.273304	24.273305
32	0x14A96F8BB	17		32.241192	32.289476	32.272834	32.272834

Table 11.1: Comparison of the exact computation of the final state entropy, with upper and lower bounds.

11.4 Lower Bound of the Entropy

We can use the algorithm of the previous section and induction to give a lower bound of the final state entropy. For a given n and ℓ we first compute the entropy for an FCSR where all the carry bits are the ℓ least significant bits. Subsequently, we show that by moving a feedback position to the left, the direction of the most significant bit, we increase the final entropy. In both steps, we study all $0 \leq h \leq q$ case by case and use the summands of the entropy H_1, H_2, H_3 and H_4 as presented in Section 11.3.

11.4.1 Basis of Induction

For a fixed n and ℓ we study the final state entropy of an FCSR with

$$q = 2^n + \sum_{i=1}^{\ell} 2^i - 1 = 2^n + 2^{\ell+1} - 3 \ .$$

This represents an FCSR which has all its feedback positions grouped together at the least significant bits (see Figure 11.4). As in the previous section, we are going to compute the entropy case by case.

- $h = 0$ and $h = 1$.
- $1 \leq i \leq \ell$: Here we have $q_i = 1$, $\ell' = i$, $r = -1$ or 0 and thus $\ell'' = 0$.
- $i = n$, *i.e.* $\ell < i < n$: We have $q_i = 0$ and $\ell' = \ell$.

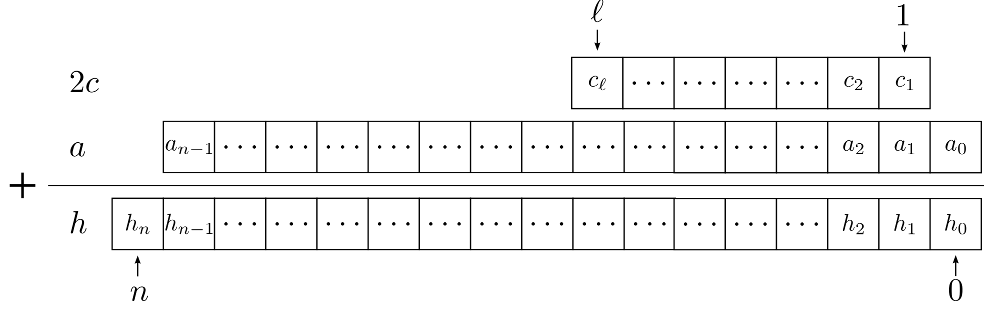


Figure 11.4: FCSR with $q = 2^n + 2^{\ell+1} - 3$.

- $2^n \leq h \leq q$: In Section 11.3.1 we showed that $-1 \leq r(h) < r^*$, where $r^* = \max\{1 \leq j \leq n-1 | q_j = 1\}$. In our case we have $r^* = \ell$. An index r requires that $q_r = 0$, thus, the only possible values for r are -1 and 0. Hence, we have $\ell'' = 0$.

So in this case, the final entropy is:

$$\begin{aligned} H_f(n,q) = & \quad 2H_4(n,\ell) \\ & + \sum_{i=1}^{\ell} \left(H_2(n,-1,\ell,i,0) + H_2(n,0,\ell,i,0)\right) \\ & + \sum_{i=\ell+1}^{n-1} H_1(n,i,\ell,\ell) \\ & + H_3(n,-1,\ell,0) + H_3(n,0,\ell,0) \\ = & \quad n + \ell\left(2^{-n+\ell+1} - 2^{-n+1}\right) - 2^{-n-\ell+2} S_2(\ell) \ . \end{aligned}$$

By using the lower bound (11.20) for $S_2(\ell)$ we can write:

$$H_f(n,q) \geq n + \frac{2^{-n+\ell+2}}{12\ln(2)}\left(3 - (4+\ell)2^{-2\ell} - 2^{-3\ell} + 2^{1-4\ell}\right) \ .$$

Let us examine the function $g(\ell) = 3 - (4+\ell)2^{-2\ell} - 2^{-3\ell} + 2^{1-4\ell}$. It is easy to verify that $g(\ell+1) - g(\ell) > 0$ for all $\ell \geq 1$. Thus, we can write $g(\ell) \geq g(1) = 7/4$ for all $\ell \geq 1$ and finally:

$$H_f(n,q) \geq n + 2^{-n+\ell}\frac{7}{12\ln(2)} \geq n \ . \qquad (11.14)$$

11.4.2 Induction Step

We show that by moving one feedback position one position to the left, which means in direction of the most significant bit, the final state entropy increases. To prove this, we

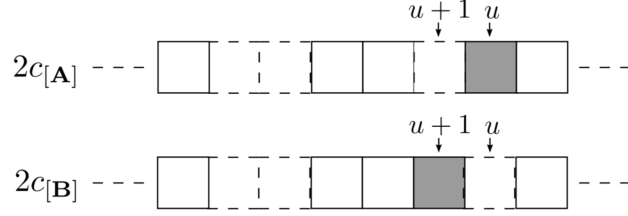


Figure 11.5: Moving the feedback position.

compare two cases **A** and **B**. In Case **A** we choose a u such that $q_u = 1$ and $q_{u+1} = 0$. It must hold that:

$$1 \leq \quad u \quad \leq n-2, \tag{11.15}$$
$$3 \leq \quad n \quad . \tag{11.16}$$

To create Case **B**, we move the feedback at index u one position to the left. This is equivalent to:

$$q_{[\mathbf{B}]} = q_{[\mathbf{A}]} - 2^u + 2^{u+1}.$$

In Figure 11.5, we display the two values $2c_{[\mathbf{A}]}$ and $2c_{[\mathbf{B}]}$ which we are going to use to compute the different finale entropies. Let

$$L = \#\{j < u | q_j = 1\}$$

be the number of feedback positions smaller than u. We mean by

$$I^r_{<u} = \{-1 \leq j < u : q_i = 0\}$$

the set of all indices smaller than u where there is no feedback and which are thus possible values for r. It must hold that $|I^r_{<s}| = u - L$.

We want to show that:

$$H_{f[\mathbf{B}]} - H_{f[\mathbf{A}]} \geq 0 \ . \tag{11.17}$$

To do this, we study the different cases of h. Each time, we examine if the summands from the algorithm in Section 11.3 are different or not. In the end we sum up the differences.

- $h = 0$ or $h = 1$: The summands of the entropy in **A** and **B** are the same.
- $i < u$: The summands of the entropy in **A** and **B** are the same.
- $n > i > u + 1$:
  - $q_i = 0$: In this case, i and ℓ' are the same for **A** and **B** and thus the summands of the entropy as well.
  - $q_i = 1$: We have:
$$L + 2 \leq \ell' \leq \ell \ .$$

    * $r < u$: In this case, i, r, ℓ' and ℓ'' are the same and thus the summands of the entropy as well.
    * $u + 1 < r$: In this case, i, r, ℓ' and ℓ'' are the same and thus the summands of the entropy as well.
    * $u \leq r \leq u + 1$:
      · **A**: Since for r it must hold that $q_r = 0$, we get $r = u + 1$ and thus $\ell'' = L + 1$. In this case, we have to count:

$$H_2(n, u + 1, \ell, \ell', L + 1) .$$

      · **B**: In this case, we have $r = u$ and $\ell'' = L$ and therefore the term:

$$H_2(n, u, \ell, \ell', L) .$$

- $2^n \leq h \leq q$:
  - $r < u$: In this case, i, r and ℓ' are the same and thus the summands of the entropy as well.
  - $u + 1 < r$: In this case, i, r and ℓ' are the same and thus the summands of the entropy as well.
  - $u \leq r \leq u + 1$:
    * **A**: We have $r = u + 1$ and $\ell'' = L + 1$. Therefore, the summand of the entropy in this case is:

$$H_3(n, u + 1, \ell, L + 1) .$$

    * **B**: We have to consider $r = u$ and $\ell'' = L$ and therefore:

$$H_3(n, u, \ell, L) .$$

- $u \leq i \leq u + 1$: We are going to use $\ell''(r)$ to denote the value of ℓ'' corresponding to a specific r.
  - $i = u$:
    * **A**: In this case, we have $q_i = 1$ and $\ell' = L + 1$. Thus we have to count for each $r \in I^r_{<u}$:

$$H_2(n, r, \ell, L + 1, \ell''(r)) .$$

    * **B**: Since $q_i = 0$ and $\ell' = L$ we get:

$$H_1(n, u, \ell, L) .$$

  - $i = u + 1$:

* **A**: In this case, we have $\ell' = L + 1$ and $q_i = 0$, thus we need to consider:

$$H_1(n, u+1, \ell, L+1) \, .$$

* **B**: This time, we have $q_i = 1$. For $\ell' = L + 1$, $r \in I^r_{<u}$ and $r = u$ we get:

$$H_2(n, r, \ell, L+1, \ell''(r)) \, .$$

In the case of $r = u$, we can write $\ell'' = L$.

By combining all these results, we get a difference of the final entropies of:

$$\begin{aligned}
H_{f[\mathbf{B}]} - H_{f[\mathbf{A}]} &= \sum_{\ell'=L+2}^{\ell} \left(H_2(n, u, \ell, \ell', L) - H_2(n, u+1, \ell, \ell', L+1) \right) \\
&+ H_3(n, u, \ell, L) - H_3(n, u+1, \ell, L+1) \\
&+ H_1(n, u, \ell, L) - \sum_{r \in I^r_{<u}} H_2(n, r, \ell, L+1, \ell''(r)) \\
&+ \sum_{r \in I^r_{<u}} H_2(n, r, \ell, L+1, \ell''(r)) + H_2(n, u, \ell, L+1, L) \\
&- H_1(n, u+1, \ell, L+1) \\
&= 2^{\ell-1} (4\ell - 4L - 2) + 2^{2\ell - L} + 2^{L+1} \left(3S_2(\ell - L - 1) - S_1(\ell - L) \right) \, .
\end{aligned}$$

If we use the lower bound (11.20) for $S_2(\ell - L - 1)$ and the upper bound (11.19) for $S_1(\ell - L)$, we can write:

$$H_{f[\mathbf{B}]} - H_{f[\mathbf{A}]} \geq 2^L \left((\ell - L) \frac{1}{4 \ln(2)} + \frac{7}{12 \ln(2)} + 2^{-(\ell - L)} \frac{14 - 12 \, 2^{-(\ell - L)}}{12 \ln(2)} \right) \, .$$

From $\ell > L$, it follows directly (11.17), which means that the difference of the final entropies is greater or equal to 0.

Every FCSR with ℓ feedback positions can be build by starting with the FCSR described in Section 11.4.1 and successively moving one feedback position to the left. Thus, by combining (11.14) and (11.17) we write the following theorem.

Theorem 11.6 *An FCSR in Galois architecture, with given values for n and ℓ, has at least $n + 2^{\ell - n} \frac{7}{12 \ln(2)} \geq n$ bits of entropy if all $2^{n+\ell}$ initial states appear with the same probability.*

11.5 Bounds for the Sums

In this section, we prove the following lower and upper bounds for the sums $S_1(k)$ and $S_2(k)$:

$$\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) \geq 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)+2^{k-2}(k+1)+\frac{1-2^{-k+1}}{24\ln(2)} , \quad (11.18)$$

$$\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) \leq 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)+2^{k-2}(k+1)+\frac{1-2^{-k}+3\,2^{1-2k}}{12\ln(2)} , \quad (11.19)$$

$$\sum_{x=1}^{2^k-1} x\log_2(x) \geq 2^{2k-1}\left(k-\frac{1}{2\ln(2)}\right)-2^{k-1}k+\frac{4+k+2^{-k+1}}{24\ln(2)} , \quad (11.20)$$

$$\sum_{x=1}^{2^k-1} x\log_2(x) \leq 2^{2k-1}\left(k-\frac{1}{2\ln(2)}\right)-k\,2^{k-1}+\frac{4+k+2^{-k}-2^{1-2k}}{12\ln(2)} . \quad (11.21)$$

The idea of this proof is that:

$$\frac{1}{2}\Big(x\log_2(x)+(x+1)\log_2(x+1)\Big) \approx \int_x^{x+1} y\log_2(y)\,dy ,$$

since $\log_2(x)$ increases much slower than x and, thus, $x\log_2(x)$ is almost a straight line. This integral can be directly computed by:

$$\begin{aligned}\int_x^{x+1} y\log_2(y)\,dy &= \frac{y^2}{2}\left(\log_2(y)-\frac{1}{2\ln(2)}\right)\Bigg|_{y=x}^{x+1} \\ &= \frac{1}{2}\Big(x\log_2(x)+(x+1)\log_2(x+1)\Big)-\frac{1+2x}{4\ln(2)} \\ &\quad +\log_2\left(1+\frac{1}{x}\right)\frac{x}{2}(x+1) .\end{aligned}$$

We use the approximation of the natural logarithm:

$$\frac{1}{\ln(2)}\left(\frac{1}{x}-\frac{1}{2x^2}+\frac{1}{3x^3}-\frac{1}{4x^4}\right) \leq \log_2\left(1+\frac{1}{x}\right) \leq \frac{1}{\ln(2)}\left(\frac{1}{x}-\frac{1}{2x^2}+\frac{1}{3x^3}\right)$$

for $x>0$ to get:

$$\frac{1+2x}{4\ln(2)}-\frac{\left(\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}\right)}{4\ln(2)} \leq \frac{x}{2}(x+1)\log_2\left(1+\frac{1}{x}\right) \leq \frac{1+2x}{4\ln(2)}-\frac{\left(\frac{1}{3x}-\frac{2}{3x^2}\right)}{4\ln(2)}$$

and finally the bounds for the integral:

$$\int_x^{x+1} y\log_2(y)\,dy \geq \frac{1}{2}\left(x\log_2(x)+(x+1)\log_2(x+1)\right)-\frac{\left(\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}\right)}{4\ln(2)} \quad (11.22)$$

$$\int_x^{x+1} y\log_2(y)\,dy \leq \frac{1}{2}\left(x\log_2(x)+(x+1)\log_2(x+1)\right)-\frac{\left(\frac{1}{3x}-\frac{2}{3x^2}\right)}{4\ln(2)} . \quad (11.23)$$

By combining the exact value of the integral:

$$\int_{2^{k-1}}^{2^k} y \log_2(y)\, dy = 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)$$

with the lower bound:

$$\begin{aligned}
&\int_{2^{k-1}}^{2^k} y \log_2(y)\, dy \\
&\geq \frac{1}{2}\sum_{x=2^{k-1}}^{2^k-1} x\log_2(x) + \frac{1}{2}\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) - \frac{1}{4\ln(2)}\sum_{x=2^{k-1}}^{2^k-1}\left(\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}\right) \\
&= \sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) - 2^{k-2}(k+1) - \frac{1}{4\ln(2)}\sum_{x=2^{k-1}}^{2^k-1}\left(\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}\right) .
\end{aligned}$$

gained by means of (11.22), we receive the upper bound:

$$\begin{aligned}
\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) \leq\ & 2^{2k-3}\left(3k+1-\tfrac{3}{2\ln(2)}\right) + 2^{k-2}(k+1) \\
& + \tfrac{1}{4\ln(2)}\textstyle\sum_{x=2^{k-1}}^{2^k-1}\left(\tfrac{1}{3x}-\tfrac{1}{6x^2}+\tfrac{1}{2x^3}\right) .
\end{aligned} \tag{11.24}$$

In the same way, by using (11.23) we get:

$$\begin{aligned}
\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) \geq\ & 2^{2k-3}\left(3k+1-\tfrac{3}{2\ln(2)}\right) + 2^{k-2}(k+1) \\
& + \tfrac{1}{4\ln(2)}\textstyle\sum_{x=2^{k-1}}^{2^k-1}\left(\tfrac{1}{3x}-\tfrac{2}{3x^2}\right) .
\end{aligned} \tag{11.25}$$

Let us have a closer look at the two functions $g_1(x) = \frac{1}{3x} - \frac{1}{6x^2} + \frac{1}{2x^3}$ and $g_2(x) = \frac{1}{3x} - \frac{2}{3x^2}$. If we analyze their first derivatives, we see that $g_1(x)$ is decreasing for $x \geq 1$ and $g_2(x)$ is decreasing for $x \geq 4$. For the upper bound of the sum, we can write directly:

$$\begin{aligned}
\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) &\leq 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right) + 2^{k-2}(k+1) \\
&\quad + \frac{1}{4\ln(2)}\sum_{x=2^{k-1}}^{2^k-1}\left(\frac{1}{3\ 2^{k-1}} - \frac{1}{6\ 2^{2k-2}} + \frac{1}{2\ 2^{3k-3}}\right) \\
&= 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right) + 2^{k-2}(k+1) + \frac{1-2^{-k}+3\ 2^{1-2k}}{12\ln(2)}
\end{aligned}$$

for all $k \geq 1$. In the case of the lower bound, we can write:

$$\begin{aligned}
\sum_{x=2^{k-1}+1}^{2^k} x \log_2(x) &\geq 2^{2k-3}\left(3k + 1 - \frac{3}{2\ln(2)}\right) + 2^{k-2}(k+1) \\
&\quad + \frac{1}{4\ln(2)} \sum_{x=2^{k-1}}^{2^k-1} \left(\frac{1}{3\,2^k} - \frac{2}{3\,2^{2k-2}}\right) \\
&= 2^{2k-1}\left(k - \frac{1}{2\ln(2)}\right) - 2^{k-1}k + \frac{4 + k + 2^{-k+1}}{24\ln(2)}
\end{aligned}$$

for $k \geq 3$. However, we can verify by numeric computation that the lower bound also holds in the cases $k = 1$ and $k = 2$. Thus, we have shown (11.18) and (11.19) for $k \geq 1$. Finally, by employing:

$$\sum_{x=1}^{2^K-1} x \log_2(x) = \sum_{k=1}^{K} \sum_{x=2^{k-1}+1}^{2^k} x \log_2(x) - K2^K$$

and the previous results, we receive the bounds (11.20) and (11.21).

11.6 Conclusion

If we allow all initial states of the FCSR with the same probability $2^{-n-\ell}$, we have an initial entropy of the state of $n + \ell$ bits. We showed in this chapter that already after one iteration, the entropy is reduced to $n + \ell/2$ bits.

As soon as the FCSR has reached its periodic behavior, the entropy does not reduce any more. In this chapter, we presented an algorithm which computes this final entropy in $\mathcal{O}(n^2)$ steps. The algorithm is exact if the results of the sums $S_1(k) = \sum_{x=2^{k-1}+1}^{2^k} x \log_2(x)$ and $S_2(k) = \sum_{x=1}^{2^k-1} x \log_2(x)$ are known for $k \leq \ell$. For large values of ℓ, the same algorithm allows us to give close upper and lower bounds for the entropy by using approximations of $S_1(k)$ and $S_2(k)$.

In the end, we used the same algorithm to prove that the final state entropy never drops under n bits. One might argue that this is evident, since there are q different values of h and $\log_2(q) \approx n$. However, it would be possible that the probabilities are not very regular distributed and that we would have a lower entropy. With our bound, it is sure that the entropy cannot drop under n bits.

The entropy of an FCSR decreases quite fast. However, it stays always larger or equal to n bits.

Chapter 12

Parallel generation of ℓ-sequences

The generation of pseudo-random sequences at a high rate is an important issue in modern communication schemes. The representation of a sequence can be scaled by decimation to obtain parallelism and more precisely a sub-sequences generator. Sub-sequences generators and therefore decimation have been extensively used in the past for linear feedback shift registers (LFSRs). However, the case of automata with a non linear feedback is still open. In this chapter, we study how to transform a feedback with carry shift register (FCSR) into a sub-sequences generator. We examine three solutions for this transformation. The first one is based on the decimation properties of ℓ-sequences, *i.e.* FCSR sequences with maximal period, the second one is based on multiple steps implementation and the last one consists in implementing an 2^d-adic FCSR with $d > 1$. We show that the solution based on the decimation properties leads to much more costly results than in the case of LFSRs. For the multiple steps implementation, we show how the propagation of carries affects the design. The use of 2^d-adic FCSRs is easy to design but might increase the size of the register.

This work represents a cooperation with Cédric Lauradoux and was presented at the international conference on SEquences and Their Applications (SETA) 2008 [LR08].

12.1 Introduction

The synthesis of shift registers consists in finding the smallest automaton able to generate a given sequence. This problem has many applications in cryptography, sequences and electronics. The synthesis of a single sequence with the smallest linear feedback shift register is achieved by the Berlekamp-Massey [Mas69] algorithm. There exists also an equivalent of Berlekamp-Massey in the case of multiple sequences [FT91, SS06]. In the case of FCSRs, we can use algorithms based on lattice approximation [KG97] or on Euclid's algorithm [ABN04]. For AFSRs we can apply the general synthesis algorithm of Klapper and Xu [KX04]. This chapter addresses the following issue in the synthesis of shift registers: *given an automaton generating a sequence $\mathcal{S}$, how to find an automaton which generates in parallel the sub-sequences associated to $\mathcal{S}$.* Throughout this chapter, we will refer to

this problem as *the sub-sequences generator problem.* We aim to find the best solution to transform a 1-bit output pseudo-random generator into a multiple outputs generator. In particular, we investigate this problem when $\mathcal{S}$ is generated by a feedback with carry shift register (FCSR) with a maximal period, *i.e.* $\mathcal{S}$ is an ℓ-sequence.

The design of sub-sequences generators has been investigated in the case of LFSRs [LE71, SC88] and two solutions have been proposed. The first solution [Rue86, Fil00] is based on the classical synthesis of shift registers, *i.e.* the Berlekamp-Massey algorithm, to define each sub-sequence. The second solution [LE71] is based on a multiple steps design of the LFSR. We have applied those two solutions to FCSRs. Our contributions are as follows:

- We explore the decimation properties of ℓ-sequences for the design of a sub-sequences generator by using an FCSR synthesis algorithm.

- We show how to implement a multiple steps FCSR in Fibonacci and Galois configuration.

- We compare the two previous methods with an implementation of an 2^d-adic FCSR where $d > 1$ is the decimation factor.

The next section presents the motivation of this work. In Section 12.3, the existing results on LFSRs are described and we show multiple steps implementations of the Galois and the Fibonacci configuration. We describe in Section 12.4 our main results on the synthesis of sub-sequences generators in the case of ℓ-sequences. Then, we give some conclusions in Section 12.5.

12.2 Motivation

The decimation is the main tool to transform a 1-bit output generator into a sub-sequences generator. An example for a 2–decimation can be found in Figure 12.1. The decimation

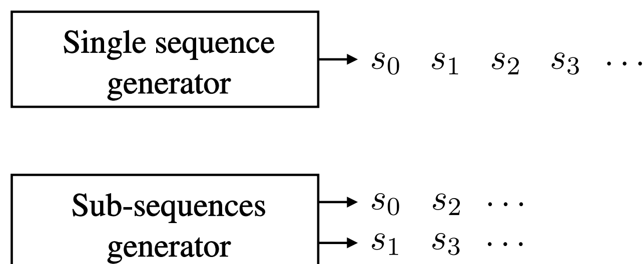


Figure 12.1: Model of a sub-sequences generator.

allows us to increase the throughput of a pseudo-random sequence generator (PRSG). Let $\mathcal{S} = (s_0, s_1, s_2, \cdots)$ be an infinite binary sequence of period T, thus $s_j \in \{0, 1\}$ and

$s_{j+T} = s_j$ for all $j \geq 0$. For a given integer d, a *d–decimation* of $\mathcal{S}$ is the set of sub-sequences defined by:

$$\mathcal{S}_d^i = (s_i, s_{i+d}, s_{i+2d}, \cdots, s_{i+jd}, \cdots)$$

where $i \in [0, d-1]$ and $j = 0, 1, 2, \cdots$. Hence, a sequence $\mathcal{S}$ is completely described by the sub-sequences:

$$\begin{array}{ccc} \mathcal{S}_d^0 & = & (s_0, s_d, \cdots) \\ \mathcal{S}_d^1 & = & (s_1, s_{1+d}, \cdots) \\ \vdots & \vdots & \vdots \\ \mathcal{S}_d^{d-2} & = & (s_{d-2}, s_{2d-2}, \cdots) \\ \mathcal{S}_d^{d-1} & = & (s_{d-1}, s_{2d-1}, \cdots) \, . \end{array}$$

A single automaton is often used to generate the pseudo-random sequence $\mathcal{S}$. In this case, it is difficult to achieve parallelism. The decomposition into sub-sequences overcomes this issue as shown by Lempel and Eastman in [LE71]. Each sub-sequence is associated to an automaton. Then, the generation of the d sub-sequences of $\mathcal{S}$ uses d automata which operate in parallel. Parallelism has two benefits, it can increase the throughput or reduce the power consumption of the automaton generating a sequence.

Throughput — The throughput $\mathcal{T}$ of a PRSG is defined by: $\mathcal{T} = n \times f$, with n is the number of bits produced every cycle and f is the clock frequency of the PRSG. Usually, we have $n = 1$, which is often the case with LFSRs. The decimation achieves a very interesting tradeoff for the throughput: $\mathcal{T}_d = d \times \gamma f$ with $0 < \gamma \leq 1$ the degradation factor of the original automaton frequency. The decimation provides an improvement of the throughput if and only if $\gamma d > 1$. It is then highly critical to find good automata for the generation of the sub-sequences. In an ideal case, we would have $\gamma = 1$ and then a d-decimation would imply a multiplication of the throughput by d.

Power consumption — The power consumption of a CMOS device can be estimated by the following equation: $P = C \times V_{dd}^2 \times f$, with C the capacity of the device and V_{dd} the supply voltage. The sequence decimation can be used to reduce the frequency of the device by interleaving the sub-sequences. The sub-sequences generator will be clocked at frequency $\frac{\gamma f}{d}$ and the outputs will be combined with a d-input multiplexer clocked at frequency γf. The original power consumption can then be reduced by the factor $\frac{\gamma}{d}$, where γ must be close to 1 to guarantee that the final representation of $\mathcal{S}$ is generated at frequency f.

The study of the γ parameter is out of the scope of this work since it is highly related to the physical characteristics of the technology used for the implementation. In the following, we consider m-sequences and ℓ-sequences which are produced respectively by LFSRs and FCSRs.

Throughout the chapter, we detail different representations of several automata. We denote by (a_i) a memory cell and by $a_i(t)$ the content of the cell (a_i) at time t. The internal state of an automaton at time t is denoted by $A(t)$, $(A(t), m(t))$ and $(A(t), C(t))$, respectivley, for a LFSR, a Fibonacci and a Galois FCSR.

12.3 Sub-Sequences Generators and m-Sequences

The decimation of LFSR sequences has been used in cryptography in the design of new stream ciphers [MR84]. There exist two approaches to use decimation theory to define the automata associated to the sub-sequences.

Construction using LFSR synthesis. This first solution associates an LFSR to each sub-sequence (*cf.* Figure 12.2). This construction is based on well-known results on the

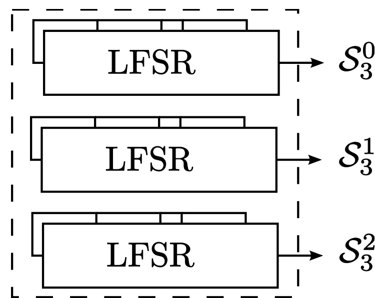


Figure 12.2: Sub-sequences generator using LFSR synthesis.

decimation of LFSR sequences. It can be applied to both Fibonacci and Galois representation without any distinction.

Theorem 12.1 ([Zie59, Rue86]) *Let $\mathcal{S}$ be a sequence produced by an LFSR whose characteristic polynomial $Q(X)$ is irreducible in $\mathbb{F}_2$ of degree n. Let α be a root of $Q(X)$ and let T be the period of $Q(X)$. Let $\mathcal{S}_d^i$ be a sub-sequence resulting from the d-decimation of $\mathcal{S}$. Then, $\mathcal{S}_d^i$ can be generated by an LFSR with the following properties:*

- *The minimum polynomial of α^d in $\mathbb{F}_{2^n}$ is the characteristic polynomial $Q^\star(X)$ of the resulting LFSR.*
- *The period $T^\star$ of $Q^\star(X)$ is equal to $\frac{T}{gcd(d,T)}$.*
- *The degree $n^\star$ of $Q^\star(X)$ is equal to the multiplicative order of $Q(X)$ in $\mathbb{Z}_{T^\star}$.*

In practice, the characteristic polynomial $Q^\star(X)$ can be determined using the Berlekamp-Massey algorithm [Mas69]. The sub-sequences are generated using d LFSRs defined by the characteristic polynomial $Q^\star(X)$ but initialized with different values. In the case of LFSRs, the degree n^* must always be smaller or equal to n.

Construction using a multiple steps LFSR. This method was first proposed by Lempel and Eastman [LE71]. It consists in clocking the LFSR d times in one clock cycle by changing the connections between the memory cells and by some duplications of the feedback function. We obtain a network of linearly interconnected shift registers. This

method differs for Galois and Fibonacci setup. Let $next^d(a_i)$ denote the cell connected to the output of (a_i). For an n-bit Fibonacci LFSR with transition function f, the transformation into an automaton which generates d bits per cycle is achieved using the following equations:

$$next^d(a_i) = (a_{i-d \bmod n}) \tag{12.1}$$

$$a_i(t+d) = \begin{cases} f\Big(A(t+i-n+d)\Big) & \text{if } n-d \leq i < n \\ a_{i+d}(t) & \text{if } i < n-d \end{cases} \tag{12.2}$$

Equation 12.1 corresponds to the transformation of the connections between the memory cells. All the cells (a_i) of the original LFSR, such that $i \bmod d = k$, are gathered to form a sub-shift register, where $0 \leq k \leq d-1$. This is the basic operation to transform a LFSR into a sub-sequences generator with a multiple steps solution. The content of the last cell of the k-th sub-shift registers corresponds to the k-th sub-sequence $\mathcal{S}_d^k$. Equation 12.2 corresponds to the transformation of the feedback function. It must be noticed that the synthesis requires to have only relations between the state of the register at time $t+d$ and t. Figure 12.3 shows an example of such a synthesis for a Fibonacci setup defined by the connection polynomial $q(X) = X^8 + X^5 + X^4 + X^3 + 1$ with the decimation factor $d = 3$. The transformation of a Galois setup is described by the Equations 12.1 and 12.3:

$$a_i(t+d) = \begin{cases} a_0(t+d-n+i) \oplus \bigoplus_{k=0}^{n-2-i} q_{i+k+1} a_0(t+d-k-1) & \text{if } n-d \leq i < n \\ a_{i+d}(t) \oplus \bigoplus_{k=0}^{d-1} q_{i+d-k} a_0(t+k) & \text{if } i < n-d \end{cases} \tag{12.3}$$

with $q(X) = 1 + q_1X + q_2X^2 + \cdots + q_{n-1}X^{n-1} + X^n$. The Equation 12.3 does not provide a direct relation between the state of the register at time $t+d$ and t. However, this equation can be easily derived to obtain more practical formulas as shown in Figure 12.4.

Comparison. We have summarized in the Table 12.1 the two methods used to synthesize the sub-sequences generator. By $wt(Q(X))$, we mean the Hamming weight of the characteristic polynomial Q, *i.e.* the number of non-zero monomials. The method based on LFSR synthesis proves that there exists a solution for the synthesis of the sub-sequences generator. With this solution, both memory cost and gate number depends on the decimation factor d. The method proposed by Lempel and Eastman [LE71] uses a constant number of memory cells for the synthesis of the sub-sequences generator.

Method	Memory cells	Logic Gates
LFSR synthesis [Mas69]	$d \times n^\star$	$d \times wt(Q^\star)$
Multiple steps LFSRs [LE71]	n	$d \times wt(Q)$

Table 12.1: Comparison of the two methods for the synthesis of a sub-sequences generator.

$$
\begin{aligned}
&next^1(a_0) = (a_7)\\
&next^1(a_i) = (a_{i-1}) \text{ if } i \neq 0\\
&a_7(t+1) = a_0(t) \oplus a_3(t) \oplus a_4(t) \oplus a_5(t)\\
&a_i(t+1) = a_{i+1}(t) \text{ if } i \neq 7
\end{aligned}
$$

$$
\begin{aligned}
&next^3(a_0) = (a_5)\\
&next^3(a_1) = (a_6)\\
&next^3(a_2) = (a_7)\\
&next^1(a_i) = (a_{i-3}) \text{ if } i > 2\\
&a_5(t+3) = a_0(t) \oplus a_3(t) \oplus a_4(t) \oplus a_5(t)\\
&a_6(t+3) = a_1(t) \oplus a_4(t) \oplus a_5(t) \oplus a_6(t)\\
&a_7(t+3) = a_2(t) \oplus a_5(t) \oplus a_6(t) \oplus a_7(t)\\
&a_i(t+3) = a_{i+3}(t) \text{ if } i < 5
\end{aligned}
$$

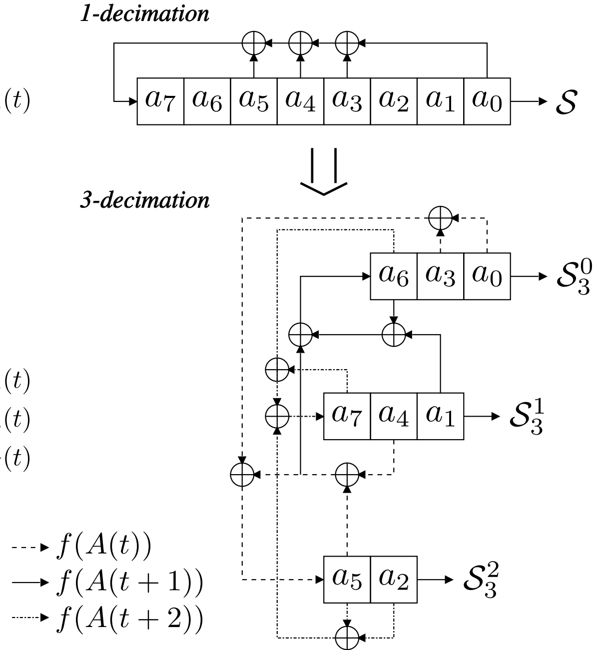


Figure 12.3: Multiple steps generator for a Fibonacci LFSR.

The sub-sequences generators defined with the Berlekamp-Massey algorithm are not suitable to reduce the power consumption of an LFSR. Indeed, d LFSRs will be clocked at frequency $\frac{\lambda f}{d}$ to produce the sub-sequences, however, the power consumption of such a sub-sequence generator is given by:

$$
\begin{aligned}
P &= d \times \left(C_d \times V_{dd}^2 \times \frac{\gamma f}{d} \right)\\
&= \lambda C \times V_{dd}^2 \times \gamma f
\end{aligned}
$$

with C and $C_d = \lambda C$ the capacity of LFSRs corresponding, respectively, to $\mathcal{S}$ and $\mathcal{S}_d^i$. We can achieve a better result with a multiple steps LFSR:

$$
P = \lambda' C \times V_{dd}^2 \times \frac{\gamma f}{d}
$$

with $C_d = \lambda' C$.

12.4 Sub-Sequences Generators and ℓ-Sequences

This section presents our main contribution. We apply the two methods used in the previous section to the case of ℓ-sequences. In addition, we consider the implementation of an 2^d-adic FCSR.

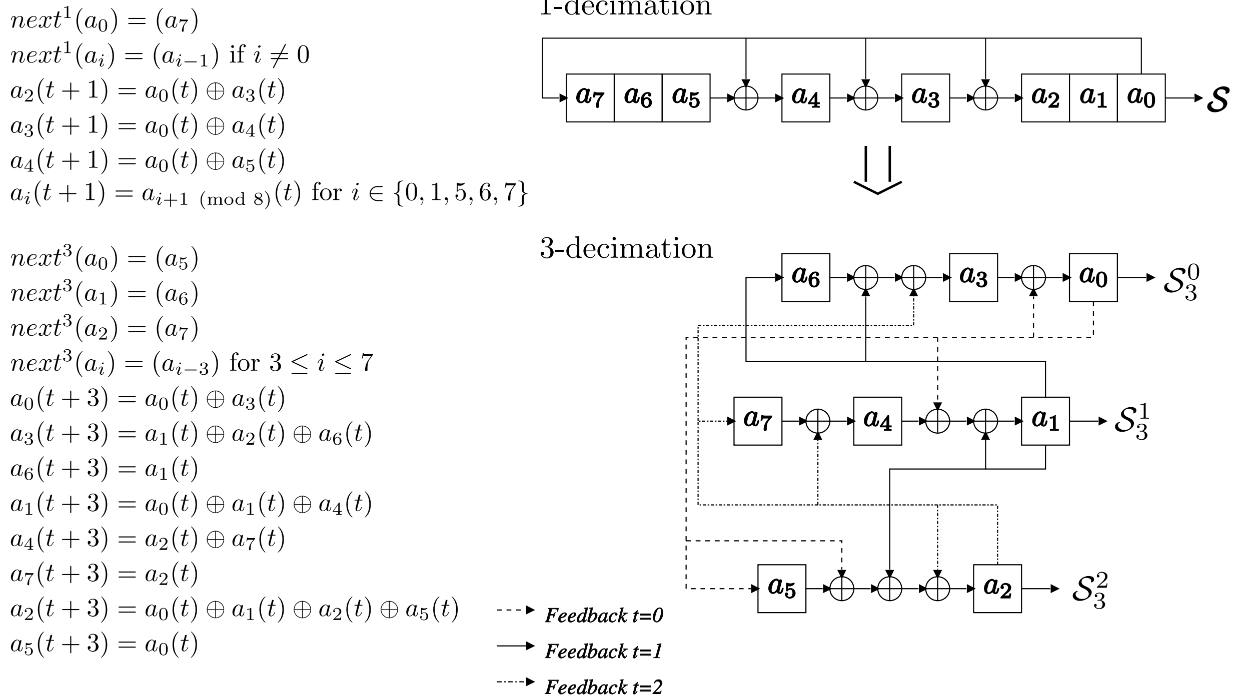


Figure 12.4: Multiple steps generator for a Galois LFSR.

Construction using FCSR synthesis. There exist algorithms based on Euclid's algorithm [ABN04] or on lattice approximation [GK97], which can determine the smallest FCSR to produce $\mathcal{S}_d^i$. These algorithms use the first k bits of $\mathcal{S}_d^i$ to find $h^\star$ and $q^\star$ such that $h^\star/q^\star$ is the 2-adic representation of the sub-sequence, $-q^\star < h^\star \leq 0$ and $\gcd(q^\star, h^\star) = 1$. Subsequently, we can find the feedback positions and the initial state of the FCSR in Galois or Fibonacci architecture. The value k is in the range of twice the 2-adic complexity of the sequence. For our new sequence $\mathcal{S}_d^i$, let $h^\star$ and $q^\star$ define the values found by one of the algorithms mentioned above. By $T^\star$ and T, we mean the periods of respectively $\mathcal{S}_d^i$ and $\mathcal{S}$.

For the period of the decimated sequences, we can make the following statement, which is true for all periodic sequences.

Lemma 12.2 *Let $\mathcal{S} = (s_0, s_1, s_2, \dots)$ be a periodic sequence with period T. For a given $d > 1$ and $0 \leq i \leq d-1$, let $\mathcal{S}_d^i$ be the decimated sequence with period $T^\star$. Then, it must hold:*

$$T^\star \left| \frac{T}{\gcd(T,d)} \right. . \tag{12.4}$$

If $\gcd(T,d) = 1$ then $T^\star = T$.

Proof.
The first property is given by:

$$s_{d[j+T/gcd(T,d)]+i} = s_{dj+i+T[d/\gcd(T,d)]} = s_{dj+i} .$$

For the case $\gcd(T, d) = 1$, there exits $x, y \in \mathbb{Z}$ such that $xT + yd = 1$ due to Bezout's lemma. Since $\mathcal{S}$ is periodic, we define $s_j = s_k$ for any $j < 0$ and $k \geq 0$, if $j \pmod T = k$. Thus, we can write for any j:

$$\begin{aligned} s_j &= s_{i+(j-i)xT+(j-i)yd} &= s_{i+(j-i)yd} \,, \\ s_{j+T^\star} &= s_{i+(j-i)xT+T^\star xT+(j-i)yd+T^\star yd} &= s_{i+(j-i)yd+T^\star yd} \,. \end{aligned}$$

However, since $T^\star$ is the period of $\mathcal{S}_d^i$ we get:

$$s_{j+T^\star} = s_{j+(j-i)yd} = s_j \,.$$

Therefore, it must hold that $T|T^\star$ which together with (12.4) proves that $T^\star = T$ if $\gcd(T, d) = 1$. ◇

In case $\gcd(T, d) > 1$, *the real value of $T^\star$ might depend on i, e.g.* if $\mathcal{S}$ is the 2-adic representation of $-1/19$ and $d = 3$ we have $T/gcd(T, d) = 6$, however, for $\mathcal{S}_3^0$ the period $T^\star = 2$ and for $\mathcal{S}_3^1$ the period $T^\star = 6$.

A critical point in this approach is that the size of the new FCSR can be exponentially bigger than the original one. In general, we only know that for the new $q^\star$ it must hold that $q^\star | 2^{T^\star} - 1$. From the previous paragraph we know that $T^\star$ can be as big as $T/gcd(T, d)$. In the case of an *allowable decimation* [GKMS04], *i.e.* a decimation where d and T are coprime, we have more information:

Corollary 12.3 ([GK97]) *Let $\mathcal{S}$ be the 2-adic representation of h/q, where $q = p^e$ is a prime power with p prime, $e \geq 1$ and $-q \leq h \leq 0$. Let $d > 0$ be relatively prime to $T = p^e - p^{e-1}$, the period of $\mathcal{S}$. Let $\mathcal{S}_d^i$ be a d-decimation of $\mathcal{S}$ with $0 \leq i < d$ and let $h^\star/q^\star$ be its 2-adic representation such that $-q^\star \leq h^\star \leq 0$ and $\gcd(q^\star, h^\star) = 1$. Then $q^\star$ divides*

$$2^{T/2} + 1 \,.$$

If the following conjecture is true, we have more information on the new value $q^\star$.

Conjecture 12.4 ([GK97]) *Let $\mathcal{S}$ be an ℓ-sequence with connection number $q = p^e$ and period T. Suppose p is prime and $q \notin \{5, 9, 11, 13\}$. Let d_1 and d_2 be relatively prime to T and incongruent modulo T. Then for any i and j, $\mathcal{S}_{d_1}^i$ and $\mathcal{S}_{d_2}^j$ are cyclically distinct,* i.e. *there exists no $\tau \geq 0$ such that $\mathcal{S}_{d_1}^i$ is equivalent to $\mathcal{S}_{d_2}^j$ shifted by τ positions to the left.*

This conjecture has already been proved for many cases [GKMS04, XQ06, TQ09] but not yet for all. Recently, Bourgain *et al.* showed that the conjecture is valid for all $q > 1.09 \times 2^{184}$ [BCPP09]. If it holds, this implies that for any $d > 1$, $\mathcal{S}_d^i$ is cyclically distinct from our original ℓ-sequence. We chose q such that the period was maximal, thus, any other sequence with the same period which is cyclically distinct must have a value $q^\star > q$. This means that the complexity of the FCSR producing the subsequence $\mathcal{S}_d^i$ will be larger than the original FCSR, *if d and T are relative prime.*

Remark 12.5 *Let us consider the special case where q is prime and the period $T = q - 1 = 2p$ is twice a prime number $p > 2$, as recommended for the stream cipher proposed in [AB05a]. The only possibilities in this case for $gcd(d, T) > 1$ is $d = 2$ or $d = T/2$.*

For $d = T/2$, we will have $T/2$ FCSRs where each of them outputs either 0101... *or* 1010..., *since for an ℓ-sequence the second half of the period is the inverse of the first half [GK97]. Thus, the size of the sub-sequences generator will be in the magnitude of T which is exponentially bigger than the 2-adic complexity of $\mathcal{S}$ which is $\log_2(q)$.*

In the case of $d = 2$, we get two new sequences with period $T^\star = p$. As for any FCSR, it must hold that $T^\star | ord_{q^\star}(2)$, where $ord_{q^\star}(2)$ is the multiplicative order of 2 modulo $q^\star$. Let $\varphi(n)$ denote Euler's function, i.e. *the number of integers smaller than n which are relative prime to n. It is well known,* e.g. *[McE87], that $ord_{q^\star}(2)|\varphi(q^\star)$ and if $q^\star$ has the prime factorization $p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$ then $\varphi(q^\star) = \prod_{i=1}^{r} p_i^{e_i - 1}(p_i - 1)$. From this follows that $p \neq \varphi(q^\star)$, because otherwise $(p+1)$ must be a prime number, which is not possible since $p > 2$ is a prime. We also know that $T^* = p|\varphi(q^*)$, thus $2 \times p = q - 1 \leq \varphi(q^*)$. This implies that $q^\star > q$, since from $T^\star = T/2$ follows that $q \neq q^\star$.*

Together with Conjecture 12.4, we obtain that for such an FCSR any decimation would have a larger complexity than the original one. This is also interesting from the perspective of stream ciphers, since any decimated subsequence of such an FCSR has larger 2-adic complexity than the original one, except for the trivial case with $d = T/2$.

Construction using a multiple steps FCSR. A multiple steps FCSR is a network of interconnected shift registers with a carry path: the computation of the feedback at time t depends directly on the carry generated at time $t - 1$. The transformation of an n-bit FCSR into a d sub-sequences generator uses first Equation (12.1) to modify the mapping of the shift register. For the Fibonacci setup, let $g\Big(A(t), m(t)\Big) = h\Big(A(t)\Big) + m(t)$ denote the feedback function of the FCSR with

$$h\Big(A(t)\Big) = \sum_{i=0}^{n-1} q_{n-i} a_i(t) \ . \tag{12.5}$$

Then, the transformation uses the following equations:

$$a_i(t+d) = \begin{cases} \text{if } n-d \leq i < n \ : \\ g\Big(A(t+d-m+i), m(t+d-m+i)\Big) \bmod 2 \\ \\ \text{if } i < n-d \ : \\ a_{i+d}(t) \end{cases} \tag{12.6}$$

$$m(t+d) = g\Big(A(t+d-1), m(t+d-1)\Big)/2 \tag{12.7}$$

Due to the nature of the function g, we can split the automaton into two parts. The first part handles the computation related to the shift register $A(t)$ and the other part is the carry path as shown in Figure 12.5 for $q = 347$.

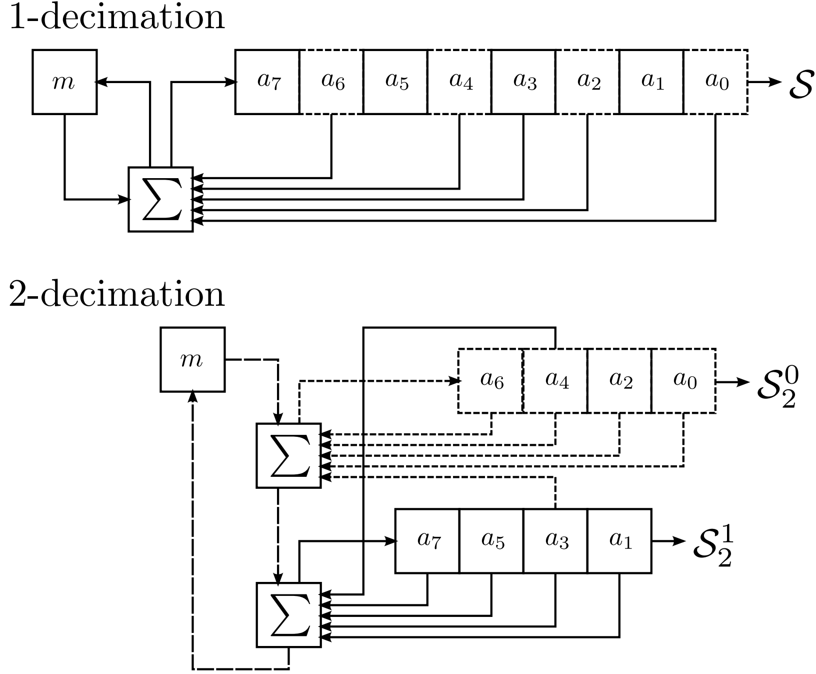


Figure 12.5: Multiple steps generator for a Fibonacci FCSR.

The case of Galois FCSRs is more difficult because the circuit can not be split into two parts: each bit of the carry register must be handle separately. The modification of the basic operator of a Galois FCSR, *i.e.* addition with carry, is the key transformation to obtain a sub-sequences generator. Let us consider a Galois FCSR with $q = 19$. This automaton has a single addition with carry as shown in Figure 12.6(a). The sub-sequences generator for $d = 2$ associated to this FCSR is defined by:

$$t+1 \begin{cases} a_1(t+1) &= \Big(a_0(t) + a_2(t) + c_2(t)\Big) \bmod 2 \\ c_2(t+1) &= \Big(a_0(t) + a_2(t) + c_2(t)\Big) \operatorname{div} 2 \end{cases} \tag{12.8}$$

$$t+2 \begin{cases} a_1(t+2) &= \Big(a_0(t+1) + a_3(t) + c_2(t+1)\Big) \bmod 2 \\ c_2(t+2) &= \Big(a_0(t+1) + a_3(t) + c_2(t+1)\Big) \operatorname{div} 2 \end{cases} \tag{12.9}$$

with c_1 the carry bit of the FCSR.

For the implementation of the multiple steps FCSR, we are interested in the description at the bit-level. An addition with carry (represented by ⊞ in the figures) can be done by 5 XORs and one multiplications (logic AND). This operator is also known as a full adder.

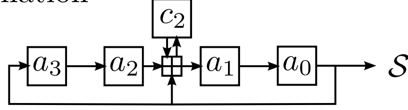
Thus on bit-level, the previous equation can be expressed as follows:

$$t+1 \left\{ \begin{array}{lcl} a_1(t+1) & = & a_0(t) \oplus a_2(t) \oplus c_2(t) \\ c_2(t+1) & = & [a_0(t) \oplus a_2(t)]\,[a_0(t) \oplus c_2(t)] \oplus a_0(t) \end{array} \right. \tag{12.10}$$

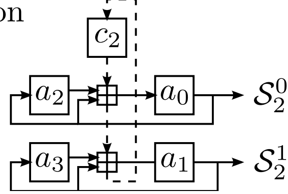
$$t+2 \left\{ \begin{array}{lcl} a_1(t+2) & = & a_0(t+1) \oplus a_3(t) \oplus c_2(t+1) \\ c_2(t+2) & = & [a_0(t+1) \oplus a_3(t)]\,[a_0(t+1) \oplus c_2(t+1)] \oplus a_0(t+1) \end{array} \right. \tag{12.11}$$

The equations corresponding to time $t+2$ depend on the carry, $c_1(t+1)$, generated at time $t+1$. This dependency between full adders is a characteristic of a well-known arithmetic circuit: the k-bit ripple carry adder (Figure 12.6(b)).

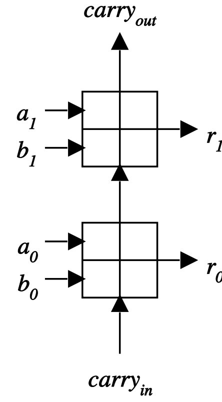
1-decimation



2-decimation



(a) Galois FCSR for $q = 19$.



(b) 2-bit ripple carry adder.

Figure 12.6: Example for a Galois FCSR with $q = 19$.

Thus, all the full adders in a d sub-sequences generator are replaced by d-bit ripple carry adders as shown in Figure 12.7.

We can derive a more general representation of a multiple steps Galois FCSR from the previous formula:

$$a_i(t{+}d) = \left\{ \begin{array}{l} \text{if } n-d \leq i < n \ : \\ a_0(t+d-m+i) \oplus \bigoplus_{k=0}^{n-2-i} q_{i+k+1}\,[a_0(t+d-k-1) \oplus c_{i+k+1}(t+d-k-1)] \\ \\ \text{if } i < n-d \ : \\ a_{i+d}(t) \oplus \bigoplus_{k=0}^{d-1} q_{i+d-k}\,[a_0(t+k) \oplus c_{i+d-k}(t+k)] \end{array} \right. \tag{12.12}$$

$$c_i(t+d) = [a_0(t+d-1) \oplus a_i(t+d-1)]\,[a_0(t+d-1) \oplus c_i(t+d-1)] \oplus a_0(t+d-1)\,. \tag{12.13}$$

Equation (12.13) shows the dependencies between the carries which corresponds to the propagation of the carry in a ripple carry adder. Equation (12.12) corresponds to the path of the content of a memory cell $a_i(t)$ through the different levels of the ripple carry adders.

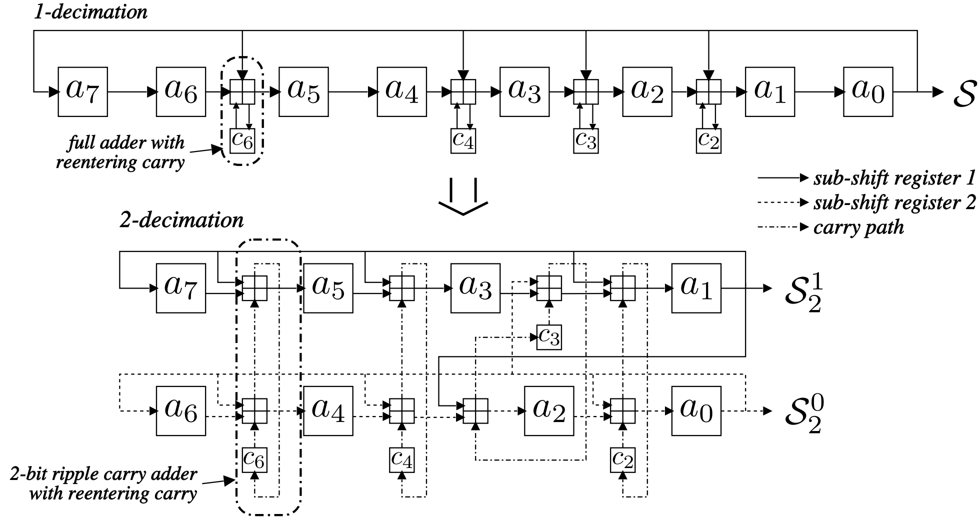


Figure 12.7: Multiple steps generator for a Galois FCSR.

Implementation as an 2^d-adic FCSR. Let $d > 1$ be the decimation factor. Until now, we saw the output sequence $\mathcal{S}$ as an 2-adic representation of a negative rational number $\frac{h}{q}$ and we used 2-adic FCSRs. In the same way, we can see $\mathcal{S}, h, q$ as a 2^d-adic numbers and use 2^d-adic FCSRs. Practically all theorems of binary FCSRs can be directly applied to the 2^d-adic case [KX89]. In such an FCSR, every cell in the main register can have a value in $\mathbb{Z}_{2^d}$ and can thus be represented by d bits. We can see the FCSR as d interconnected sub-register. The connection integer q is defined like in (10.14) as

$$q = -1 + \sum_{i=1}^{n} q_i (2^d)^i ,$$

however, this time $q_i \in \mathbb{Z}_{2^d}$.

The main advantage of the 2^d-adic implementation is that it is easy to design. For a given 2-adic FCSR with a connection integer q we can directly obtain the feedback position of its 2^d-adic counterpart. Though, there are some small disadvantages:

- We have to implement the multiplications by $q_i \in \mathbb{Z}_{2^d}$.
- If $q \not\equiv -1 \pmod{2^d}$ we cannot use the original q and h. However, we can use $h^\star = x \times h$ and $q^\star = x \times q$ with $x > 0$ and $q^\star \equiv -1 \pmod{2^d}$. Then $\frac{h^\star}{q^\star} = \frac{h}{q}$ will produce the same output sequence $\mathcal{S}$.
- The size of the main register is a multiple of d-bits. This might change the size of the resulting FCSR.

To illustrate the method, let us consider the example of an 2-adic Galois FCSR with connection integer $q = 37$ and a $d = 2$ decimation. Since $37 \equiv 1 \pmod{3}$, we have to

change the the connection integer to $q^\star = 111$. This means, that we have to multiply all values h by 3 to obtain $h^\star$. The resulting 4-adic FCSR for $q^\star$ can be found in Figure 12.8(b). It has in total six bits in the main register and two bits in the carry register and we need one multiplication by three.

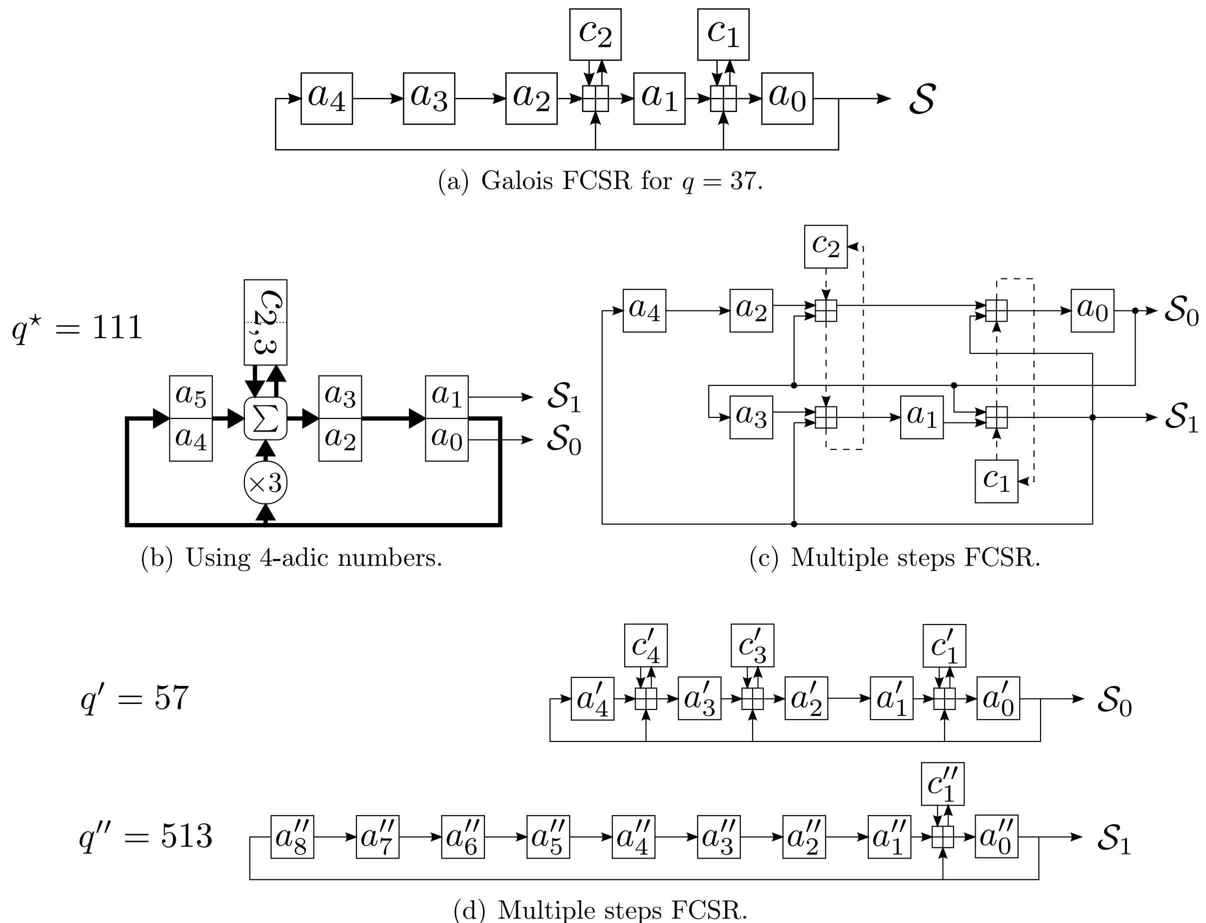


Figure 12.8: FCSR synthesis.

Comparison. The construction using FCSR synthesis is more complicated than in the case of LFSRs. The size of the minimal generator producing $\mathcal{S}_d^i$ can depend on i, and we can only give upper bounds for $q^\star$ namely that $q^\star | 2^{T^\star} - 1$ in the general case and that $q^\star | 2^{T/2} + 1$ if $\gcd(d, T) = 1$. Based on Conjecture 12.4, we saw that $q^\star > q$ if $\gcd(T, d) = 1$. Moreover, in the case $q = 2p + 1$ with p and q prime, the resulting sub-sequences generator is always larger than the original one.

Apart from the carry path and the cost of the addition with carry, the complexity of a multiple steps implementation of a FCSR is very similar to the multiple steps implementation of an LFSR. There is no overhead in memory and the number of logic gates for a

Galois FCSR is $5d \times wt(q)$ where $wt(q)$ is the number of ones in the binary representation of q and the number 5 corresponds to the five gates required for a full-adder (four XORs and one AND *cf.* Equation 12.10). In the case of a Fibonacci setup, the number of logic gates is given by:

$$d \times (5 \times (wt(q) - \lceil \log_2(wt(q)+1) \rceil) + 2 \times (\lceil \log_2(wt(q)+1) \rceil - wt(wt(q)) + 5 \times size(m))$$

with $(5 \times (wt(q) - \lceil \log_2(wt(q)+1) \rceil) + 2 \times (\lceil \log_2(wt(q)+1) \rceil - wt(wt(q)))$ the cost of the implementation of a parallel bit counter [MP75], *i.e.* the h function (*cf.* Equation 12.5) and $5 \times size(m)$ is the cost of a ripple carry adder which adds the content of $size(m)$ bits of the carry with the output of h.

The implementation of an 2^d-adic FCSR is easy to design. However, the result might be less efficient than a multiple steps FCSR. The size of the 2^d-adic main register will always be a multiple of d bits. Sometimes we have to multiply the connection integer with a constant factor to achieve that $q^\star \equiv -1 \pmod{2^d}$. This has also an influence of the initial value of the FCSR since we have to change h to $h^\star$. In the case of 2^d-adic numbers, we have not only to implement the addition modulo 2^d but also the multiplications for all $q_i \in \mathbb{Z}_{2^d}$ with $1 \leq i \leq n$.

In Figure 12.4, we compare all three methods for an Galois FCSR with connection integer $q = 37$ and a $d = 2$ decimation. In this case $\gcd(T, d) > 1$.

12.5 Conclusion

We have presented in this chapter how to transform an FCSR into a sub-sequences generator to increase its throughput or to reduce its power consumption. Our results emphasize the similarity between LFSRs and FCSRs in terms of implementation properties. In both cases, the solution based on the classical synthesis algorithm fails to provide a minimal solution. Even worse, in the case of FCSRs the memory needed is in practice exponentially bigger than for the original FCSR. Thus, we need to use multiple steps implementations as for LFSRs. The propagation of carries is the main problem in multiple steps implementations of FCSRs: if we consider d sub-sequences, we obtain d-bit ripple carry adders with carry instead of additions with carry in a Galois setup. In the case of a Fibonacci setup, the situation is different since we can split the feedback function into two parts. A third alternative is the use of 2^d-adic FCSRs. Their design is very simple, but in some cases they are less efficient than the multiple steps implementation. The last two parallelization methods can significantly improve the hardware implementation of FCSRs but it may also be possible to improve software implementations.

Conclusion and Perspectives

In this thesis, we considered the generation of (pseudo) random numbers for cryptography. In a first part, we examined the HAVEGE generator, which produces random numbers from the volatiles states in a processor. These states, like the cache or the branch predictor, are not observable without altering them and hard to control. This means that we are not able to produce the same sequence twice. We studied the quality of the data collected from the processor by considering different versions of the generator and different test settings. We could confirm the assumption that the collected entropy is highest if the generator exploits the uncertainty in the branch predictor and in the instruction cache. In addition, we applied an entropy estimator for Markov chains on the test data. We could see that the estimated entropy decreases if we consider higher orders Markov chains. However, the number of needed test data increases exponentially with the order of the MC. This effect limited our studies to order of size 4. We could not find any specific weakness of structure in HAVEGE which demanded further studies. Thus, we stopped our examination at this point.

In the rest of this thesis, we studied pseudo random number generators (PRNGs) which produce completely deterministic sequences from a short initial value. Particularly, we considered the case where these generators are used as stream ciphers. We examined a specific cipher (Dragon), a general model (using random functions), and possible components (FCSRs). Dragon is a 32-bit word based stream cipher which consists of non-linear shift register combined with a 64-bit counter. We studied several properties of the non-linear state update function. Amongst other things, we considered the bias of the linear approximations of combined output words. Finally, we were not able to find a better distinguisher than the two proposed by Englund/Maximov and Cho/Pieprzyk.

We also considered a general model of stream ciphers. This time we assumed that the state update function is realized by a random function and that the initial state is uniformly distributed. This can be seen as an attack model where an attacker has no information about the initial state or the update function, because they depend on the secret key and the IV. We were interested in how much entropy we lose in the state. In a first part, we found an approximation of the state entropy after several iterations of the random function. This was done by exploiting the structure of the functional graph of a random mapping. We gave an recursive formula of the approximation. In addition, we showed that for a number of iterations k up to a few hundreds, we are able to give an estimate of the entropy which is much better than the upper bound given by the number

of image points. Subsequently, we studied if this loss of entropy can be exploited for a collision search in the inner state of our stream cipher model. We showed that in the case of a random update function, such a search is not more efficient than a search for a collision directly in the initial states. The only way for these attacks to be efficient would be a family of functions which either lose more entropy or where we can compute the k'th iterate in less than k steps.

In the last big part of this thesis, we considered Feedback with Carry Shift Registers (FCSRs). These are fast and non-linear feedback shift register and have a structure similar to Linear Feedback Shift Registers (LFSRs). Where LFSRs can be described by the theory of polynomials, FCSRs uses the theory of 2-adic numbers. In a first part, we considered FCSRs in Galois setup. We assume that the initial values of all the bits in the state (main register and carry register) are independent and uniformly distributed. This means that the entropy of the initial state is the size of the main register plus the size of the carry register. The state update of a Galois FCSR is a non-injective function, thus, we lose entropy in the inner state. After a finite number of iterations, the entropy arrives at a stable value. This is true for any function which maps a finite set onto itself. We could show that in the case of a Galois FCSR, we lose already a lot of entropy after one iteration. We also gave an algorithm, which allows to compute the final entropy of any Galois FCSR. In addition, we could use this algorithm to prove that the final entropy can never decrease under the size of the main register. If we chose only the bits of the main register in the initial state, and set the bits of the carry register to all 0's we have an initial and final entropy of the size of the main register. Thus, we could show that concerning the entropy we are finally not worse if we allow to set all bits in the initial state.

Finally, we studied an aspect of the implementation of FCSRs. The original implementation of an FCSR generates the output bit per bit. We were interested in an implementation which allows to generate multiple bits in parallel. Inspired by the similarities between LFSRs and FCSRs, we tried the methods of parallelization which are already known for LFSRs. The first technique generates each subsequence by an independent shift register. This works fine for LFSRs. However, we showed that in the case of FCSRs, the new registers are in general much bigger than the original one. Thus, we cannot use this method to increase the efficiency of the register. The second approach partitions the original register in interconnected sub-registers and duplicates the feedback paths. In contrary to LFSRs, we have to consider the subsequent generation of the carry bits. However, based on components like the n-bit ripple carry adder, we are still able to provide an efficient parallelization of FCSR sequences.

Perspectives

In this Section, we mention a few question which we would like to examine further.

Random Mappings and Hash Functions

We considered in detail the stream cipher model using a random update function. It would be interesting to know if we can use the same approach for other primitives. Hash functions are an example of primitives which are currently in the center of attention. In October 2008, the National Institute of Standards and Technology (NIST) started a four year project to find a new standard of cryptographic hash functions [SHA]. In the context of hash functions, an important property is the probability of a collision or a multi-collision. We would like to know, if we are able to find multi-collisions based on the structure of random functions with a higher probability than the generalized birthday paradox [DM89, STKT06]. In connection with our stream cipher model, we already showed that it is not possible to find a single collision faster by iteration a random function several times. However, it might be possible to exploit the structure of the function graph in a different way to find multi-collisions efficiently.

FCSRs

In the area of FCSRs there are still a lot of open questions. Are there are other possible applications for FCSRs than stream ciphers? In this thesis we mainly considered binary Galois and Fibonacci FCSRs. However there exists a multitude of extensions, like d-FCSRs or the new architecture examined by Arnault, Berger, Minier, and Pousse which represents a hybrid of the Galois and the Fibonacci setup. Is it possible to apply our parallelization on these more general cases? Can we make any statements about the state entropy?

It was shown that neither a binary Galois FCSR nor a binary Fibonacci FCSR is optimal for the use in stream ciphers. The first one changes only one bit in the main register in each iteration [FMS08]. The second one has an almost linear behavior with a relative high probability [HJ08]. Does these attacks apply also for extensions of the FCSRs?

The family of FCSRs and its extensions are very large. They have the advantage that they are non-linear and supported by a strong theory. This leaves a lot of interesting questions in this area.

Bibliography

[AB82] James Arney and Edward A. Bender. Random mappings with constraints on coalescence and number of origins. *Pacific Journal of Mathematics*, 103(2):269–294, 1982.

[AB05a] François Arnault and Thierry P. Berger. Design and properties of a new pseudorandom generator based on a filtered FCSR automaton. *IEEE Trans. Computers*, 54(11):1374–1383, 2005.

[AB05b] François Arnault and Thierry P. Berger. F-FCSR: Design of a new class of stream ciphers. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption - FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2005.

[AB08] François Arnault and Thierry P. Berger. Correction to "feedback with carry shift registers synthesis with the euclidean algorithm" [may 04 910-917]. *IEEE Transactions on Information Theory*, 54(1):502, 2008.

[ABL05] François Arnault, Thierry Berger, and Cédric Lauradoux. Update on F-FCSR stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/025, 2005. `http://www.ecrypt.eu.org/stream`.

[ABL08] François Arnault, Thierry P. Berger, and Cédric Lauradoux. F-FCSR stream ciphers. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 170–178. Springer, 2008.

[ABLM07] François Arnault, Thierry P. Berger, Cédric Lauradoux, and Marine Minier. X-FCSR - a new software oriented stream cipher based upon FCSRs. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology - INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 341–350. Springer, 2007.

[ABM08] François Arnault, Thierry P. Berger, and Marine Minier. Some results on FCSR automata with applications to the security of FCSR-based pseudorandom generators. *IEEE Transactions on Information Theory*, 54(2):836–840, 2008.

[ABN02] François Arnault, Thierry P. Berger, and Abdelkader Necer. A new class of stream ciphers combining lfsr and FCSR architecture. In Alfred Menezes and Palash Sarkar, editors, *Progress in Cryptology - INDOCRYPT 2002*, volume 2551 of *Lecture Notes in Computer Science*, pages 22–33. Springer, 2002.

[ABN04] François Arnault, Thierry P. Berger, and Abdelkader Necer. Feedback with carry shift registers synthesis with the euclidean algorithm. *IEEE Transactions on Information Theory*, 50(5):910–917, 2004.

[Bab95] Steve H. Babbage. Improved "exhaustive search" attacks on stream ciphers. In *European Convention on Security and Detection*, volume 408, pages 161–166. IEEE Conference Publication, 1995.

[BAL05] Thierry Berger, François Arnault, and Cédric Lauradoux. F-FCSR. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/008, 2005. `http://www.ecrypt.eu.org/stream`.

[BBC+08] Come Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cedric Lauradoux, Marine Minier, Thomas Pornin, and Herve Sibert. SOSEMANUK, a fast software-oriented stream cipher. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 98–118. Springer, 2008.

[BCPP09] Jean Bourgain, Todd Cochrane, Jennifer Paulhus, and Christopher Pinner. Decimations of ℓ-sequences and permutations of even residues $\mod p$. *SIAM Journal on Discrete Mathematics*, 23(2):842–857, 2009.

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

[BD05] Steve Babbage and Matthew Dodd. The stream cipher MICKEY (version 1). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/015, 2005. `http://www.ecrypt.eu.org/stream`.

[BD08] Steve Babbage and Matthew Dodd. The MICKEY Stream Ciphers. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2008.

[Ber68a] Elwyn R. Berlekamp. *Algebraic Coding Theory*, chapter 7 and 10. McGraw-Hill Inc., New York, NY, USA, 1968.

[Ber68b] Elwyn R. Berlekamp. Nonbinary BCH decoding. *IEEE Trans. Inform. Theory*, 14(2):242, 1968.

[Ber08] Daniel J. Bernstein. The Salsa20 Family of Stream Ciphers. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2008.

[BFSS01] Cyril Banderier, Philippe Flajolet, Gilles Schaeffer, and Michèle Soria. Random maps, coalescing saddles, singularity analysis, and airy phenomena. *Random Struct. Algorithms*, 19(3-4):194–246, 2001.

[BG84] Manuel Blum and Shafi Goldwasser. An efficient probabilistic public-key encryption scheme which hides all partial information. In *Advances in Cryptology - CRYPTO'84*, pages 289–302, 1984.

[BGL+03] Marco Bucci, Lucia Germani, Raimondo Luzzi, Alessandro Trifiletti, and Mario Varanonuovo. A high-speed oscillator-based truly random number source for cryptographic applications on a smart card ic. *IEEE Trans. Computers*, 52(4):403–409, 2003.

[BM05] Thierry P. Berger and Marine Minier. Two algebraic attacks against the F-FCSRs using the IV mode. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005*, volume 3797 of *Lecture Notes in Computer Science*, pages 143–154. Springer, 2005.

[BS00] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2000.

[BVZ08] Martin Boesgaard, Mette Vesterager, and Erik Zenner. The Rabbit Stream Cipher. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2008.

[CDR98] Thomas W. Cusick, Cunsheng Ding, and Ari Renvall. *Stream Ciphers and Number Theory.* North-Holland Mathematic Library. Elsevier, Amsterdam, The Netherlands, 1998.

[CHM+04] Kevin Chen, Matthew Henricksen, William Millan, Joanne Fuller, Leonie Ruth Simpson, Ed Dawson, Hoon Jae Lee, and Sang-Jae Moon. Dragon: A fast word based stream cipher. In Choonsik Park and Seongtaek Chee, editors, *ICISC 2004*, volume 3506 of *Lecture Notes in Computer Science*, pages 33–50. Springer, 2004.

[CJS00] Vladimor V. Chepyzhov, Thomas Johansson, and Ben Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In Bruce Schneier, editor,

Fast Software Encryption - FSE 2000, volume 1978 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2000.

[CL94] Raymond Couture and Pierre L'Ecuyer. On the lattice structure of certain linear congruential sequences related to AWC/SWB generators. *Mathematics of Computations*, 62(206):799–808, 1994.

[CL97] Raymond Couture and Pierre L'Ecuyer. Distribution properties of multiply-with-carry random number generators. *Mathematics of Computations*, 66(218):591–607, 1997.

[CM03] Nicolas Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2003.

[CP07] Joo Yeon Cho and Josef Pieprzyk. An improved distinguisher for dragon. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/002, 2007. `http://www.ecrypt.eu.org/stream`.

[CP08] Christophe De Cannière and Bart Preneel. TRIVIUM. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 244–266. Springer, 2008.

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

[Den82] Dorothy Elizabeth Robling Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[DHS08] Ed Dawson, Matt Hendricksen, and Leonie Simpson. The dragon stream cipher: Design, analysis, and implementation issues. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2008.

[Dic07] Markus Dichtl. Bad and good ways of post-processing biased physical random numbers. In Alex Biryukov, editor, *Fast Software Encryption - FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2007.

[DM89] Persi Diaconis and Frederick Mosteller. Methods for Studying Coincidences. *Journal of the American Statistical Association*, 84(408):853–861, 1989.

[DS97] Michael Drmota and Michèle Soria. Images and preimages in random mappings. *SIAM Journal on Discrete Mathematics*, 10(2):246–269, 1997.

[dW79] Benjamin M.M. de Weger. Approximation lattices of p-adic numbers. *J. Number Theory*, 24(1):70–88, 1979.

[EM05] Håkan Englund and Alexander Maximov. Attack the dragon. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005*, volume 3797 of *Lecture Notes in Computer Science*, pages 130–142. Springer, 2005.

[EÖ05] Salih Ergün and Serdar Özoguz. A truly random number generator based on a continuous-time chaotic oscillator for applications in cryptography. In Pinar Yolum, Tunga Güngör, Fikret S. Gürgen, and Can C. Özturan, editors, *ISCIS*, volume 3733 of *Lecture Notes in Computer Science*, pages 205–214. Springer, 2005.

[eST] eSTREAM, the ECRYPT stream cipher project. `http://www.ecrypt.eu.org/stream/`.

[FFG+06] Philippe Flajolet, Éric Fusy, Xavier Gourdon, Daniel Panario, and Nicolas Pouyanne. A hybrid of Darboux's method and singularity analysis in combinatorial asymptotics. *Electr. J. Comb.*, 13(1), 2006.

[FFK05] James A. Fill, Philippe Flajolet, and Nevin Kapur. Singularity analysis, Hadamard products, and tree recurrences. *J. Comput. Appl. Math.*, 174(2):271–313, 2005.

[Fil00] Eric Filiol. Decimation attack of stream ciphers. In Bimal K. Roy and Eiji Okamoto, editors, *Progress in Cryptology - INDOCRYPT 2000*, volume 1977 of *Lecture Notes in Computer Science*, pages 31–42. Springer, 2000.

[Fla99] Philippe Flajolet. Singularity analysis and asymptotics of Bernoulli sums. *Theor. Comput. Sci.*, 215(1-2):371–381, 1999.

[FMS08] Simon Fischer, Willi Meier, and Dirk Stegemann. Equivalent representations of the F-FCSR keystream generator. In *SASC 2008, Workshop Records*, pages 87–96, 2008.

[FO89] Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. In Alfredo DeSantis, editor, *Advances in Cryptology - EUROCRYPT'89*, volume 434 of *Lecture Notes in Computer Science*, pages 329–354. Springer, 1989.

[FO90] Philippe Flajolet and Andrew Odlyzko. Singularity analysis of generating functions. *SIAM Journal on Discrete Mathematics*, 3(2):216–240, 1990.

[FS08] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics.* preprint, published January 2009 by Cambridge University Press, 31 August 2008. `http://algo.inria.fr/flajolet/Publications/book.pdf`.

[FT91] Gui Liang Feng and Kenneth K. Tzeng. A Generalization of the Berlekamp-Massey Algorithm for Multisequence Shift-Register Synthesis with Applications to Decoding Cyclic Codes. *IEEE Transactions on Information Theory*, 37(5):1274–1287, 1991.

[Gau01] Carl Friedrich Gauss. *Disquisitiones Arithmeticae.* reprinted in 1965 by Yale University Press, New Haven, 1801. English translation by Arthur A. Clarke.

[GB07] Tim Good and Mohammed Benaissa. Hardware results for selected stream cipher candidates. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/023, 2007. `http://www.ecrypt.eu.org/stream`.

[GK94] Mark Goresky and Andrew Klapper. Feedback registers based on ramified extensions of the 2-adic numbers (extended abstract). In Alfredo DeSantis, editor, *Advances in Cryptology - EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 215–222. Springer, 1994.

[GK97] Mark Goresky and Andrew Klapper. Arithmetic crosscorrelations of feedback with carry shift register sequences. *IEEE Transactions on Information Theory*, 43(4):1342–1345, 1997.

[GK02] Mark Goresky and Andrew Klapper. Fibonacci and Galois representations of feedback-with-carry shift registers. *IEEE Transactions on Information Theory*, 48(11):2826–2836, 2002.

[GK03] Mark Goresky and Andrew Klapper. Efficient multiply-with-carry random number generators with maximal period. *ACM Trans. Model. Comput. Simul.*, 13(4):310–321, 2003.

[GK04] Mark Goresky and Andrew Klapper. Periodicity and correlation properties of d-FCSR sequences. *Des. Codes Cryptography*, 33(2):123–148, 2004.

[GK06] Mark Goresky and Andrew Klapper. Periodicity and distribution properties of combined FCSR sequences. In Guang Gong, Tor Helleseth, Hong-Yeop Song, and Kyeongcheol Yang, editors, *Sequences and Their Applications - SETA 2006*, volume 4086 of *Lecture Notes in Computer Science*, pages 334–341. Springer, 2006.

[GK08] Mark Goresky and Andrew Klapper. *Algebraic Shift Register Sequences.* preprint, 10 Apr. 2008. `http://www.cs.uky.edu/~klapper/algebraic.html`.

[GKMS04] Mark Goresky, Andrew Klapper, Ram Murty, and Igor Shparlinski. On Decimations of ℓ-Sequences. *SIAM Journal of Discrete Mathematics*, 18(1):130–140, 2004.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984. Previous version in STOC 1982.

[Gol81] Solomon W. Golomb. *Shift Register Sequences.* Aegean Park Press, Laguna Hills, CA, USA, 1981.

[Gol90] Oded Goldreich. A note on computational indistinguishability. *Information Processing Letters*, 34:277–281, 1990.

[Gol97] Jovan Dj. Golic. Cryptanalysis of alleged A5 stream cipher. In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 1997.

[GW96] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr. Dobbs Journal*, Jannuary 1996. `http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html`.

[Har28] Ralph V. L. Hartley. Transmission of information. *Bell System Technical Journal*, 7(4):535–563, 1928.

[Har60] Bernard Harris. Probability distributions related to random mappings. *The Annals of Mathematical Statistics*, 31(4):1045–1062, 1960.

[Hel80] Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[HJ08] Martin Hell and Thomas Johansson. Breaking the F-FCSR-H stream cipher in real time. In J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, volume 5350 of *to appear in Lecture Notes in Computer Science*, pages 557–569. Springer, 2008.

[HJMM08] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain Family of Stream Ciphers. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 179–190. Springer, 2008.

[HK05] Jin Hong and Woo-Hwan Kim. TMD-tradeoff and state entropy loss considerations of streamcipher MICKEY. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005*, volume 3797 of *Lecture Notes in Computer Science*, pages 169–182. Springer, 2005.

[HP02] John. L Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, third edition, 2002.

[HS05] Jin Hong and Palash Sarkar. New applications of time memory data tradeoffs. In Bimal K. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 353–372. Springer, 2005.

[JDP05] Joseph Lano Joan Daemen and Bart Preneel. Chosen Ciphertext Attack on SSS. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/044, 2005. http://www.ecrypt.eu.org/stream.

[JF99] A. J. Johansson and H. Floberg. Random number generation by chaotic double scroll oscillator on chip. In *ISCAS (5)*, pages 407–409. IEEE, 1999.

[JM05a] Éliane Jaulmes and Frédéric Muller. Cryptanalysis of ECRYPT candidates F-FCSR-8 and F-FCSR-H. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/046, 2005. http://www.ecrypt.eu.org/stream.

[JM05b] Éliane Jaulmes and Frédéric Muller. Cryptanalysis of the F-FCSR stream cipher family. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2005.

[JM06] Antoine Joux and Frédéric Muller. Chosen-ciphertext attacks against MOSQUITO. In Matthew J. B. Robshaw, editor, *Fast Software Encryption - FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 390–404. Springer, 2006.

[KG93] Andrew Klapper and Mark Goresky. 2-adic shift registers. In Ross J. Anderson, editor, *Fast Software Encryption - FSE'93*, volume 809 of *Lecture Notes in Computer Science*, pages 174–178. Springer, 1993.

[KG95a] Andrew Klapper and Mark Goresky. Cryptanalysis based on 2-adic rational approximation. In Don Coppersmith, editor, *Advances in Cryptology - CRYPTO'95*, volume 963 of *Lecture Notes in Computer Science*, pages 262–273. Springer, 1995.

[KG95b] Andrew Klapper and Mark Goresky. Large periods nearly de Bruijn FCSR sequences. In Louis C. Guillou and Jean-Jacques Quisquater, editors, *Advances in Cryptology - EUROCRYPT'95*, volume 921 of *Lecture Notes in Computer Science*, pages 263–273. Springer, 1995.

[KG97] Andrew Klapper and Mark Goresky. Feedback shift registers, 2-adic span, and combiners with memory. *Journal of Cryptology*, 10(2):111–147, 1997.

[Kla94] Andrew Klapper. Feedback with carry shift registers over finite fields (extended abstract). In Bart Preneel, editor, *Fast Software Encryption - FSE'94*, volume 1008 of *Lecture Notes in Computer Science*, pages 170–178. Springer, 1994.

[Kla04] Andrew Klapper. Distributional properties of d-FCSR sequences. *J. Complex.*, 20(2-3):305–317, 2004.

[Kla08] Andrew Klapper. Expected π-adic security measures of sequences. In Solomon W. Golomb, Matthew G. Parker, Alexander Pott, and Arne Winterhof, editors, *Sequences and Their Applications - SETA*, volume 5203 of *Lecture Notes in Computer Science*, pages 219–229. Springer, 2008.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley, 1981.

[Kol86] Valentin F. Kolchin. *Random Mappings.* Optimization Software, Inc, New York, NY, USA, 1986.

[KSF99] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In Howard M. Heys and Carlisle M. Adams, editors, *Selected Areas in Cryptography*, volume 1758 of *Lecture Notes in Computer Science*, pages 13–33. Springer, 1999.

[KX89] Andrew Klapper and Jinzhong Xu. Feedback with carry shift registers over $\mathbb{Z}/(n)$. In Cunsheng Ding, Tor Helleseth, and Harald Niederreiter, editors, *Sequences and Their Applications - SETA'89*, Discrete Mathematics and Theoretical Computer Science, pages 379–392. Springer, 1989.

[KX99] Andrew Klapper and Jinzhong Xu. Algebraic feedback shift registers. *Theor. Comput. Sci.*, 226(1-2):61–92, 1999.

[KX04] Andrew Klapper and Jinzhong Xu. Register synthesis for algebraic feedback shift registers based on non-primes. *Des. Codes Cryptography*, 31(3):227–250, 2004.

[Lac08] Patrick Lacharme. Post-processing functions for a biased physical random number generator. In Kaisa Nyberg, editor, *Fast Software Encryption - FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2008.

[LE71] Abraham Lempel and Willard L. Eastman. High speed generation of maximal length sequences. *IEEE Transactions on Computers*, 2:227–229, 1971.

[L'E98] Pierre L'Ecuyer. Uniform random number generators. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 97–104, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[LR08] Cédric Lauradoux and Andrea Röck. Parallel generation of ℓ-sequences. In Solomon W. Golomb, Matthew G. Parker, Alexander Pott, and Arne Winterhof, editors, *Sequences and Their Applications - SETA 2008*, volume 5203 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2008.

[Mar94] George Marsaglia. Yet another RNG. Posted to electronic bulletin board *sci.stat.math.*, Aug 1 1994.

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

[Mas69] James L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. Inform. Theory*, 15:122–127, 1969.

[Mat93] Mitsuru Matsui. Linear cryptoanalysis method for des cipher. In *EUROCRYPT*, pages 386–397, 1993.

[Mau92] Ueli M. Maurer. A universal statistical test for random bit generators. *J. Cryptology*, 5(2):89–105, 1992.

[McE87] Robert J. McEliece. *Finite field for scientists and engineers*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

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

[MP75] David E. Muller and Franco P. Preparata. Bounds to complexities of networks for sorting and switching. *JACM*, 22:1531–1540, 1975.

[MR84] James L. Massey and Rainer A. Rueppel. Linear ciphers and random sequence generators with multiple clocks. In Thomas Beth, Norbert Cot, and Ingemar Ingemarsson, editors, *Advances in Cryptology - EUROCRYPT'84*, volume 209 of *Lecture Notes in Computer Science*, pages 74–87. Springer, 1984.

[MR89] James L. Massey and Rainer A. Rueppel. *Method of, and apparatus for, transforming a digital data sequence into an encoded form.* U. S. Patent Office, January 1989. US Patent 4,797,922.

[MS89] Willi Meier and Othmar Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1(3):159–176, 1989.

[Mut88] Ljuben R. Mutafchiev. Limit theorem concerning random mapping patterns. *Combinatorica*, 8(4):345–356, 1988.

[Mut89] Ljuben R. Mutafchiev. The limit distribution of the number of nodes in low strata of a random mapping. *Statist. Prabab. Lett.*, 7(3):247–251, 1989.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 0-8493-8523-7, 1996.

[MZ91] George Marsaglia and Arif Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, 1991.

[Nat91] National Institute of Standards and Technology (NIST). *Security Requirements for Cryptographic Modules, FIPS-PUB 140-1*, 1991. Available from `http://csrc.nist.gov/publications`.

[Nat01] National Institute of Standards and Technology (NIST). *Security Requirements for Cryptographic Modules, FIPS-PUB 140-2*, 2001. Available from `http://csrc.nist.gov/publications`.

[NES] NESSIE, New European Schemes for Signatures, Integrity, and Encryption. `https://www.cosic.esat.kuleuven.be/nessie/`.

[Nie92] Harald Niederreiter. *Random number generation and quasi-Monte Carlo methods.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[NW06] Kaisa Nyberg and Johan Wallén. Improved linear distinguishers for SNOW 2.0. In Matthew J. B. Robshaw, editor, *Fast Software Encryption - FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2006.

[pLa] plab: Random number generators. `http://random.mat.sbg.ac.at/`.

[Pre05] Bart Preneel. Nessie project. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security.* Springer, 2005.

[PW68] Paul W. Purdom and John H. Williams. Cycle length in a random function. *Transactions of the American Mathematical Society*, 133:547–551, 1968.

[QD88] Jean-Jacques Quisquater and Jean-Paul Delescaille. Other cycling tests for DES (abstract). In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 255–256. Springer, 1988.

[QD89] Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search. new results and applications to des. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 408–413. Springer, 1989.

[RB08] Mathew Robshaw and Olivier Billet, editors. *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science.* Springer, 2008.

[Röc07a] Andrea Röck. Attaques par collision basées sur la perte d'entropie causée par des fonctions aléatoires. In *MajecSTIC 2007*, pages 115–124, October 2007.

[Röc07b] Andrea Röck. The impact of entropy loss caused by random functions. In *WEWoRC 2007, Western European Workshop on Research in Cryptology*, pages 130–135, Bochum, Germany, July 2007.

[Röc08a] Andrea Röck. Entropy of the internal state of an FCSR in Galois representation. In Kaisa Nyberg, editor, *Fast Software Encryption - FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 343–362. Springer, 2008.

[Röc08b] Andrea Röck. Stream ciphers using a random update function: Study of the entropy of the inner state. In Serge Vaudenay, editor, *AFRICACRYPT*, volume 5023 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2008.

[RSN+01] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800-22, May 2001.

[Rue86] Rainer A. Rueppel. *Analysis and design of stream ciphers*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[SC88] Ben J. M. Smeets and William G. Chambers. Windmill generators: A generalization and an observation of how many there are. In Christoph G. Günther, editor, *Advances in Cryptology - EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, pages 325–330. Springer, 1988.

[Sch95] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Sch03] B. Schneier. Yarrow, a secure pseudorandom number generator. `http://www.schneier.com/yarrow.html`, November 2003.

[SHA] Cryptographic hash algorithm competition (sha-3). `http://csrc.nist.gov/groups/ST/hash/sha-3/`. National Institute of Standards and Technology (NIST).

[Sha49] Claude E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[Sha04] Adi Shamir. Stream ciphers: Dead or alive? In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, page 78. Springer, 2004.

[Sie84] Thomas Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, 30(5):776–780, 1984.

[SS02] Nicolas Sendrier and André Seznec. HArdware Volatile Entropy Gathering and Expansion: generating unpredictable random numbers at user level. Technical report, INRIA, 2002.

[SS03] André Seznec and Nicolas Sendrier. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Trans. Model. Comput. Simul.*, 13(4):334–346, 2003.

[SS06] Georg Schmidt and Vladimir Sidorenko. Linear shift-register synthesis for multiple sequences of varying length. In *IEEE International Symposium on Information Theory - ISIT 2006*, pages 1738–1742. IEEE, 2006.

[Ste69] V. E. Stepanov. Limit distributions of certain characteristics of random mappings. *Theory of Probability and its Applications*, XIV(4), 1969.

[STKT06] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday Paradox for Multi-collisions. In Min Surp Rhee and Byoungcheon Lee, editors, *Information Security and Cryptology – ICISC 2006, Proceedings*, volume 4296 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 2006.

[SW49] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, Urbana, IL, USA, 1949. Republished in paperback 1963.

[TccSD08] Meltem Sönmez Turan, Çagdas Çalik, Nurdan Buz Saran, and Ali Doganaksoy. New distinguishers based on random mappings against stream ciphers. In Solomon W. Golomb, Matthew G. Parker, Alexander Pott, and Arne Winterhof, editors, *Sequences and Their Applications - SETA 2008*, volume 5203 of *Lecture Notes in Computer Science*, pages 30–41. Springer, 2008.

[TM71] Myron Tribus and Edward C. McIrvine. Energy and information (thermodynamics and information theory). *Scientific American*, 225(3):179–188, September 1971.

[TQ09] Tian Tian and Wen-Feng Qi. Autocorrelation and distinctness of decimations of l-sequences. *SIAM Journal on Discrete Mathematics*, 23(2):805–821, 2009.

[Ts'99] Theodore Ts'o. random.c – A strong random number generator, September 1999. /driver/char/random.c in Linux Kernel 2.4.20, `http://www.kernel.org/`.

[vT05] Henk C. A. van Tilborg, editor. *Encyclopedia of Cryptography and Security.* Springer, 2005.

[Wal98] John Walker. ENT a pseudorandom number sequence test program. Available from `http://www.fourmilab.ch/random/`, October 1998.

[Wal06] John Walker. HotBits: Genuine random numbers, generated by radioactive decay, september 2006. `http://www.fourmilab.ch/hotbits/`.

[Weg98] Stefan Wegenkittl. *Generalized ϕ-Divergence and Frequency Analysis in Markov Chains*. PhD thesis, University of Salzburg, Austria, 1998.

[Weg01] Stefan Wegenkittl. Entropy estimators and serial tests for ergodic chains. *IEEE Transactions on Information Theory*, 47(6):2480–2489, 2001.

[Wu08] Hongjun Wu. The Stream Cipher HC-128. In Mathew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 39–47. Springer, 2008.

[XQ06] Hong Xu and Wenfeng Qi. Further Results on the Distinctness of Decimations of ℓ-Sequences. *IEEE Transactions on Information Theory*, 52(8):3831–3836, 2006.

[Zie59] Neal Zierler. Linear recurring sequences. *Journal of the Society for Industrial and Applied Mathematics*, 7(1):31–48, 1959.

[ZWFB04] Bin Zhang, Hongjun Wu, Dengguo Feng, and Feng Bao. Chosen ciphertext attack on a new class of self-synchronizing stream ciphers. In Anne Canteaut and Kapalee Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004*, volume 3348 of *Lecture Notes in Computer Science*, pages 73–83. Springer, 2004.

List of Figures

1.1 Statue of Claude E. Shannon, MIT, Cambridge, USA. 3

2.1 Schematic diagram of a general communication system [SW49]. 8
2.2 Decomposition of a choice from three possibilities [SW49]. 9

3.1 Model of a synchronous stream cipher. 15
3.2 Model of a additive stream cipher. 15
3.3 Model of a self-synchronizing stream cipher. 16

5.1 Memory devices. 24
5.2 Branch predictor. 26
5.3 Scheduling. 27

6.1 Detail of histogram in normal/idle setting. 37
6.2 Histogram of the normal version. 38
6.3 Histogram of version: No Branch Predictor. 38
6.4 Histogram of version: One Loop. 39
6.5 Histogram of version: More Loops. 39
6.6 Autocorrelation in the normal/idle setting with a shift of up to 200. 40
6.7 Autocorrelation in setting: Idle. 41
6.8 Autocorrelation in setting: File Copy (Random). 41
6.9 Autocorrelation in setting: File Copy (Ordered). 42
6.10 Influence of data size for Markov chain entropy estimator. 47
6.11 Entropy estimation for Markov chains of order $\kappa < 4$. 48

7.1 Key and IV setup in 128-bit word representation. 54
7.2 Key and IV setup in 32-bit word representation. 55
7.3 Update function. 55

8.1 Example of a functional graph for $\varphi : x \mapsto ((x-1)^2+2) \pmod{20}+1 \in \mathcal{F}_{20}$. 72

9.1 Model of a simple stream cipher. 82
9.2 Upper bound and empirical average of the entropy for $n = 2^{16}$. 84
9.3 Example of a functional graph to illustrate $\mathrm{rn}_k^{\varphi}(r)$. 85
9.4 Example of the structure explained in 1.-3. for the node A. 89

9.5 The course of $c_k(r)\ r\ \log_2(r)$ for different values of k and r. 93
9.6 Collision search in the last column. 95
9.7 Collision search in the second half of the iterations. 96
9.8 The attack using distinguished points. 98
9.9 Empirical results for an attack with DP and $n = 2^{20}$. 98

10.1 Fibonacci LFSR. 105
10.2 Galois LFSR. 105
10.3 Fibonacci FCSR. 107
10.4 Galois FCSR. 108
10.5 Functional graph of a Galois FCSR with $n = 3$ and $q = 13$. 111
10.6 F-FCSR. 117
10.7 X-FCSR-256. 121
10.8 Fibonacci d-FCSR. 123
10.9 The $\mathbb{Z}[\pi]$ adder in detail for $d = 2$ and $\sum_{i=1}^{n} q_i a_{n-i} \leq 7$. 123
10.10 Galois d-FCSR for $d = 2$. 124
10.11 AFSR. 124

11.1 Evaluation of $h = a + 2c$ bit per bit. 131
11.2 Example for $a = 13$ and $c = 2$. 131
11.3 Reduction of $h, a, 2c$ to h', a', c'. 133
11.4 FCSR with $q = 2^n + 2^{\ell+1} - 3$. 138
11.5 Moving the feedback position. 139

12.1 Model of a sub-sequences generator. 146
12.2 Sub-sequences generator using LFSR synthesis. 148
12.3 Multiple steps generator for a Fibonacci LFSR. 150
12.4 Multiple steps generator for a Galois LFSR. 151
12.5 Multiple steps generator for a Fibonacci FCSR. 154
12.6 Example for a Galois FCSR with $q = 19$. 155
12.7 Multiple steps generator for a Galois FCSR. 156
12.8 FCSR synthesis. 157

List of Algorithms

1 Scheme of HAVEGE. 29
2 Final entropy. 136

List of Tables

3.1 Final eSTREAM portfolio. 17

5.1 Without instruction pipeline. 25
5.2 Using the instruction pipeline. 26
5.3 Summary of HAVEGE. 33

6.1 Mean (PC 1). 37
6.2 Sample entropy $H(\hat{\mathbf{P}})$ on PC 1. 43
6.3 Sample entropy $H(\hat{\mathbf{P}})$ on PC 2. 43
6.4 Number of possible tuples compared with sample size. 47

7.1 Relation among words. 62
7.2 Bias of equations. 63
7.3 Possible combinations. 64
7.4 Biases for $\Gamma = 0x600018d$. 65

8.1 Some examples of generating functions and their coefficients. 75

9.1 Comparison of different methods to estimate the entropy loss. 94
9.2 Complexities of attacks. 99

11.1 Comparison of the exact computation of the final state entropy, with upper and lower bounds. 137

12.1 Comparison of the two methods for the synthesis of a sub-sequences generator. 149