On the Design and Analysis of Stream Ciphers

Hell, Martin

2007

On the Design and Analysis of Stream Ciphers

Martin Hell

LUND UNIVERSITY

Ph.D. Thesis

September 13, 2007

Martin Hell
Department of Electrical and Information Technology
Lund University
Box 118
S-221 00 Lund, Sweden
e-mail: martin@eit.lth.se
http://www.eit.lth.se/

ISBN: 91-7167-043-2
ISRN: LUTEDX/TEIT-07/1039-SE

Abstract

This thesis presents new cryptanalysis results for several different stream cipher constructions. In addition, it also presents two new stream ciphers, both based on the same design principle.

The first attack is a general attack targeting a nonlinear combiner. A new class of weak feedback polynomials for linear feedback shift registers is identified. By taking samples corresponding to the linear recurrence relation, it is shown that if the feedback polynomial has taps close together an adversary to take advantage of this by considering the samples in a vector form.

Next, the self-shrinking generator and the bit-search generator are analyzed. Both designs are based on irregular decimation. For the self-shrinking generator, it is shown how to recover the internal state knowing only a few keystream bits. The complexity of the attack is similar to the previously best known but uses a negligible amount of memory. An attack requiring a large keystream segment is also presented. It is shown to be asymptotically better than all previously known attacks. For the bit-search generator, an algorithm that recovers the internal state is given as well as a distinguishing attack that can be very efficient if the feedback polynomial is not carefully chosen.

Following this, two recently proposed stream cipher designs, Pomaranch and Achterbahn, are analyzed. Both stream ciphers are designed with small hardware complexity in mind. For Pomaranch Version 2, based on an improvement of previous analysis of the design idea, a key recovery attack is given. Also, for all three versions of Pomaranch, a distinguishing attack is given. For Achterbahn, it is shown how to recover the key of the latest version, known as Achterbahn-128/80.

The last part of the thesis introduces two new stream cipher designs, namely Grain and Grain-128. The ciphers are designed to be very small in hardware. They also have the distinguishing feature of allowing users to increase the speed of the ciphers by adding extra hardware.

Contents

Preface

This thesis presents the results from my time as a Ph.D. student at the Department of Electrical and Information Technology at Lund University in Sweden. Parts of the material have been presented at international conferences and are based on results from the following papers:

- √ H. ENGLUND, M. HELL AND T. JOHANSSON. Correlation Attacks Using a New Class of Weak Feedback Polynomials. In B. Roy and W. Meier, editors, *Fast Software Encryption – 2004, Delhi, India*, volume 3017 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 2004.

- √ M. HELL AND T. JOHANSSON. Some Attacks on the Bit-Search Generator. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption – 2005, Paris, France*, volume 3557 of *Lecture Notes in Computer Science*, pages 215–227. Springer-Verlag, 2005.

- √ M. HELL AND T. JOHANSSON. On the Problem of Finding Linear Approximations and Cryptanalysis of Pomaranch Version 2. In E. Biham and A. M. Youssef, editors, *Selected Areas in Cryptography – 2006, Montréal, Canada*, volume 4356 of *Lecture Notes in Computer Science*, pages 220–234. Springer-Verlag, 2007.

- √ M. HELL AND T. JOHANSSON. Cryptanalysis of Achterbahn-Version 2. In E. Biham and A. M. Youssef, editors, *Selected Areas in Cryptography – 2006, Montréal, Canada*, volume 4356 of *Lecture Notes in Computer Science*, pages 45–55. Springer-Verlag, 2007.

- √ M. HELL, T. JOHANSSON, A. MAXIMOV AND W. MEIER. A Stream Cipher Proposal: Grain-128. In A. Barg and R. W. Yeung, editors, *IEEE International Symposium on Information Theory – 2006, Seattle, USA*, CD-ROM, 2006.

- √ H. ENGLUND, M. HELL AND T. JOHANSSON. Two General Attacks on Pomaranch-like Keystream Generators. *Fast Software Encryption – 2007, Luxembourg*. Preproceedings, 2007.

Parts of the results are based on the following journal papers:

- √ M. HELL AND T. JOHANSSON. Two New Attacks on the Self-Shrinking Generator. *IEEE Transactions on Information Theory*, 52(8):3837–3843, 2006.

- √ M. HELL, T. JOHANSSON, AND W. MEIER. Grain - a Stream Cipher for Constrained Environments. *International Journal of Wireless and Mobile Computing, Special Issue on Security of Computer Network and Mobile Systems*, 2006.

- √ M. HELL AND T. JOHANSSON. Cryptanalysis of Achterbahn-128/80. *IET Information Security*, 1(2):47–52, 2007.

During my time as a Ph.D. student, I have also co-authored the following papers which are not included in this thesis:

- √ M. HELL, A. MAXIMOV, AND S. MAITRA. On Efficient Implementation of a Search Strategy for RSBFs. In *Proceedings of the International Workshop on Algebraic and Combinatorial Coding Theory, Kranevo, Bulgaria*, pages 214–222, 2004.

- √ A. MAXIMOV, M. HELL, AND S. MAITRA. Plateaued Rotation Symmetric Boolean Functions on Odd Number of Variables. In *Proceedings of the Workshop on Boolean Functions: Cryptography and Applications – 2005, Rouen, France*.

- √ G. GONG, K. C. GUPTA, M. HELL, AND Y. NAWAZ. Towards a General RC4-like Keystream Generator. In D. Feng, D. Lin, and M. Yung, editors, *Conference on Information Security and Cryptology – CISC 2005, Beijing, China*, volume 3822 of *Lecture Notes in Computer Science*, pages 162–174. Springer-Verlag, 2005.

- √ H. ENGLUND, M. HELL, AND T. JOHANSSON. A Note on Distinguishing Attacks. In *Proceedings of the 2007 IEEE Information Theory Workshop on Information Theory for Wireless Networks, Solstrand, Norway.*, pages 87–90, 2007.

Acknowledgements

Many people have contributed in various ways to this thesis. I would like to take this opportunity to express my gratitude to all of them. In particular I would like to mention some of them.

First of all, I want to thank my supervisor Thomas Johansson. His patience, knowledge and support have been invaluable to me. His brilliant research ideas have always kept me busy. However, thanks to his quick rejections of my own, sometimes not so brilliant, ideas I never felt too busy.

My friends and colleagues at the Department of Electrical and Information Technology have created an inspiring atmosphere to work in. Especially my friends in the crypto group, Håkan and Sasha. Also, and in no order of importance, I would like to mention Marcus, Kora, Fredrik, Maja, Thomas and Suleyman. I would also like to express thanks and appreciation to the technical and administrative staff at the department.

I attribute the success of reaching this point in my life to the invaluable support and encouragement from my parents. Finally, behind every successful man there is a very surprised woman. My deepest thanks go to Anna for all her love and understanding.

Martin Hell

1

Introduction

The amount of data being exchanged and stored electronically is rapidly increasing. A significant amount of this data needs to be protected and the protection is in many cases of great importance. The need to protect messages or information is not new but has been present for a long time. A popular and very well-known example is the secret writing used by the Roman military and political leader Julius Caesar in the first century BC. In the communication with his generals each letter in the alphabet was replaced by the letter 3 positions later, known as the Caesar cipher. Another ancient cipher is the scytale, used by the ancient Greeks and first mentioned in the 7th century BC. It consists of a cylinder and a strip of leather. When the strip of leather is wrapped around the cylinder, the secret message is written on it. Once unwrapped, the message appeared to be random letters. Only a recipient with a cylinder of the same diameter can then read the message. The first documented use of secret writing dates back to around 1900 BC from when non-standard hieroglyphs have been found in inscriptions. However, it seems reasonable that the need for secret writing developed shortly after writing was invented.

1.1 Cryptology

The science referred to as cryptology is divided into two parts, *cryptography* and *cryptanalysis*. Historically, cryptography is the science of secret writing, though that definition requires a slight update as the range of modern cryptographic services incorporates additional features. Nowadays it can be referred to the science of designing any algorithm intended to offer a security goal. Cryptanalysis, on the other hand, is the science of breaking the same algorithms.

The security goals in modern cryptography can be divided into four categories

(*i*) *Authentication.* The process of verifying the identity of the sender/user. A computer login authenticates a user by requesting a password. The user proves his identity by showing that he knows a secret. In a *challenge-response scheme* the verifier sends a challenge, e.g., a random number A, to the prover. The prover calculates a new number $B = f(A, K)$ where K is some shared secret, and then return B to the verifier. Since the verifier knows K, he can also find B and if the returned number is correct, the prover has proved his identity. In a zero-knowledge proof, the goal is to allow the prover to prove that he knows a secret by not revealing the secret to the verifier. Authentication is closely related to *authorization*. Authorizing a user means to verify that an authenticated user has access to information. Thus, authentication must be performed before authorization.

(*ii*) *Confidentiality.* Ensuring that only the intended recipient (an authorized user) is able to read the message. This is achieved by encrypting the data using a *cipher*. The Caesar cipher and the scytale mentioned before are examples of classical ciphers.

(*iii*) *Integrity.* Assuring the receiver of a message that it has not been altered. Data sent on a computer network, passing through several hosts, can be maliciously altered on one host before sent to the next. Ensuring message integrity can be done using a *Message Authentication Code* (MAC), which computes a key dependent checksum of the message.

(*iv*) *Non-repudiation*. The goal here is to prove that the sender really sent the data. As an example, after signing a contract, the signer should not be able to deny that he signed it. Non-repudiation can be provided using digital signatures.

1.2 Cryptographic Primitives

A cryptographic primitive is an algorithm that attempts to realize one or several of the security goals given in Section 1.1. The primitives can be divided into three categories, namely *unkeyed primitives*, *secret key primitives* and *public key primitives*.

A hash function is an example of an unkeyed primitive. It takes as input a string of arbitrary length and outputs a string (hash value or message digest) of fixed length, typically 128, 160, 256 or 512 bits. A hash function should fulfill several requirements

- *Ease of Computation.* Given a message it should be easy to find its hash value.
- *Preimage Resistance.* It should be hard to find a message with a given hash value.
- *Second Preimage Resistance.* Given one message it should be hard to find another message with the same hash value.
- *Collision Resistance.* It should be hard to find two messages with the same hash value.

Some well-known examples of hash functions are SHA-1, MD4 and MD5. Recent cryptanalysis of these algorithms has lately increased the research focus on hash functions. The American National Institute of Standards and Technology (NIST) has initiated a competition to find a new standard for hash functions and the result is scheduled for late 2011.

Secret key primitives, also known as symmetric primitives, is a class of primitives that uses the same key for both encryption and decryption. This requires the sender and receiver to negotiate a key before the communication starts. Since anyone that possesses the secret key can decrypt messages, it is vital for the security that this key is exchanged over a secure channel. A secret key primitive can be a cipher, such as a block cipher or stream cipher, which provides confidentiality. It can also be a MAC providing message authentication and integrity protection. Then we talk about signing and verifying instead of encryption and decryption. A MAC is usually constructed using a block cipher or by using a hash function together with a key.

Public key primitives, or asymmetric primitives, use different keys for encryption and decryption. These primitives have the advantage over secret key primitives that they do not require a secure channel in order to share the key. Instead they use one public and one private key. A user's public key, used for encryption, can e.g., be posted on a webpage while the private key, used for decryption, is kept secret and is only known to the

user. Public key cryptography was introduced in 1976 by Diffie and Hellman [DH76] and two years later Rivest, Shamir and Adleman presented the encryption scheme RSA based on the public key principle. Still today, RSA is probably the most well-known public key scheme used. Another type of public key primitive is the *digital signature*. A digital signature can be created by encrypting the hash value of the message with the private key. Then the public key can be used to verify the signature. As with a MAC, a digital signature provides message authentication and integrity protection. However, digital signatures can also provide non-repudiation since a signature can only be produced by someone with knowledge of the secret key. On the other hand, anyone can verify a signature since the public key is assumed known to all. In the case of MACs, anyone that can verify a MAC can also create a MAC since the same key is used for both signing and verifying. Public key cryptography has also spawned new problems. In particular, how can we be certain that a public key belongs to a certain person? The solution to this is the use of certificates, in which a trusted third party guarantees the connection between a user and a public key. In general, a public key encryption scheme is computationally much slower than symmetric encryption schemes. Thus, instead of encrypting a long message with e.g., RSA, it is common to use RSA to encrypt a symmetric key. The symmetric key is then used to encrypt or decrypt the message.

1.3 Block Ciphers and Stream Ciphers

Symmetric primitives used to provide confidentiality can be divided into block ciphers and stream ciphers. The message to be encrypted is called *plaintext* and the result of the encryption is called *ciphertext*. A stream cipher generates a keystream, which is used to encrypt the plaintext, usually bit by bit. However, many software optimized stream ciphers, especially modern ones, encrypts the plaintext word by word where a word consists of several bits. Stream ciphers is the topic of this thesis and Chapter 2 is devoted entirely to this class of primitives.

In a block cipher, the plaintext is divided into blocks of n bits, with n being typically 64, 128 or 256 bits. Each block is then transformed by an invertible key dependent function into a ciphertext block. Thus, each key defines a permutation on n-bit blocks. A block cipher can be used in one of several modes of operation. Let K be the key, E_K the encryption function, D_K the decryption function, $\mathbf{m} = (m_1, m_2, \ldots)$ the message and $\mathbf{c} = (c_1, c_2, \ldots)$ the ciphertext[1]. Further, let IV be a public initialization vector. We give five common modes of operation for block ciphers.

[1] As with most programming languages, the index sometimes starts with 0 instead of 1.

- *Electronic Code Book Mode (ECB Mode).* This is the most obvious mode in which all plaintext blocks are encrypted and decrypted independently. Thus we have $c_t = E_K(m_t)$ and $m_t = D_K(c_t)$. A problem with ECB Mode is that redundancy in the plaintext blocks will be preserved in the ciphertext blocks.

- *Cipher Block Chaining Mode (CBC Mode).* This mode chains the previous ciphertext with the current plaintext. Encryption and decryption is given by $c_t = E_K(m_t \oplus c_{t-1})$ and $m_t = D_K(c_t) \oplus c_{t-1}$ respectively, where $c_0 = IV$.

- *Output Feedback Mode (OFB mode).* This will turn the block cipher into a stream cipher. A keystream $\mathbf{z} = (z_1, z_2, \ldots)$ is generated as $z_t = E_K(z_{t-1})$ with $z_0 = IV$. The encryption and decryption is given as $c_t = m_t \oplus z_t$ and $m_t = c_t \oplus z_t$ respectively.

- *Cipher Feedback Mode (CFB Mode).* This mode of operation also behaves like a stream cipher. The keystream $\mathbf{z} = (z_1, z_2, \ldots)$ is generated as $z_t = E_K(c_{t-1})$ with $c_0 = IV$. Like OFB Mode, encryption and decryption is given as $c_t = m_t \oplus z_t$ and $m_t = c_t \oplus z_t$ respectively.

- *Counter Mode.* This is another mode of operation that turns a block cipher into a stream cipher. The keystream $\mathbf{z} = (z_1, z_2, \ldots)$ is generated as $z_t = E_K(IV \| ctr)$ where *ctr* is a counter which is incremented for each encryption.

A block cipher in CFB Mode defines a self-synchronizing stream cipher while OFB Mode and Counter Mode define a synchronous stream cipher, see Section 2.1. Several other modes of operation have been suggested e.g., modes that authenticates the data as well, but the ones mentioned above are the most well known.

The *Data Encryption Standard* (DES) is a well-known block cipher. It was selected as a standard in the United States in 1976 and has been extensively analyzed since then. DES remained a standard until 2002, when the *Advanced Encryption Standard* (AES) replaced DES as a standard for block ciphers. AES was the result of a public competition initiated by NIST in 1997. The most important reason to replace DES as a standard was the small key size of 56 bits supported by DES. Already in 1998 it was suggested to use triple-DES instead, which runs the DES algorithm 3 successive times using a key of 112 or 168 bits. Though this solution solved the problem of short keys, it should be seen as a temporary solution since the speed of the algorithm suffered from the tweak. The AES algorithm, named Rijndael, supports keys of 128, 192 or 256 bits.

1.4 Thesis Outline

This thesis, as its name suggests, is devoted to the area of stream ciphers. It will show new ideas in both stream cipher analysis and stream cipher design. The rest of the thesis is outlined as follows.

Chapter 2 introduces the reader to the area of stream ciphers. The motivation to study stream ciphers and a general framework is first given. Then we talk about different design blocks that are common in stream ciphers and also give a few examples of existing stream ciphers. The chapter continues by discussing some selected ways to attack stream ciphers and it is ended with some basic theory regarding hypothesis testing.

Chapter 3 will define a new class of feedback polynomials that should be avoided when designing stream ciphers based on linear feedback shift registers. We show that it is possible to find efficient attacks if the feedback polynomial is of this weak form. Also, if the feedback polynomial has a low degree multiple of the weak form, then the attack can also be efficient.

Chapter 4 will consider cryptanalysis of the self-shrinking generator, which is conceptually one of the simplest stream ciphers imaginable. We present two new ideas on how to recover the initial state of the shift register used in the construction. The two new ideas will be shown to offer important advantages over previously known attacks on the self-shrinking generator.

Chapter 5 will consider a construction called the bit-search generator. It is similar to the self-shrinking generator in that it is very simple and can be built using very few components. We give both an attack that recovers the initial state of the shift register and also a distinguishing attack, which distinguishes the output sequence from a truly random sequence.

Chapter 6 shows possible ways to cryptanalyze the Pomaranch family of stream ciphers. At the time of writing there has been three versions and in total five variants of the cipher presented. We show attacks for all versions and variants. We show how to recover the key in version 1 and version 2 and we additionally show distinguishing attacks for all variants of the cipher. The chapter is ended with an attack considering a non-standard attack scenario. We show that all Pomaranch ciphers are vulnerable in this scenario.

Chapter 7 is the last chapter on cryptanalysis. We show how to recover the key in this family of ciphers. Like Pomaranch, there are several versions of the Achterbahn ciphers, but the chapter will only give cryptanalysis results for the, at the time of writing, latest version. However, the chapter will give an overview of the many, sometimes confusing, steps in the development of all different versions and variants of Achterbahn.

In Chapter 8 we leave the destructive behaviour of cryptanalysis behind

and focus on designing stream ciphers. The stream cipher family Grain is introduced. This family of ciphers aims to provide a secure primitive offering confidentiality while at the same time being extremely small and easy to implement in hardware. The design is, to the best of our knowledge, the smallest known stream cipher considering the key size. Grain also has the distinguishing feature that it is easy to increase the speed of the cipher by adding some extra hardware. Thus, it provides a very flexible solution for confidentiality in hardware environments where area and/or power consumption must be kept small.

2

Stream Ciphers

In this chapter we give a more detailed introduction to stream ciphers. The keystream produced by a stream cipher should be as random looking as possible in order to make it more resistant to attacks. However, good randomness properties are not enough in a modern stream cipher. Since a block cipher can be used in a stream cipher mode of operation, a dedicated stream cipher must offer at least one advantage over a block cipher in e.g., OFB mode. Lately the research on stream ciphers has focused on ciphers that, compared to AES, offer one or both of the following.

- Faster software performance, may it be either 8-bit, 16-bit, 32-bit or 64-bit architectures.
- Smaller hardware implementation in terms of gates, area and/or power consumption.

This focus is based on a widespread belief that stream ciphers *can* provide better performance than block ciphers in terms of the above mentioned properties and this seems to be the main motivation to study stream ciphers. This widespread belief has also been reflected by the eSTREAM project [ECR], a European project with a goal to identify new promising stream ciphers. In 2005, many new designs proposals were submitted to the project and according to the time schedule, eSTREAM will finish in May 2008. There are two profiles in eSTREAM, namely a software and a hardware profile. All proposals in the software profile has to offer better performance than AES in software and the proposals in the hardware profile has to be smaller and more efficient than AES in hardware.

Synchronous stream ciphers also have the property that the keystream can be generated before plaintext is available. This can also be seen as an advantage over block ciphers.

It is easy to design a secure stream cipher. The difficult task is to make it secure and at the same time provide excellent software or hardware performance. Many different design blocks have been suggested and are also implemented in various stream ciphers. Some of these design blocks will be discussed in this chapter.

The outline of this chapter is as follows. In Section 2.1 we give a formal description of a stream cipher and show how stream ciphers can be classified. Section 2.2 will give an overview of some common design blocks and also present a few examples of stream cipher designs. Cryptanalysis of stream ciphers is discussed in Section 2.3 and some common approaches and methods are given. Hypothesis testing is an important tool in cryptanalysis and a short theoretical background is given in Section 2.4. The chapter is summarized in Section 2.5.

2.1 Classification of Stream Ciphers

In this section we discuss how stream ciphers can be classified. We give a formal framework for a stream cipher. In large, it is based on the framework given in [MvOV97] but it should be noted that initialization vectors are not mentioned in [MvOV97] but is included in our treatment.

Let $\mathcal{M}$ be the set of possible plaintext symbols, $\mathcal{C}$ be the set of possible ciphertext symbols, $\mathcal{K}$ be the set of possible keys and $\mathcal{IV}$ be the set of possible initialization vectors. The set of possible keystream symbols is denoted by $\mathcal{Z}$ and the tth ciphertext symbol $c_t \in \mathcal{C}$ is given by

$$c_t = h(z_t, m_t), \tag{2.1}$$

where $z_t \in \mathcal{Z}$ and $m_t \in \mathcal{M}$. In decryption we have $m_t = h^{-1}(z_t, c_t)$. Depending on the structure, a stream cipher belongs to one of two categories,

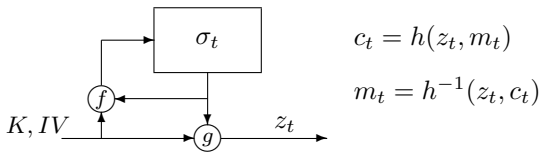

Figure 2.1: A model of a synchronous stream cipher.

synchronous or *self-synchronizing*.

Definition 2.1: In a *synchronous* stream cipher, the keystream is generated independently of the plaintext and the ciphertext.

Let $K \in \mathcal{K}$ and $IV \in \mathcal{IV}$. In a synchronous stream cipher, there is an internal state, denoted σ_t, which is updated as $\sigma_{t+1} = f(\sigma_t, K, IV)$. The initial state, σ_0, in the keystream generation phase is the result of an initialization phase, $\sigma_0 = \gamma(K, IV)$. The keystream is produced by $z_t = g(\sigma_t, K, IV)$. A model of a synchronous stream cipher is given in Fig. 2.1. Most proposed stream ciphers are *binary additive stream ciphers*. In that case, the plaintext, ciphertext and keystream are binary digits, $\mathcal{M} = \mathcal{C} = \mathcal{Z} = \mathbb{F}_2$, and the function $h(z_t, m_t)$ is the xor function, $c_t = m_t \oplus z_t$. The decryption is then given by $m_t = c_t \oplus z_t$

The initialization vector, IV, is not always included in the framework of stream ciphers, at least not in older works. Assume that we want to encrypt two messages, $\mathbf{m_1} = m_{1_1}, m_{1_2} \ldots$ and $\mathbf{m_2} = m_{2_1}, m_{2_2} \ldots$, using the same key K. Without a unique IV, the keystream $\mathbf{z} = z_1, z_2 \ldots$ generated will be the same for both messages. Assuming a binary additive stream cipher, the two ciphertexts, $\mathbf{c_1} = c_{1_1}, c_{1_2} \ldots$ and $\mathbf{c_2} = c_{2_1}, c_{2_2} \ldots$ will be given as

$$\begin{aligned} (c_{1_1}, c_{1_2}, c_{1_3} \ldots) &= (m_{1_1} \oplus z_1, m_{1_2} \oplus z_2, m_{1_3} \oplus z_3 \ldots) \\ (c_{2_1}, c_{2_2}, c_{2_3} \ldots) &= (m_{2_1} \oplus z_1, m_{2_2} \oplus z_2, m_{2_3} \oplus z_3 \ldots). \end{aligned} \tag{2.2}$$

Knowing c_{1_t} and c_{2_t} will give information about the pair (m_{1_t}, m_{2_t}) since there will then only be two possible values for this pair. Thus, the ciphertext is leaking information about the plaintext. Using an initialization vector, which can be public, together with the secret key will give a different keystream since the initialization process depends on both the key and the IV.

A self-synchronizing stream cipher can be defined as follows.

Definition 2.2: In a *self-synchronizing* stream cipher the keystream is generated as a function of the key and δ previous ciphertext bits.

The internal state consists of the previous δ keystream bits,

$$\sigma_t = (c_{t-\delta}, c_{t-\delta+1}, \ldots, c_{t-1}). \tag{2.3}$$

For the first δ symbols, the previous δ symbols do not exist and instead, these are defined by the initialization vector shared between the sender and the transmitter. Similar to synchronous stream ciphers, the keystream is produced by $z_t = g(\sigma_t, K)$ and the ciphertext is given by $c_t = h(z_t, m_t)$. Using $c_t = m_t \oplus z_t$ is the most common function also for self-synchronizing stream ciphers. The two types of stream ciphers behave very differently in terms of errors on the transmission channel.

- *Deletion or insertion of a ciphertext bit.* If a ciphertext bit is deleted or inserted during transmission, the sender and receiver will lose synchronization. For a synchronous cipher, assuming that c_x is inserted at time t_x, we then have[1]

$$m_t = \begin{cases} h^{-1}(z_t, c_x), & t = t_x \\ h^{-1}(z_t, c_{t-1}), & t > t_x \end{cases} \tag{2.4}$$

  and it is clear that the input to h is wrong whenever $t \geq t_x$. In a self-synchronizing stream cipher, assume that c_x is inserted at time t_x, then we have

$$m_t = \begin{cases} h^{-1}(g(c_{t-\delta}, c_{t-\delta+1}, \ldots, c_{t-1}, K), c_x), & t = t_x \\ h^{-1}(g(c_{t-\delta}, c_{t-\delta+1}, \ldots, c_x, K), c_{t_x}), & t = t_x + 1 \\ \vdots & \\ h^{-1}(g(c_x, c_{t-\delta}, \ldots, c_{t-2}, K), c_{t_x+\delta-1}), & t = t_x + \delta \\ h^{-1}(g(c_{t-\delta-1}, c_{t-\delta}, \ldots, c_{t-2}, K), c_{t-1}), & t > t_x + \delta \end{cases} \tag{2.5}$$

  and we see that whenever $t > t_x + \delta$ we get the same decryption, except for a time delay, as we would get without the insertion of an extra ciphertext bit. The error propagation will be at most δ bits, excluding the inserted bit.

- *bit error caused by noise on the channel.* If ciphertext bit c_{t_x} is modified, it is easy to see that, for a synchronous stream cipher, only the corresponding plaintext bit m_{t_x} will be affected. However, in a self-synchronizing stream cipher, also the following δ plaintext bits will be affected, since the modified ciphertext bit will be a part of the state σ_t, $t_x < t \leq t_x + \delta$.

[1] As an example, if we have $t_x = 3$, then $c_1, c_2, c_3, c_4, c_5 \ldots$ in the transmitter will become $c_1, c_2, c_x, c_3, c_4, c_5 \ldots$ in the receiver. Thus, c_t will appear at time $t+1$ in the receiver whenever $t \geq t_x$.

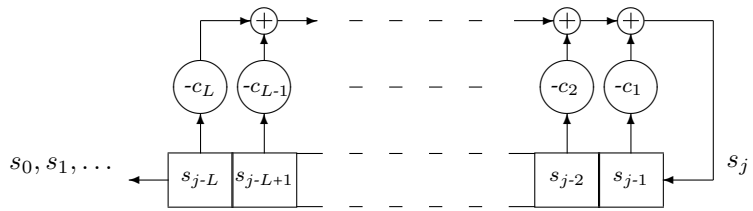

Figure 2.2: Model of a linear feedback shift register (LFSR).

Most stream ciphers, including all ciphers investigated or proposed in this thesis, are synchronous. Examples of self-synchronizing stream ciphers are Mosquito [DK05] and its successor Moustique [DK07] and SSS [RHPdV05]. Mosquito and SSS have been cryptanalyzed in [JM06] and [DLP05] respectively. The most common implementation of a self-synchronizing stream cipher is to use a block cipher in CFB mode.

2.2 Common Design Blocks

There are many ways to design stream ciphers. This section discusses some of the most common building blocks used in stream cipher design. There exist several more or less irrational designs that use building blocks and ideas not mentioned in this section.

2.2.1 Feedback Shift Registers

One of the most important and widely used design blocks in a stream cipher is the feedback shift register, and in particular, the *Linear Feedback Shift Register* (LFSR), see Fig. 2.2. An LFSR is well suited for hardware implementations and can also be implemented very efficiently in software. The input to an LFSR is a linear combination of its state variables. Let the size of the LFSR be denoted L and the coefficients determining the linear feedback function be denoted $c_1, c_2, \ldots, c_L \in \mathbb{F}_p$. The output of the LFSR is a sequence $\mathbf{s} = s_0, s_1, s_2, \ldots$ satisfying the relation

$$s_j = -c_1 s_{j-1} - c_2 s_{j-2} - \ldots - c_L s_{j-L}. \tag{2.6}$$

By introducing $c_0 = 1$ we get the shift register equation

$$\sum_{i=0}^{L} c_i s_{j-i}, \quad j \geq L. \tag{2.7}$$

The first L symbols $s_0, s_1, \ldots, s_{L-1}$ form the initial state of the LFSR. An LFSR is usually specified using its connection polynomial or feedback polynomial

$$C(D) = 1 + c_1 D + c_2 D^2 + \ldots + c_L D^L, \tag{2.8}$$

together with its length L.

Definition 2.3: A polynomial $C(D)$ of degree L with coefficients in the field $\mathbb{F}_p$ is said to be *irreducible* if it can not be written as the product of two polynomials both of degree $< L$ with coefficients in $\mathbb{F}_p$.

Primitive feedback polynomials are the most interesting polynomials for stream ciphers.

Definition 2.4: An irreducible polynomial $C(D)$ of degree L with coefficients in the field $\mathbb{F}_p$ is said to be *primitive* if the smallest positive integer n for which $C(D)$ divides $D^n - 1$ is $n = p^L - 1$.

From now on, we restrict our treatment to the case $p = 2$, i.e., the LFSR is defined over the field $\mathbb{F}_2$.

The output of an LFSR is periodic if $C(D)$ has degree L. The period of a sequence is the smallest positive integer T such that $s_t = s_{t+T}$ for all $t > 0$. The output sequence produced by an LFSR with primitive feedback polynomial $C(D)$ of degree L has period $2^L - 1$. This sequence is also called an m-sequence. In this case, the LFSR goes through all non-zero states before returning to the initial state. LFSRs with primitive feedback polynomial have very good statistical properties, which make them suitable for stream cipher applications.

- The distribution of k-bit patterns, $k \leq L$ is almost uniform. Looking at $2^L + k - 2$ consecutive output bits, each non-zero sequence of length k appears 2^{L-k} times and the zero sequence of length k appears $2^{L-k} - 1$ times.

- The autocorrelation function

$$R(i) = \frac{1}{2^L - 1} \sum_{t=0}^{2^L - 1} (2u_t - 1)(2u_{t+i} - 1) \tag{2.9}$$

  is small for $0 < i < 2^L - 1$. More specifically, we have

$$R(i) = \begin{cases} 1, & \text{if } i = 0, \\ -\frac{1}{N}, & \text{if } 0 < i < 2^L - 1. \end{cases} \tag{2.10}$$

Despite these nice statistical properties, an LSFR alone can not be used as a stream cipher. This is due to the linearity of the produced output bits. If we know L output bits, and the feedback polynomial, we can solve the corresponding system of linear equations and we can predict all other bits produced by the LFSR.

An LFSR can be used to produce *any* sequence of bits. If the output of a stream cipher can be produced by a relatively short LFSR of length L, then this LFSR can be found using $2L$ consecutive output bits. Then, all other output bits can be predicted because of the linearity. Using this as background, we define the *linear complexity*.

Definition 2.5: The *linear complexity* of a finite binary sequence $\mathbf{s} = s_0, s_1, \ldots$, denoted $L(\mathbf{s})$, is the length of the shortest LFSR that can generate $\mathbf{s}$.

The Berlekamp-Massey algorithm is an efficient algorithm that can determine the linear complexity $L(\mathbf{s})$ of a finite sequence $\mathbf{s}$ of length n. The basis for the algorithm is due to Berlekamp as a way to decode BCH codes, see e.g., [LC04]. Later, in [Mas69], Massey showed that the algorithm could be used to find the shortest LFSR that can produce a given sequence. The computational complexity of the Berlekamp-Massey algorithm is at most $O(n^2)$.

2.2.2 Boolean Functions

Boolean functions are very common in stream ciphers. A Boolean function $f(x_1, x_2, \ldots, x_n)$ on n variables may be viewed as a mapping from a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \in \mathbb{F}_2$, $1 \leq i \leq n$ to a single output bit,

$$f : \mathbb{F}_2^n \to \mathbb{F}_2. \tag{2.11}$$

The number of Boolean functions of n variables is 2^{2^n} and we denote the set of all Boolean functions on n variables by $\mathcal{B}_n$. A function $f(x_1, \ldots, x_n) \in \mathcal{B}_n$ can be defined by its *truth table*, i.e., a binary string of length 2^n,

$$f = [f(0,0,\ldots,0), f(1,0,\ldots,0), f(0,1,\ldots,0), \ldots, f(1,1,\ldots,1)], \tag{2.12}$$

defining the output for each possible input. A Boolean function f is $balanced$ if the truth table contains an equal number of 1's and 0's.

Any Boolean function $f(x_1, \ldots, x_n)$ can be written in one of several normal forms. In cryptology, the most common normal form is the *Algebraic Normal Form* (ANF). This represents the Boolean function as a polynomial over $\mathbb{F}_2$.

Definition 2.6: The *algebraic normal form* of a Boolean function $f(x_1, \ldots, x_n) \in \mathcal{B}_n$ is given by

$$a_0 \oplus \bigoplus_{1 \leq i \leq n} a_i x_i \oplus \bigoplus_{1 \leq i < j \leq n} a_{ij} x_i x_j \oplus \ldots \oplus a_{12\ldots n} x_1 x_2 \ldots x_n, \tag{2.13}$$

where the coefficients $a_0, a_{ij}, \ldots, a_{12\ldots n} \in \mathbb{F}_2$.

Other examples of normal forms are the disjunctive normal form and the conjunctive normal form. The *algebraic degree*, $\deg(f)$, is the number of variables in the highest order term in the ANF with non-zero coefficient. A Boolean function is *affine* if there exists no term of degree > 1 in the ANF. We denote the set of all affine functions of n variables by $A(n)$. An affine function with the constant term $a_0 = 0$ is called a *linear* function. The *Hamming weight* of a binary string S is the number of ones in the string. This number is denoted by $wt(S)$. The *Hamming distance* between two strings, S_1 and S_2 is denoted $d_H(S_1, S_2)$ and is the number of places where S_1 and S_2 differ. Note that $d_H(S_1, S_2) = wt(S_1 \oplus S_2)$. This allows us to define the *nonlinearity*.

Definition 2.7: The *nonlinearity* of a Boolean function $f \in \mathcal{B}_n$, denoted $nl(f)$, is the minimum distance from the set of all n-variable affine functions,

$$nl(f) = \min_{g \in A(n)} (d_H(f, g)). \tag{2.14}$$

Another important property is the *correlation immunity*, which is a measure of to which degree its output is correlated to some subset of its inputs. The mutual information $I(X;Y)$ is an information theoretic measure of the amount of information one random variable contains about another random variable.

Definition 2.8: Let the inputs to an n-variable Boolean function be random variables $X_1, X_2, \ldots, X_n$ and let the output be a random variable Z. Then a Boolean function with *correlation immunity* of order m satisfies

$$I(Z; X_{i_1}, X_{i_2}, \ldots, X_{i_m}) = 0, \qquad i_1 \leq i_2 \leq \ldots \leq i_m. \tag{2.15}$$

The immediate application to cryptology is the fact that a biased linear (or affine) non-constant approximation of an mth order correlation immune Boolean function must have at least $m + 1$ terms. An attack taking advantage of low correlation immunity is the correlation attack described in Section 2.3.4. A balanced mth order correlation immune function is called *m-resilient*. In [Sie84], Siegenthaler showed that there is an important trade-off between algebraic degree d and correlation immunity, namely that

$$m + d \leq n, \tag{2.16}$$

with strict inequality if the function is balanced[2].

[2]With the exception of the parity check function $X_1 \oplus X_2 \oplus \ldots \oplus X_n$ which is balanced, of degree $d = 1$ and correlation immune of order $m = n - 1$.

Many properties of Boolean functions can be described by the *Walsh transform*. Let $x = (x_1, \ldots, x_n)$ and $\omega = (\omega_1, \ldots, \omega_n)$ both belonging to $\mathbb{F}_2^n$ and $x \cdot \omega = x_1\omega_1 \oplus \ldots \oplus x_n\omega_n$. Let $f(x) \in \mathcal{B}_n$. Then the *Walsh transform* of $f(x)$ is a real valued function over $\mathbb{F}_2^n$ defined as

$$W_f(\omega) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus x \cdot \omega}. \tag{2.17}$$

Using the Walsh transform, a Boolean function f is balanced if and only if $W_f(0) = 0$. The nonlinearity of f is given by

$$nl(f) = 2^{n-1} - \frac{1}{2} \max_{\omega \in \mathbb{F}_2^n} |W_f(\omega)|. \tag{2.18}$$

A function is m-resilient (respectively mth order correlation immune) if and only if its Walsh transform satisfies

$$W_f(\omega) = 0, \forall \omega \in \mathbb{F}_2^n \text{ s.t. } 0 \leq wt(\omega) \leq m \text{ (respectively } 1 \leq wt(\omega) \leq m). \tag{2.19}$$

Parseval's Theorem states that the sum of the square of the Walsh transform is constant, $\sum_{\omega \in \mathbb{F}_2^n} W_f^2(\omega) = 2^{2n}$.

Bent functions are a special class of Boolean functions. They are characterized by the fact that the Walsh transform is constant. Combining (2.18) with Parseval's Theorem, it is easy to see that Bent functions have the highest possible nonlinearity, $nl(f) = 2^{n-1} - 2^{n/2-1}$, that Bent functions must have n even and that they cannot be balanced. A simple Bent function, used in Chapter 8, is

$$f(x_1, x_2, \ldots, x_n) = x_1x_2 \oplus x_3x_4 \oplus \ldots \oplus x_{n-1}x_n. \tag{2.20}$$

Webster and Tavares [WT86] introduced the concept of *Strict Avalanche Criterion* (SAC).

Definition 2.9: An n-variable Boolean function $f(x)$ satisfies the SAC if $f(x) \oplus f(x \oplus \alpha)$ is balanced for any $\alpha \in \mathbb{F}_2^n$ such that $wt(\alpha) = 1$.

The *Propagation Criterion* was introduced in [PLL+91] and is a generalization of SAC.

Definition 2.10: A Boolean function $f \in \mathcal{B}_n$ satisfies the propagation criterion $PC(l)$ of degree l if $f(x) \oplus f(x \oplus \alpha)$ is balanced for any $\alpha \in \mathbb{F}_2^n$ such that $1 \leq wt(\alpha) \leq l$.

Obviously SAC is equivalent to $PC(1)$. Higher order propagation criterion is motivated by the fact that some input bits to the Boolean function can be fixed in a known plaintext scenario.

Definition 2.11: A Boolean function $f \in \mathcal{B}_n$ satisfies the propagation criterion $PC(l)$ of degree l and order m if any function obtained from f by keeping m input bits constant satisfies $PC(l)$.

Higher order SAC was introduced in [For90]. SAC of order m is equivalent to $PC(1)$ of order m.

2.2.3 S-Boxes

S-boxes are very common in block ciphers, but are also widely used in stream cipher designs. Theory of S-boxes is not vital for the understanding of this thesis, but their importance motivates a short overview.

An (n, m) S-box is a mapping $f : \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^m}$. Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be the input to the S-box and let $\mathbf{y} = (y_1, y_2, \ldots, x_m)$ denote the output of the S-box. Then we can write

$$\mathbf{y} = f(\mathbf{x}) \quad \Rightarrow \quad \begin{cases} y_1 & = f_1(x_1, x_2, \ldots, x_n), \\ y_2 & = f_2(x_1, x_2, \ldots, x_n), \\ \vdots & \\ y_m & = f_m(x_1, x_2, \ldots, x_n). \end{cases} \tag{2.21}$$

The Boolean functions $f_1, f_2, \ldots, f_m$ are called component functions. An S-box can be represented by the truth tables or the ANF of the component functions. Sometimes it is also possible to represent an S-box as an operation in a finite field.

Basically, an S-box substitutes (nonlinearly) one value for another. Some S-boxes are static and thus assumed known to the adversary, but it is also common to have key dependent S-boxes. A static S-box can be designed to counter some specific attacks efficiently. A key dependent (or varying) S-box can not have optimum parameters at all times. On the other hand, they can be good in average and are unknown to the adversary.

The properties of S-boxes are closely related to those of Boolean functions. Let $\mathcal{L}$ be the set of all non-constant m-variable linear functions. Let $(l \circ f)(x) = l(f(x))$ be the n-variable Boolean function given by the composition of l and f. An (n, m) S-box is balanced if $l \circ f$ is balanced $\forall l \in \mathcal{L}$ and an (n, m) S-box is correlation immune of order t if $l \circ f$ is correlation immune of order t $\forall l \in \mathcal{L}$. Further, an (n, m) S-box is t-resilient if $l \circ f$ is t-resilient $\forall l \in \mathcal{L}$.

Definition 2.12: The *nonlinearity* of an S-box f is defined as the minimum nonlinearity of all non-zero linear combinations of the component functions

$$nl(f) = \min_{l} \{nl(l \circ f)\}. \tag{2.22}$$

```
i = j = 0;
while (1)
  i = (i + 1) mod 256;
  j = (j + S[i]) mod 256;
  Swap(S[i],S[j]);
  out = S[(S[i] + S[j]) mod 256];
```

Figure 2.3: The keystream generation phase in RC4.

The nonlinearity of an (n, m) S-box f is upper bounded by $nl(f) \leq 2^{n-1} - 2^{(n-2)/2}$ with equality only if n is even and $m \leq n/2$ [Nyb91].

Definition 2.13: The *algebraic degree* of an S-box f is defined as the minimum algebraic degree of all non-zero linear combinations of the component functions

$$nl(f) = \min_{l}\{\deg(l \circ f)\}. \tag{2.23}$$

Probably, the most well-known S-box today is the $(8, 8)$ S-box used in AES (Rijndael) [DR02]. The input is represented as an element in $\mathbb{F}_{2^8}$. The output is then defined as the multiplicative inverse[3] in the finite field followed by an affine transformation. In [Nyb93], it was shown that the inversion mapping in a finite field has high nonlinearity and that it has also very good properties in countering differential attacks. This particular S-box has also been used in several stream ciphers, e.g., SNOW 2.0 [EJ02], Hermes-8 [Kai05] and Hiji-bij-bij [Sar03].

2.2.4 Large Tables

Using large tables is extremely efficient in software and some of the fastest stream ciphers are based on this principle. As an example we use the most well-known and widely used stream cipher, RC4. RC4 uses a table S defining a permutation of all 256 possible bytes. Additionally, two index pointers i and j are used. The keystream generation phase is given in Fig. 2.3. Each iteration updates i and j, swaps two entries in the table and outputs an entry as a function of i and j. There are many cryptanalytic results on RC4, see e.g., [MS01, FM00, KMP+98, PP03, PP04a, Gol97b]. Other stream ciphers using large tables are VMPC [Zol04], RC4A [PP04b] and Py [BS05]. To the best of our knowledge, there is still no cipher based solely on a large table that does not suffer from (at least) distinguishing attacks. In [Max05b], attacks on VMPC and RC4A are given. For an attack on Py we refer to [WP07].

[3] The zero element is mapped onto itself.

2.2.5 T-functions

Compared to the previously described building blocks, *T-functions* are relatively new. The usage of T-functions was first proposed in 2002 by Klimov and Shamir [KS03]. Assume an n-bit state $S = (s_0, s_1, \ldots, s_{n-1})$, with s_0 being the least significant and s_{n-1} the most significant bit. A T-function updates every bit in the state as

$$s_i(t+1) = s_i(t) + f\left(s_0(t), s_1(t), \ldots, s_{i-1}(t)\right), \tag{2.24}$$

i.e., every bit is updated as a linear combination of itself and a function of its less significant bits. This bijective mapping is interesting in cryptography, especially because it is possible to find mappings resulting in S having a single cycle. In [KS03], it was shown that the state update function $S(t+1) = S(t) + (S^2(t) + C) \mod 2^n$ is a permutation with a single cycle of length 2^n if and only if, for the constant $C = (c_0, c_1, \ldots, c_{n-1})$, we have $c_0 = c_2 = 1$. Following [KS03], there has been several papers written on T-functions [KS04a, KS04b, KS05, HLYH05, Dau05]. There are also some recent stream cipher proposals based on T-functions, TSC-1, TSC-2 [HLYH05], TSC-3 [HLY+05], TSC-4 [MKH+06], and MIR-1 [Max05a].

2.2.6 Some Well-Known Stream Ciphers

In this section we give a few examples of existing stream ciphers. The designs are only briefly discussed and we refer to the design documents for a more detailed description.

Two classical stream cipher designs are the *nonlinear combiner* and the *nonlinear filter generator*. Many modern ciphers are based on one of these ideas and many attacks are also focused on cryptanalysis of these ciphers. The nonlinear combiner consists of a set of n LFSRs denoted $R_1, R_2, \ldots, R_n$. Denote the output of R_i at time t by $s_i(t)$. Then the keystream $z(t)$ is given as

$$z(t) = f(s_1(t), s_2(t), \ldots, s_n(t)), \tag{2.25}$$

where f is an n-variable nonlinear Boolean function. The nonlinear filter generator is an alternative to the nonlinear combiner and uses only one LFSR. The keystream is then given as a Boolean function of a subset of the LFSR bits. Fig. 2.4 shows the principle of the nonlinear combiner and the nonlinear filter generator. Constructing a secure stream cipher using one of these ideas is not very practical, given the development in cryptanalysis. This is e.g., shown in [BL05]. Instead, other design blocks are added to improve security. One example is the LILI stream ciphers, which add clock control to the nonlinear filter. LILI-128 has 2 LFSRs, R_c of size 39 and R_d of size 89. The function f_c takes, at time t, two bits from R_c as input and

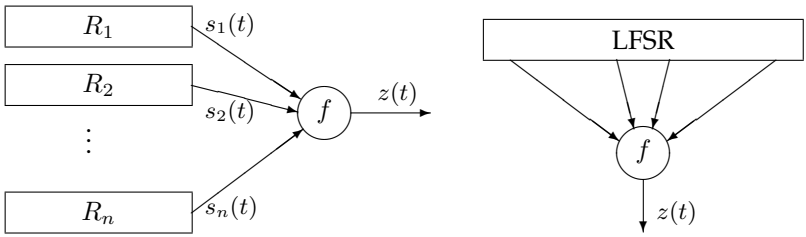

Figure 2.4: Principle of the nonlinear combiner (left) and the nonlinear filter generator (right).

outputs an integer $c(t) \in \{1, 2, 3, 4\}$. Then R_d is clocked $c(t)$ times. The keystream $z(t)$ is given as the output of a 10-variable Boolean function, taking inputs from R_d. The function f_d is chosen to be balanced, 3-resilient, has algebraic degree 6 and nonlinearity 480. Due to several attacks on LILI-128, an improved version denoted LILI-II was designed. It uses larger LFSRs, of size 128 and 127 respectively, and also a more complex Boolean output function f_d.

A variant of the nonlinear combiner is the summation generator. It was proposed by Rueppel [Rue86] and adds memory to the construction. The memory is a carry value $c(t)$ of $\lceil \log n \rceil$ bits. The keystream $z(t)$ and the carry update is given by

$$z(t) = s_1(t) \oplus s_2(t) \oplus \ldots \oplus s_n(t) \oplus c_0(t) \quad (2.26)$$

$$c(t+1) = \lfloor (s_1(t) + s_2(t) + \ldots + s_n(t) + c(t))/2 \rfloor \quad (2.27)$$

where $c_0(t)$ is the least significant bit of $c(t)$. The motivation for this design is that it is possible to avoid the tradeoff between degree and resiliency of the Boolean combining function as given in (2.16) by using memory. The stream cipher E_0 [Blu04], used in the Bluetooth standard is related to the summation generator. The difference is that the (2 bit) memory is updated in a slightly more complicated way.

The Achterbahn family of stream ciphers is another variant of the nonlinear combiner. Instead of using LFSRs, these ciphers deploy a set of nonlinear feedback shift registers. Cryptanalysis of this family is given in Chapter 7.

Some of the simplest stream ciphers are the shrinking generator and the self-shrinking generator. These are described in more detail in Chapter 4.

Several modern stream ciphers use word-based LFSRs. Using an LFSR defined over $\mathbb{F}_{2^{32}}$ allows the cipher to take advantage of the statistical properties provided by LFSR sequences and at the same time have an efficient

implementation in software. Probably, the most well-known cipher in this category is SNOW 2.0 [EJ02], an improvement of SNOW [EJ00]. The SNOW stream ciphers are inspired by the nonlinear filter generator. Instead of a Boolean output function, some LFSR state variables are taken as input to a finite state machine (FSM).

2.3 Methods of Cryptanalysis

2.3.1 Classifying the Attack

Before we discuss different cryptanalytic approaches, we need to introduce and define some basic terminology. Most attacks on stream ciphers belongs to one of two main categories depending on the goal of the attack.

- *Key recovery attack.* In a key recovery attack, the adversary will recover the secret key used in the cipher. The complexity of this attack is directly comparable to the complexity of trying all possible keys, an exhaustive search or a brute force attack, described in Section 2.3.2.
- *Distinguishing attack.* In a distinguishing attack, the adversary will not retrieve the secret key. Instead, the goal is to distinguish the keystream sequence produced by the stream cipher from a truly random sequence.

Any key recovery attack is also a distinguishing attack. A distinguishing attack is obviously much less powerful than a key recovery attack. It has been argued that a distinguishing attack with high complexity (but lower than exhaustive key search) does not render the cipher broken [RH02]. A distinguisher can be seen as a black box, taking a sequence of symbols as input and outputs either *cipher* or *random*. It is possible to imagine attacks falling outside or slightly in between a distinguishing attack and a key recovery attack. The attack on Pomaranch given in Section 6.9 is an example of this. That attack can be used as a distinguishing attack but it will also recover some part of the plaintext when only the ciphertext is known to the adversary. Another possible attack is a *state recovery attack*, in which the goal is to recover the state of the cipher.

Another classification of an attack is based on the amount of knowledge we assume that the adversary has. We distinguish between 4 different scenarios.

- *Ciphertext only attack.* The adversary has only knowledge of the encrypted string. In this case, the plaintext must have some redundancy in order for an attack to be applicable. The redundancy is usually the knowledge that the plaintext is English text or that the text is in ASCII

format and no special characters are used. In the latter case we then know that every eighth bit is zero.

- *Known plaintext attack.* The adversary knows both the plaintext and the corresponding ciphertext. This means that the adversary also has knowledge of the keystream.
- *Chosen plaintext attack.* In this scenario, the adversary can choose a plaintext and construct the corresponding ciphertext. This can also be seen as the adversary has access to the encryption device.
- *Chosen ciphertext attack.* Same as above, but the adversary has access to the decryption device and can construct plaintext from a chosen ciphertext.

Again, it is possible to imagine extensions to these scenarios. As an example, in a related key attack it is also assumed that the adversary knows that two different keys are related to each other in some known way, e.g., some bits are the same.

A third classification of an attack is at which part, the initialization phase or the keystream generation phase, of the cipher the attack targets.

- *Initialization phase.* In general, the cipher is reinitialized several times with the same key but with different IVs. If the key and/or the IV is not properly diffused into the state it is sometimes possible to get information about the key. These attacks can also be divided into *Chosen IV* attack and *Known IV* attacks, depending on to which extent the adversary can control the generation of IVs.
- *Keystream generation phase.* In general, only one initialization of the cipher is required and the keystream corresponding to this IV is used in the attack. The attack can take advantage of statistical or algebraic weaknesses in the keystream generation in order to recover the key or to distinguish the keystream from random.

2.3.2 Brute Force Attack

The easiest and most straight forward way of recovering the key is to search through all possible keys, known as a brute force attack[4]. Given a keysize of $|K|$ bits, this method will require at most $2^{|K|}$ tries, while the expected number of tries is $2^{|K|-1}$. Brute force, or exhaustive key search, in a known-plaintext scenario is applicable to all stream ciphers and the notion of breaking a cipher can be defined as finding a method that finds the secret key

[4]The brute force attack should not be confused by the literal meaning of the phrase in which the adversary, using physical abuse or public humiliation, retrieves the secret key.

faster than an exhaustive key search. Naturally, the size of the key in a cipher is chosen such that a brute force is impossible in practice. The most common choice for the key size is 128 or even 256 bits. In eSTREAM, a need for ciphers allowing a key size of 80 bits has been recognized, though the adequacy of these short keys has been disputed. As a comparison, the data encryption standard has a key size of 56 bits and because of U.S. Government restrictions a version of RC4 using key size of only 40 bits was sometimes used to "secure" IEEE 802.11 wireless networks.

The complexity of a brute force attack should not just be seen as the upper limit of the complexity of successful cryptanalysis. Considerable effort has also been put into building machines or clusters that can perform a brute force attack efficiently. In [Ber05], the author shows that many computers in parallel can be used in a brute force search, and the cost for the attack is also given. The author also argues that several attacks believed to be faster than brute force are in fact more expensive and should not be considered as breaking the algorithm.

The EFF DES Cracker [EFF07] is a DES cracking machine that was built in 1998. The cost of the machine is about $250000 and consists of 29 circuit boards with 64 chips on each. The machine can try all 2^{56} possible DES keys in about 5 days. This was almost 10 years ago and technology has developed since then. More recently, an FPGA based machine named Copacobana [KPP+06] was built, costing less than $10000. The machine is optimized for cryptographic hardware. It can brute force a DES key in about one week but it can also perform other cryptographic tasks.

2.3.3 Time-Memory Tradeoff Attacks

This attack has two phases, a *preprocessing phase* and a *real time phase*. The preprocessing phase collects information about the cipher and this information is saved in tables. In the real time phase, the precomputed tables together with data from an unknown key are used to find the key. The first time-memory tradeoff attack was given by Hellman in [Hel80] and was applied to block ciphers in a chosen plaintext scenario. Some notation in this section will be adopted from the published work on time-memory-tradeoff attacks and it will thus slightly differ from the notation in the rest of the thesis.

Let P be the plaintext block and $C = E_k(P)$ be the ciphertext block encrypted with cipher E using the key k. Let R be a reduction function. In the preprocessing phase we calculate m chains of keys as

$$k_i = f(k_{i-1}) = R(E_{k_{i-1}}(P)), \tag{2.28}$$

starting with some value k_1. Each chain is t keys long and thus we create a matrix containing mt keys, see Fig. 2.5. Only the m starting points

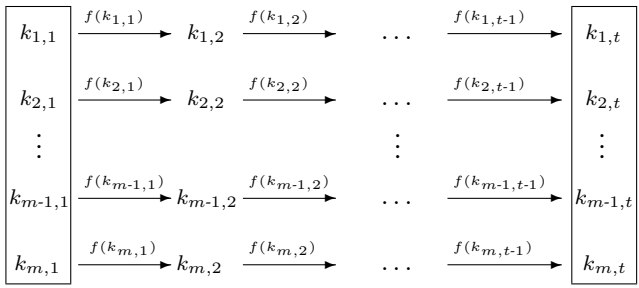

Figure 2.5: Matrix created in Hellman's attack.

$k_{1,1}, k_{2,1}, \ldots, k_{m,1}$ and the m endpoints $k_{1,t}, k_{2,t}, \ldots, k_{m,t}$ are saved, giving the tradeoff between the real time computational complexity and the memory needed. In the real time phase, the adversary observes a ciphertext and tries to find its predecessor in the chain (2.28). This is done by going forward in the chain until a saved endpoint is reached. When we know in which chain the key is, we can go forward from the corresponding starting point until we find the observed ciphertext k_i. The key used in that step, k_{i-1}, is then the key used in the encryption. If two values in the matrix collide, the two chains will merge and all subsequent values in that chain will also collide. We assume that the first m chains, covering mt keys, are disjoint. If N is the size of the search space, then, as long as $t \cdot mt \leq N$ the next chain is also disjoint with high probability. Thus, m and t are chosen such that $mt^2 = N$. An $m \times t$ matrix with $mt^2 = N$ will only cover a fraction of $1/t$ of the search space. To solve this, t matrices are created, all using a different reduction function R. Then, if two values in different tables intersect, they will not merge.

An improvement is to use distinguished points, referenced to Rivest in [Den82]. Instead of having chains of constant length, a chain is terminated when the value k_i is of a special form, e.g., the last $\log t$ bits are zero. This will significantly reduce the memory access. The value k_i is compared with the list of endpoints only if it is of the special form. Another advantage is that all merging chains will have the same endpoint and can thus be detected.

Another improvement are so called rainbow tables [Oec03]. Instead of having t tables of size $m \times t$ each, only one table is used of size $mt \times t$. The idea is to use a different type of chains, which can collide within the same table without merging. Each point in the chain uses a different reduction function, $R_1, R_2, \ldots, R_{t-1}$. Colliding chains will then only merge if they collide in the same point in the table, i.e., they merge with probability $1/t$.

This version of the time-memory tradeoff attack is widely used in practice to find passwords after intercepting their hash values.

Only one chosen plaintext block is used in the time-memory tradeoff attack on block ciphers and it is not clear how more plaintext can improve the attack. The tradeoff curve between time T and memory M in the attack is given by

$$TM^2 = N^2 \tag{2.29}$$

and a typical point on the curve is $T = M = N^{2/3}$. The preprocessing time is $P = N$.

In the case of stream ciphers, the situation is different. The size of the search space N is given by the number of possible states and since the keystream is determined only by the state[5], we can include data D as a part of the tradeoff, resulting in a time-memory-data tradeoff. The first time-memory-data tradeoff attack on stream ciphers was independently described by Golić [Gol97a] and Babbage [Bab95]. Each state is associated with the first $\log N$ bits of keystream produced from that state. In the preprocessing phase, M random states with corresponding $\log N$ bits of keystream are saved in memory and sorted by keystream. In the real time phase, the adversary is given a keystream sequence of $D+\log N-1$ bits and extracts all D possible windows of $\log N$ consecutive bits. For each window, the memory is logarithmically searched in order to find if it collides with a keystream sequence saved in the table. We expect to find a collision if $DM = N$. The preprocessing time is $P = M$ and the time complexity in the real time phase is $T = D$. The tradeoff curve is thus

$$TM = N. \tag{2.30}$$

A typical point on the curve is $T = M = N^{1/2}$. This suggests that any stream cipher should have a state at least twice the keysize.

The two attacks on block ciphers and stream ciphers can be combined to decrease the amount of memory and keystream needed, making the attack more suited for practical applications. Clearly, it is more practical to assume a limited amount of known data and memory is much more expensive than clock cycles. The combined approach was given by Biryukov and Shamir in [BS00]. The chain is produced by loading the state with the previously produced $\log N$ bits. Only the starting points and the endpoints are stored in memory. The number of points covered by the tables is $P = N/D$ with $N = mt^2$. Thus, with D observed keystream windows, the adversary will expect to find a collision. The tradeoff curve in this case is given by

$$TM^2D^2 = N^2 \tag{2.31}$$

[5]If the key and/or IV is used in the state update function, these are seen as a part of the state here.

with the restriction $D^2 \leq T \leq N$. It is possible to decrease the lower bound if the stream cipher has low sampling resistance. An example is the attack on LILI-128 given in [Saa02]. We refer to [BS00] for a more detailed analysis of sampling resistance. A typical point on this curve is $P = T = N^{2/3}$, $M = D = N^{1/3}$.

A more general framework for the time-memory-data tradeoff attack is given in [BMS06]. It is shown that the Babbage-Golić attack and the Biryukov-Shamir attack can be seen as special cases of this framework.

2.3.4 Correlation Attacks

One of the most important and studied attacks on stream ciphers are the correlation attacks. The target of correlation attacks is the nonlinear combiner using a nonlinear Boolean combining function f. We denote the ith LFSR used in the nonlinear combiner by R_i and its size by L_i. An exhaustive search, recovering the initial states of all LFSRs used in the nonlinear combiner would require

$$T = \prod_{i=1}^{M} 2^{L_i} \tag{2.32}$$

tries, where M is the total number of LFSRs. However, it is possible to do much better using a correlation attack. The correlation attack was first described in [Sie85]. It uses the fact that there is always a linear correlation between the output of a subset of the LFSRs and the output of the nonlinear Boolean function. This correlation can be found by looking at the Walsh transform of the Boolean function and can be written as

$$\Pr(z(t) = x_{i_1}(t) \oplus x_{i_2}(t) \oplus x_{i_3}(t) \oplus \ldots \oplus x_{i_m}(t)) = \frac{1}{2}(1 + \varepsilon), \tag{2.33}$$

where $z(t)$ is the keystream bit at time t and $x_{i_j}(t)$ is the output of register x_{i_j} at time t. The smallest subset of LFSRs that has to be considered in order to have $|\varepsilon| > 0$ is given by the resiliency[6] m of the Boolean function. For the mutual information between any subset of m' LFSR bits and the keystream bit z it holds

$$I(X_{i_1}, X_{i_2}, \ldots, X_{i_{m'}}; Z) = 0, \quad m' \leq m, \tag{2.34}$$

where X_i and Z are random variables. Moreover, there is at least one subset of $m + 1$ LFSR bits and the keystream bit z such that

$$I(X_{i_1}, X_{i_2}, \ldots, X_{i_{m'}}; Z) > 0, \quad m' = m + 1. \tag{2.35}$$

[6] We assume that the function f is balanced.

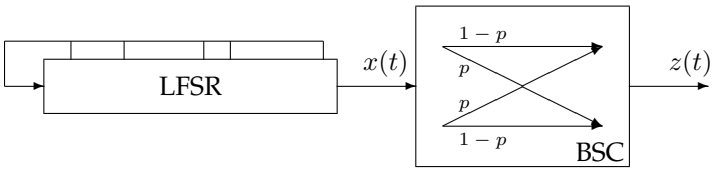

Figure 2.6: The keystream can be seen as the output of an LFSR sent over a binary symmetric channel.

Thus we need to consider $m' > m$ LFSR outputs in our linear approximation of f. If we consider the simplest case when $m = 0$, the computational complexity will be

$$T = \sum_i 2^{L_i} \approx \max(L_1, L_2, \ldots, L_M), \tag{2.36}$$

which is much faster than the exhaustive search attack complexity in (2.32).

The output bits of the LFSRs used in (2.33) are linearly generated so the sum

$$x(t) = x_{i_1}(t) \oplus x_{i_2}(t) \oplus x_{i_3}(t) \oplus \ldots \oplus x_{i_{m'}}(t) \tag{2.37}$$

can be expressed as the output of another, larger LFSR of size

$$L = lcm(L_{i_1}, L_{i_2}, \ldots, L_{i_{m'}}). \tag{2.38}$$

Further, the nonlinear Boolean combining function can be replaced by a binary symmetric channel (BSC). Fast correlation attacks are usually studied in this BSC model. The output of the generator is seen as a noisy version of the output of the LFSR. The cryptanalysis can then be seen as a decoding problem where the initial state bits of the LFSR are the information bits, the output of the LFSR is the corresponding codeword and the keystream is the received codeword after the introduction of a noisy channel with error probability $p \neq 0.5$, see Fig. 2.6. The number for keystream bits needed in order to decode the received codeword can be derived from the channel coding theorem. Let the rate of the code be $R = L/N$. The highest possible rate, called the capacity C, for the binary symmetric channel is

$$C_{BSC} = 1 - h(p), \tag{2.39}$$

where

$$h(p) = -p \log p - (1 - p) \log(1 - p) \tag{2.40}$$

is the binary entropy function. For any code with $R < C$ there exists a coding technique that can reconstruct the codeword at the receiver with arbitrarily low error probability. If $R > C$ arbitrarily low error probability is not achievable. Letting $C_{BSC} > L/N$ gives us a lower bound on the keystream length N needed for correct decoding,

$$N > \frac{L}{1 - h(p)}. \tag{2.41}$$

We can write $p = 0.5(1 + \varepsilon)$, where ε is called the bias or the imbalance. Then, for small values of p, we can use the Taylor series $\ln(1+x) \approx x - x^2/2$ together with $\log(1 + x) = \ln(1 + x)/\ln(2)$ and approximate $1 - h(p)$ as

$$\begin{aligned} 1 - h(p) &= 1 + p\log p(1-p)\log(1-p) \\ &= 1 + \frac{1}{2}(1+\varepsilon)\log\left(\frac{1}{2}(1+\varepsilon)\right) + \frac{1}{2}(1-\varepsilon)\log\left(\frac{1}{2}(1-\varepsilon)\right) \\ &\approx 1 + \frac{1}{2\ln 2}\left((1+\varepsilon)(\varepsilon - \frac{\varepsilon^2}{2} - \ln 2) - (1-\varepsilon)(\varepsilon + \frac{\varepsilon^2}{2} + \ln 2)\right) \\ &= 1 + \frac{\varepsilon^2 - 2\ln 2}{2\ln 2} = \frac{\varepsilon^2}{2\ln 2} \end{aligned} \tag{2.42}$$

Thus, the keystream length can be given as

$$N > \frac{2L\ln 2}{\varepsilon^2}. \tag{2.43}$$

The correlation attack only uses weaknesses found in the Boolean combining function. It does not take advantage of the linear feedback in the LFSRs and the fact that there are always parity check equations in the LFSR output bits that holds with probability 1. However, this property is used in the fast correlation attacks introduced by Meier and Staffelbach [MS89]. All fast correlation attacks are divided into two steps, a preprocessing step and a processing step. In the preprocessing step, parity check equations involving the output bits of the LFSR are found and in the processing step, these equations are used to find the initial state of the LFSR. For a more detailed analysis of fast correlation attacks we refer to [JÖ02].

2.3.5 Algebraic Attacks

Algebraic attacks are a relatively recent addition to the list of cryptanalysis techniques. Yet, the formulation of the idea dates back to Shannon [Sha49]. For several ciphers, algebraic attacks have been shown to be much more powerful than any other known attacks. The idea is to write the cipher as a system of equations involving only key bits and keystream bits. Since all

ciphers include a nonlinear part, this system of equations will be of degree ≥ 2. Solving a system of nonlinear equations over $\mathbb{F}_2$ is known to be an NP-complete problem even for quadratic equations. However, if the system is overdefined, things may get a little bit easier. Assume that we have M equations in n variables and of degree $d \leq n/2$. There are

$$T = \sum_{i=1}^{d} \binom{n}{i} \approx \binom{n}{d} \tag{2.44}$$

possible monomials and if $M \geq T$, each monomial can be replaced with a new variable and the resulting linear system can be solved. This is known as linearization. If not enough equations are available, other methods such as the XL algorithm or Gröbner bases can be used. This situation is more common in algebraic attacks on block ciphers and we refer to [CKPS00, Fau02] for more information on this.

In the case of stream ciphers based on LFSRs with linear feedback and a nonlinear Boolean output function f, each new known keystream bit will result in a new equation. Thus, it is favourable to require enough keystream bits in order to apply linearization instead of using the less efficient XL algorithm or Gröbner bases. Though, it should be noted that in the first attack on the stream cipher Toyocrypt in [Cou03b], the XL-algorithm was used. In that attack, f had to be approximated by a low degree function and the amount of keystream required for linearization was too much to keep the error probability sufficiently small. In [CM03], it was shown that even if f is of high degree, algebraic attacks could still be very efficient if there was a function g such that $f \cdot g$ was of low degree or 0. The year after, in [MPC04], this approach was generalized. It was shown that it is enough to find a function g of low degree such that $g \cdot f = 0$ or $g \cdot (f + 1) = 0$. This motivated a new property for a Boolean function f, the algebraic immunity, $AI(f)$.

Definition 2.14: The smallest degree of a function g such that $g \cdot f = 0$ or $g \cdot (f + 1) = 0$ is called the algebraic immunity of f, denoted $AI(f)$. The function g is called an annihilator of f.

In [MPC04], two algorithms for finding the algebraic immunity of a function f was given. The computational complexity of these algorithms was $O(D^3)$ with $D = \binom{n}{d}$, where d is the algebraic immunity i.e., the degree of g. An improved algorithm was given in [ACG+06] with computational complexity $O(D^2)$. Also, in [DT06] an algorithm with complexity $O(n^d)$ is given. In [DGM05], it is shown how to construct Boolean functions with good algebraic immunity.

An extension of algebraic attacks is the fast algebraic attack. In this attack, the degree of the equations is reduced in a precomputation step. We refer to [Cou03a, Arm04] for a more detailed description.

Algebraic attacks have also been shown to be applicable to nonlinear combiners using memory. The stream cipher used in Bluetooth is an example of a combiner with memory and in [AK03] it was shown that E0 could be broken by an algebraic attack. Some other stream ciphers shown to be vulnerable to algebraic attacks are Toyocrypt [Cou03b,CM03,Cou03a], LILI-128 [CM03, Cou03a], the summation generator [LKH$^+$04], SOBER-t16 and SOBER-t32 [CP04].

Another paper that deserves mentioning is [HR04] in which the authors show that all previous papers on fast algebraic attacks have underestimated the complexity of inserting the observed keystream into the equations. Moreover, an efficient way of doing this based on the fast Fourier transform is given.

2.3.6 Guess and Determine Attacks

A guess and determine attack can be divided into 3 main steps.

(*i*) A part of the internal state of the stream cipher is guessed. This can be e.g., a part of an LFSR.

(*ii*) Using relationships between internal state variables and the keystream, the rest of the state is determined.

(*iii*) The determined state is verified by running the generator forward, comparing the produced keystream $\tilde{\mathbf{z}} = (\tilde{z}_t, \tilde{z}_{t+1}, \ldots)$ with the known keystream $\mathbf{z} = (z_t, z_{t+1}, \ldots)$. If $\tilde{\mathbf{z}} \neq \mathbf{z}$, the guess was wrong and we return to step 1, making a new guess. Otherwise, if $\tilde{\mathbf{z}} = \mathbf{z}$ we found the correct state.

All parts of the guessed state in the first phase do not have to belong to the same time instant, but the state that is determined in the second phase must be the full state at a certain time. In this basic setting, the attack requires a very short known keystream. However, it is likely that the relationship between the state and the keystream requires too many variables to be guessed. Assuming some special relationship between variables can remove nonlinearity and keep the number of guessed values sufficiently small. Then the required known keystream and also the computational complexity is increased since the adversary has to repeat the attack until the assumed relationship is fulfilled. An example of this is the attack on SNOW by Hawkes and Rose in [HR02].

2.3.7 Side Channel Attacks

A side channel attack is fundamentally different from the previous described attacks. Instead of attacking the mathematical description of the crypto-

graphic algorithm, side channel attacks target the actual implementations of the algorithm. Both hardware and software implementations have been subjects to different attacks.

A timing analysis can give information about the secret key. *Timing attacks* are based on the time it takes for a device to perform operations. Different inputs take different amount of time to process.

In a *power analysis*, the idea is to measure the power used by a device, e.g., a smart card or an RFID tag. The power consumption of a hardware device is roughly proportional to the amount of bits being flipped at a certain time. This can be used to gain information about the secret key. Power analysis can be divided into *simple power analysis* (SPA) and *differential power analysis* (DPA). In SPA, the power consumption is directly interpreted in order to collect information about operations in a device or to retrieve key material. In DPA [KJJ99], statistical analysis of the power consumption used in several runs of the algorithm is used to retrieve information about the key.

2.4 Hypothesis Testing

One of the most important concepts in cryptanalysis, and in particular in stream cipher cryptanalysis, is hypothesis testing. In this section we give the background theory for hypothesis testing. For more details we refer to [CT91, Ch. 12].

The following notation will be used in this section. Random variables $X, Y, \ldots$ are denoted by capital letters and their domains are denoted by $\mathcal{X}, \mathcal{Y}, \ldots$. The realizations of random variables $x \in \mathcal{X}, y \in \mathcal{Y}, \ldots$ are denoted by small letters. A distribution is denoted by P. The probability function of a random variable X following the distribution P is denoted by $\Pr_P[x]$ and sometimes as just $P[x]$.

We start by defining the relative entropy between two distributions.

Definition 2.15: The relative entropy between two probability mass functions $\Pr_{P_0}[x]$ and $\Pr_{P_1}[x]$ over the same domain $\mathcal{X}$ is defined as

$$D\left(\Pr_{P_0}[x] \| \Pr_{P_1}[x]\right) = \sum_{x \in \mathcal{X}} \Pr_{P_0}[x] \log \frac{\Pr_{P_0}[x]}{\Pr_{P_1}[x]}. \tag{2.45}$$

In literature the relative entropy is sometimes also referred to as information divergence or Kullback-Leibler distance. For convenience and ease of reading, in the following we write the relative entropy as $D(P_0 \| P_1)$ or $D(P_0[x] \| P_1[x])$. Note that in general $D(P_0 \| P_1) \neq D(P_1 \| P_0)$.

In a binary hypothesis test we observe a collection of independent and identically distributed data. Denote the distribution of the observed data by

P_{obs}. We consider two hypotheses, the null hypothesis H_0 and the alternate hypothesis H_1:

$$H_0 : P_{obs} = P_0, \tag{2.46}$$
$$H_1 : P_{obs} = P_1. \tag{2.47}$$

We define a *decision rule* which is a function $\delta : \mathcal{X} \to \{0, 1\}$ such that

$$\delta = \begin{cases} 0, & P_{obs} = P_0, \\ 1, & P_{obs} = P_1. \end{cases} \tag{2.48}$$

The function δ makes a decision for each $x \in \mathcal{X}$. The decision rule divides the domain $\mathcal{X}$ into two regions denoted $\mathcal{A}$ and $\mathcal{A}^c$. $\mathcal{A}$ is called the *acceptance region* of δ and corresponds to the decision to accept the null hypothesis.

There are two types of errors associated with a binary hypothesis test. We can reject the null hypothesis when it is in fact true, i.e., $\delta(x) = 1$ when $P_{obs} = P_0$. This is called a type I error and the probability of this error is denoted α. The other alternative is that we accept the null hypothesis when the alternate hypothesis is true, i.e., $\delta(x) = 0$ when $P_{obs} = P_1$. This is called a type II error. The probability of this error is denoted β.

How to perform the optimal hypothesis test is given by the Neyman-Pearson lemma.

Lemma 2.1 (Neyman-Pearson): Let $X_1, X_2, \ldots, X_m$ be drawn i.i.d. according to mass function P_{obs}. Consider the decision problem corresponding to the hypotheses $P_{obs} = P_0$ vs. $P_{obs} = P_1$. For $T \geq 0$ define a region

$$A_m(T) = \left\{ \frac{P_0(x_1, x_2, \ldots, x_m)}{P_1(x_1, x_2, \ldots, x_m)} > T \right\}. \tag{2.49}$$

Let $\alpha_m = P_0^m(\mathcal{A}_m^c(T))$ and $\beta_m = P_1^m(\mathcal{A}_m(T))$ be the error probabilities corresponding to the decision region $\mathcal{A}_m$. Let $\mathcal{B}_m$ be any other decision region with associated error probabilities α^* and β^*. If $\alpha^* \leq \alpha$, then $\beta^* \geq \beta$.

If we want the two probabilities of error to be equal we should set $T = 1$. Assuming that all samples are independent, (2.49) is equivalent to

$$A_m(T) = \left\{ \sum_{i=1}^{m} \log \left(\frac{P_0(x_i)}{P_1(x_i)} \right) > \log T \right\}. \tag{2.50}$$

No general expression for the error probabilities α and β exists. Hence, we know how to perform the optimal test, but we do not know the performance of the test (in the general case). However, there exist *asymptotic* expressions for the error probabilities, i.e., expressions that hold when the

number of samples is large. The relative entropy is related to the asymptotic error probabilities through Stein's lemma. Stein's lemma states that if we fix the error probability α then β decreases so that

$$\lim_{n\to\infty} \frac{\log \beta}{n} = -D(P_0\|P_1). \tag{2.51}$$

The value of α does not affect the exponential rate at which β decreases and according to Stein's lemma, this situation always occurs. According to (2.51) we can asymptotically write

$$\beta \approx 2^{-nD(P_0\|P_1)}. \tag{2.52}$$

From (2.52) we can derive the following. Assume that we have a set of 2^L sequences. We know that $2^L - 1$ of them are drawn from a uniform distribution P_1 and one is drawn from a biased distribution P_0. The goal is to determine which sequence is drawn from the biased distribution. For a fixed α, the expected number of misclassified sequences is about

$$(2^L - 1) \cdot 2^{-nD(P_0\|P_1)}. \tag{2.53}$$

If we put this ≈ 1 the number of samples needed to find the biased sequence is roughly given by

$$N \approx \frac{L}{D(P_0\|P_1)}. \tag{2.54}$$

In cryptanalysis it is common that the two distributions P_0 and P_1 are very close to each other. We denote the *bias* of a distribution by ε. In cryptology literature, there are two definitions of this bias for a binary distribution, namely

$$\Pr(X = 0) = 0.5 + \varepsilon_{(1)}, \tag{2.55}$$
$$\Pr(X = 0) = 0.5(1 + \varepsilon_{(2)}), \tag{2.56}$$

where we temporarily distinguish them by index. When considering the sum of k binary *independent* variables $(X_1, X_2, \ldots, X_k)$, the bias $\varepsilon_{(i)tot}$ of the sum is given differently depending on what definition has been used,

$$\varepsilon_{(1)tot} = 2^{k-1}\varepsilon_{(1)}^k, \tag{2.57}$$
$$\varepsilon_{(2)tot} = \varepsilon_{(2)}^k. \tag{2.58}$$

(2.57) is also known as the *piling-up lemma* and (2.58) can be seen as a variant. Note that we always have $\varepsilon_{(2)} = 2\varepsilon_{(1)}$. We will adopt the second notation and from here on we drop our temporary index.

Now, consider a binary distribution, we want to determine if a random variable X is drawn from the cipher distribution P_0 with

$$\begin{array}{rcl} \Pr_{P_0}(X=0) & = & 0.5(1+\varepsilon), \\ \Pr_{P_0}(X=1) & = & 0.5(1-\varepsilon), \end{array} \tag{2.59}$$

or if it is from the uniform distribution P_1 with $\Pr_{P_1}(X=0) = \Pr_{P_1}(X=1) = 0.5$. We have

$$\begin{aligned} D(P_0\|P_1) &= \frac{1}{2}(1+\varepsilon)\log\frac{\frac{1}{2}(1+\varepsilon)}{\frac{1}{2}} + \frac{1}{2}(1-\varepsilon)\log\frac{\frac{1}{2}(1-\varepsilon)}{\frac{1}{2}} \\ &= 1-h(p) \overset{(2.42)}{\approx} \frac{\varepsilon^2}{2\ln 2}. \end{aligned} \tag{2.60}$$

Combining (2.54) and (2.60) allows us to write the number of samples needed to find one biased distribution out of 2^L distributions as

$$N \approx \frac{L \cdot 2\ln 2}{\varepsilon^2}. \tag{2.61}$$

We see that this agrees with (2.43), the lower bound of samples needed in a correlation attack. If we put $L = 1$, i.e., when there are two possible initial states, we get a very common rule of thumb, widely used in cryptanalysis. When the two error probabilities α and β are about equal, a distinguisher needs asymptotically

$$N \approx \frac{1}{\varepsilon^2} \tag{2.62}$$

samples to determine if an observed distribution is the cipher distribution or the uniform distribution. Note that asymptotically, there is no difference if we put 1 or $2\ln 2$ in the numerator. For a more rigorous treatment of the number of samples needed in a distinguisher, the Chernoff information should be used. Using the Chernoff information $C(P_0, P_1)$, an asymptotic expression that minimizes the overall probability of error $P_e = \pi_0\alpha + \pi_1\beta$, where π_i are the *a priori* probabilities of the distributions, is achieved. The asymptotic error probability can then be written as

$$P_e \approx 2^{-nC(P_0,P_1)}, \tag{2.63}$$

where

$$C(P_0, P_1) = -\min_{0\le\lambda\le 1} \log\left(\sum_x \Pr{}^{\lambda}_{P_0}[x]\Pr{}^{1-\lambda}_{P_1}[x]\right). \tag{2.64}$$

The Chernoff information between two distributions is often hard to find since we have to minimize over λ. A common way of avoiding this is to

pick a value of λ, e.g., $\lambda = 0.5$. This will give a lower bound for the Chernoff information and thus an upper bound for the error probability. For more details we refer to [CT91, Ch. 12].

We end this section on hypothesis testing by considering the error probabilities when the number of samples are given by multiples of (2.62) and (2.61). Assume that the cipher distribution P_0 is given by (2.59) and let P_1 be the uniform distribution. The expected values and standard deviations are given by

$$\begin{aligned} E_{P_0}\{X\} &= \tfrac{1}{2}(1-\varepsilon), \quad & \sigma_{P_0} &= \tfrac{1}{2}\sqrt{(1-\varepsilon^2)} \\ E_{P_1}\{X\} &= \tfrac{1}{2}, & \sigma_{P_1} &= \tfrac{1}{2}. \end{aligned} \tag{2.65}$$

Let $S = \sum_{i=1}^{N} X_i$ denote the sum of N values. If N is large, then according to the central limit theorem, S is normally distributed with

$$\begin{aligned} E_{P_0}\{S\} &= \tfrac{N}{2}(1-\varepsilon), \quad & \sigma_{P_0} &= \tfrac{1}{2}\sqrt{N(1-\varepsilon^2)} \\ E_{P_1}\{S\} &= \tfrac{N}{2}, & \sigma_{P_1} &= \tfrac{\sqrt{N}}{2}. \end{aligned} \tag{2.66}$$

The two distributions are given in Fig. 2.7. The two error probabilities are given by

$$\alpha = \int_{N/2(1-\varepsilon/2)}^{\infty} \frac{1}{\sigma_{P_0}\sqrt{2\pi}} e^{-\frac{(t-E_{P_0}\{S\})^2}{2\sigma_{P_0}^2}} dt \tag{2.67}$$

$$\beta = \int_{-\infty}^{N/2(1-\varepsilon/2)} \frac{1}{\sigma_{P_1}\sqrt{2\pi}} e^{-\frac{(t-E_{P_1}\{S\})^2}{2\sigma_{P_1}^2}} dt \tag{2.68}$$

Assume that we want to determine if an observed distribution $P_{obs} = P_0$ or $P_{obs} = P_1$. If the *a priori* probabilities are equal, $\pi_0 = \pi_1 = 0.5$, and if ε is small, then the error probability is given by $P_e = \alpha = \beta$. We write (2.62) as

$$N = \frac{d}{\varepsilon^2}, \tag{2.69}$$

and for different values of d we get the error probabilities given in Table 2.1.

In (2.61) we assume that we have a set of 2^L sequences and only one follows the cipher distribution P_0. Assume for simplicity that each test is independent. The error probability for each test is $P_e = \pi_0\alpha + \pi_1\beta$ with $\pi_0 = 1/2^L$ and $\pi_1 = (2^L - 1)/2^L$. If we let $\pi_0\alpha = \pi_1\beta$ we get

$$\alpha = (2^L - 1)\beta. \tag{2.70}$$

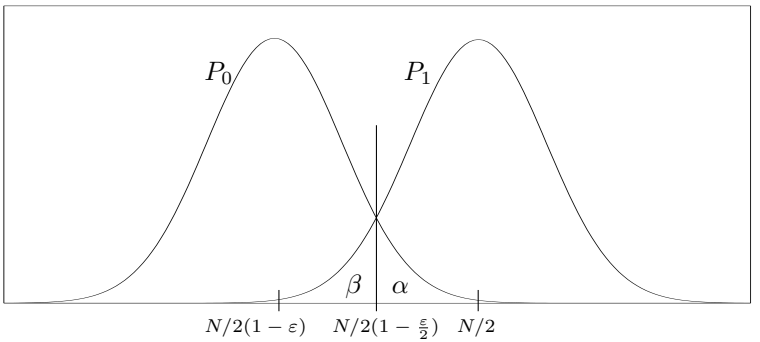

Figure 2.7: The two distributions P_0 and P_1.

d	1	$2\ln 2$	2	3	4	5	10	20
P_e	0.31	0.28	0.24	0.19	0.16	0.13	0.06	0.013

Table 2.1: The error probability when the number of samples is as given in (2.69), for different choices of d.

For the 2^L hypothesis tests we want to make the correct decision for *all* sequences. Hence, the total error probability is given by

$$P_e = 1 - \left((1-\beta)^{2^L - 1} (1-\alpha) \right). \tag{2.71}$$

We write (2.61) as

$$N = \frac{d \cdot L \cdot 2\ln 2}{\varepsilon^2}. \tag{2.72}$$

Using (2.70) and (2.71) gives the error probabilities in Table 2.2. Throughout this thesis $d = 1$ will be used.

We can note that in cryptanalysis it is not always necessary that *only* the sequence distributed according to P_0 stems from P_0 also in the test. It is possible that we can allow many sequences distributed according to P_1 to appear to be distributed according to P_0, i.e., several sequences are in the region β in Fig. 2.7. Then we have a set of sequences where perhaps each sequence corresponds to one value of the secret key. Then each of these key candidates can be tested individually. In this case we could benefit from decreasing α to avoid missing the correct key candidate at the expense of more key candidates.

P_e		d			
		1	2	3	4
L	3	0.54	0.31	0.18	0.10
	5	0.55	0.24	0.10	0.04
	10	0.56	0.14	0.03	$2^{-7.8}$
	20	0.57	0.05	$2^{-9.0}$	2^{-14}
	50	0.59	$2^{-8.7}$	2^{-20}	2^{-31}
	100	0.61	2^{-15}	2^{-37}	2^{-60}

Table 2.2: The error probability when the number of samples is as given in (2.72), for different choices of d and L.

2.5 Summary

In this chapter, an introduction to stream ciphers has been given. We have discussed the motivation to study stream ciphers and given a formal description of the primitive. Some of the possible design blocks have been introduced, such as LFSRs, Boolean functions and S-boxes. An overview of possible cryptanalysis methods has also been given, e.g., correlation attacks and algebraic attacks. Finally, we have presented some mathematical background to hypothesis testing which will be shown to be very useful in the following chapters.

3

Correlation Attacks Using a New Class of Weak Feedback Polynomials

Correlation attacks and fast correlation attacks, discussed in Section 2.3.4, have received lots of attention. The development in this area of cryptanalysis has inspired several new designs. The fast correlation attack [MS88] is very effective if the feedback polynomial has a special form, namely, if its weight is very low. Due to fast correlation attacks, it is a well-known fact that one avoids low weight feedback polynomials in the design of LFSR based stream ciphers. This chapter identifies a new class of such weak feedback polynomials, namely, polynomials of the form

$$f(x) = g_1(x) + g_2(x)x^{M_1} + \ldots + g_l(x)x^{M_{l-1}}, \tag{3.1}$$

where $g_1, g_2, \ldots, g_l$ are all polynomials of low degree. For such feedback polynomials, we identify an efficient correlation attack in the form of a distinguishing attack.

This chapter is based on [EHJ04] and is outlined as follows. In Section 3.1 we give the basic preliminaries for the attack. In Section 3.2 we discuss how a basic distinguishing attack is mounted when we have an LFSR with a low weight feedback polynomial. This basic attack is then extended to a general case in Section 3.3 using vectors with noise variables. Section 3.4 discusses what happens when the parameters in the attack are changed. Section 3.5 discusses the problem of finding a multiple of the characteristic polynomial. In Section 3.6 the new attack is compared to the basic attack and in Section 3.7 the chapter is summarized.

3.1 Preliminaries

The model used for the attack is the standard model for a correlation attack, illustrated in Fig. 2.6. The target stream cipher is a binary additive stream cipher and uses two different components, one linear and one nonlinear. The linear part is an LFSR and the nonlinear part can be modeled as a black box. The nonlinear function can then, through a linear approximation, be seen as a binary symmetric channel (BSC) with crossover probability p (correlation probability $1-p$) with $p \neq 0.5$. This model was previously discussed in Section 2.3.4.

The output bits from the LFSR are denoted s_t, $t = 0, 1, \ldots$, and the keystream bits are denoted z_t, $t = 0, 1, \ldots$. From the BSC it follows immediately that $\Pr(s_t = z_t) = 1 - p \neq 0.5$. Assuming a known plaintext attack, the goal in our distinguishing attack is the following. Given the observed keystream sequence $\mathbf{z} = z_0, z_1, \ldots$ we want to distinguish the keystream from a truly random sequence. Recalling Section 2.4, let P_0 be the cipher distribution and P_1 the uniform distribution. We try to determine if the distribution P_{obs} for the observed samples $(z_t, t = 0, 1, \ldots)$ is more likely to be P_0 than P_1, or vice versa.

3.2 A Basic Distinguishing Attack From a Low Weight Feedback Polynomial

We start our investigation by simplifying the Meier-Staffelbach approach and turn their original ideas into a distinguishing attack. Referring to the assumed model (Fig. 2.6), the observed output is considered as a noisy version of the sequence from the LFSR,

$$z_t = s_t \oplus e_t, \tag{3.2}$$

where e_t, $t = 0, 1, \ldots$, are variables representing the noise introduced by the approximation. The noise has a biased distribution

$$\Pr(e_t = 0) = 0.5(1 + \varepsilon), \tag{3.3}$$

where $|\varepsilon| \leq 1$ is nonzero (but usually rather small). In the following, we assume that all noise variables e_t are independent. The bits s_t produced by the LFSR with feedback polynomial $\pi(x) = 1 + c_1 x + \ldots + c_L x^L$ will satisfy the linear recurrence relation

$$\bigoplus_{j=0}^{L} c_j s_{t-j} = 0, \quad t \geq j. \tag{3.4}$$

By adding the corresponding positions in the keystream $\mathbf{z}$, all s_t cancel out, leaving only noise variables. In more detail, if we introduce

$$x_t = \bigoplus_{j=0}^{L} c_j z_{t-j}, \tag{3.5}$$

then

$$x_t = \bigoplus_{j=0}^{L} c_j z_{t-j} = \bigoplus_{j=0}^{L} c_j s_{t-j} \oplus \bigoplus_{j=0}^{L} c_j e_{t-j} = \bigoplus_{j=0}^{L} c_j e_{t-j}. \tag{3.6}$$

Since the distribution of the noise e_t introduced by the BSC is nonuniform it is possible to distinguish the sample sequence x_t, $t = L, L+1, \ldots$, from a truly random sequence using a hypothesis test. If we assume the binary case (all variables are binary), the sum x_t of the noise will have probability

$$\Pr\left(\bigoplus_{j=0}^{L} c_j e_{t-j} = 0\right) = 0.5(1+\varepsilon^w), \tag{3.7}$$

where $w = wt(c_0, c_1, \ldots, c_L)$, i.e., the weight of $\pi(x)$. By (2.62), we know that the required number of samples for a successful attack is in the order of $1/(\varepsilon^{2w})$.

Note that the ideas behind this simple attack has appeared in many attack scenarios before, even if it might not have been described exactly in this context before. We see that the weight of the feedback polynomial is directly connected to the success of the attack.

3.3 A More General Distinguisher Using Vectors

As we have seen in the previous section, low weight polynomials should be avoided in order to resist the attack. When the weight of $\pi(x)$ grows linearly, the required keystream and the computational complexity in the attack grows exponentially. At some point we might require more computing power than an exhaustive key search and the attack is no longer interesting (to apply). In this section we describe a similar attack but with a more general approach that can be applied to another set of feedback polynomials. However, to simplify the presentation we will use the characteristic polynomial of the LFSR instead of the feedback polynomial. The characteristic polynomial is simply the reciprocal of the feedback polynomial.

Consider a length L LFSR with characteristic polynomial

$$f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 + \ldots + f_L x^L, \tag{3.8}$$

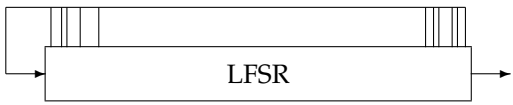

Figure 3.1: The characteristic polynomial $a(x) = g_1(x) + x^M g_2(x)$ corresponds to an LFSR with taps concentrated to two groups.

where $f_i \in \mathbb{F}_2$. We try to find a multiple, $a(x)$, of the characteristic polynomial $f(x)$ such that $a(x)$ can be written as

$$a(x) = h(x)f(x) = g_1(x) + x^M g_2(x), \tag{3.9}$$

where $g_1(x)$ and $g_2(x)$ are polynomials of some small degree $\leq k$. A special case is when $h(x) = 1$, i.e., when $f(x)$ is already on the desired form. The problem of finding a multiple of the form (3.9) will be addressed in Section 3.5. For now, we assume that such a multiple is given. The characteristic polynomial in (3.9) corresponds to an LFSR with taps concentrated to two groups far away from each other as shown in Fig. 3.1. The linear recurrence relation can then be written as the two sums istic polynomial in (3.9) corresponds to an LFSR with taps concentrated to two groups far away from each other as shown in Fig. 3.1. The linear recurrence relation can then be written as the two sums

$$\bigoplus_{i=0}^{k} s_{t+i}a_i \oplus \bigoplus_{i=0}^{k} s_{t+M+i}a_{M+i} = 0, \tag{3.10}$$

where $a_i,\ i = 0, 1, \ldots, L$ are the coefficients in the characteristic polynomial $a(x)$. We now consider the standard model for a correlation attack where the output of the cipher is considered as a noisy version of the LFSR sequence $z_t = s_t \oplus e_t$. The noise variables e_t are introduced by the approximation of the nonlinear part of the cipher. Furthermore, the biased noise has distribution $\Pr(e_t = 0) = 0.5(1 + \varepsilon)$ and as before, we assume that the noise variables are independent. Now we denote by Q_t the sum

$$Q_t = \bigoplus_{i=0}^{k} z_{t+i}a_i \oplus \bigoplus_{i=0}^{k} z_{t+M+i}a_{M+i} = \bigoplus_{i=0}^{k} e_{t+i}a_i \oplus \bigoplus_{i=0}^{k} e_{t+M+i}a_{M+i}. \tag{3.11}$$

This can also be written as

$$\begin{aligned} Q_0 &= e[0,k]\cdot \underline{g_1} \oplus e[M, M+k]\cdot \underline{g_2}, \\ Q_1 &= e[1,k+1]\cdot \underline{g_1} \oplus e[M+1, M+k+1]\cdot \underline{g_2}, \\ &\vdots \\ Q_{V-1} &= e[V-1, V+k-1]\cdot \underline{g_1} \oplus e[M+V-1, M+V+k-1]\cdot \underline{g_2}, \end{aligned} \tag{3.12}$$

by introducing $e[i,j] = (e_i, \ldots, e_j)$ for $i \le j$ and $\underline{g_1} = (g_{1,0}, g_{1,1}, \ldots, g_{1,k})^T$ where $g_{1,j}$, $j = 0, 1, \ldots, k$ are the coefficients of the $g_1(x)$ polynomial. A corresponding notation is assumed for $\underline{g_2}$.

The interesting observation here is that even though the noise variables (e_t, $t = 0, 1, \ldots$) are independent, Q_i values close to each other will not be independent in general. This is because of the fact that several Q_i will contain common noise variables. We take advantage of this fact by moving to a vector representing the noise as follows.

We introduce the vectorial noise vector E_t of length V as

$$E_t = (Q_{V\cdot t}, \ldots, Q_{V(t+1)-1}). \tag{3.13}$$

Alternatively, E_t can be expressed as

$$E_t = (e_{V\cdot t}, \ldots, e_{V(t+1)+k-1})G_1 \oplus (e_{V\cdot t+M}, \ldots, e_{V(t+1)+M+k-1})G_2, \tag{3.14}$$

where G_1 and G_2 are the $(V+k)\times V$ matrices

$$G_1 = \begin{pmatrix} g_{1,0} & & & \\ g_{1,1} & g_{1,0} & & \\ \vdots & \vdots & & g_{1,0} \\ g_{1,k} & g_{1,k-1} & \cdots & g_{1,1} \\ & g_{1,k} & & \vdots \\ & & & g_{1,k} \end{pmatrix} \quad G_2 = \begin{pmatrix} g_{2,0} & & & \\ g_{2,1} & g_{2,0} & & \\ \vdots & \vdots & & g_{2,0} \\ g_{2,k} & g_{2,k-1} & \cdots & g_{2,1} \\ & g_{2,k} & & \vdots \\ & & & g_{2,k} \end{pmatrix}.$$

To prepare the attack, we derive the distribution of the noise vector E_n given in (3.14). This can be easily done for moderately small k and V. This distribution is the cipher distribution, denoted P_0. The uniform distribution is denoted P_1. The simulation of P_0 together with the search for $a(x)$ can be seen as the preprocessing step.

In the processing step, we collect a sample sequence

$$Q_0, Q_1, Q_2, \ldots, Q_{N'}, \tag{3.15}$$

by

$$Q_t = \bigoplus_{i=0}^{k} z_{t+i}a_i \oplus \bigoplus_{i=0}^{k} z_{t+M+i}a_{M+i}. \tag{3.16}$$

1. Find a multiple $a(x)$.
2. For $t = 0 \dots N'$
   $Q_t = \bigoplus_{i=0}^{k} z_{t+i} a_i \oplus \bigoplus_{i=0}^{k} z_{t+M+i} a_{M+i}$
   end for.
3. For $t = 0 \dots N$
   $E_t = (Q_{V \cdot t}, \dots, Q_{V(t+1)-1})$
   end for.
4. Calculate $I = \sum_{t=0}^{N} \left(\log_2 \frac{P_0(E_t)}{P_1(E_t)} \right)$.

If $I > 0$ then output **cipher**, else output **random**.

Figure 3.2: Summary of the proposed algorithm.

This sample sequence is then transformed into a vectorial sample sequence $E_0, E_1, \dots, E_N$ by (3.13).

The final step is to use optimal hypothesis testing, i.e., the Neyman-Pearson lemma, to decide whether $E_0, E_1, \dots, E_N$ is most likely drawn from P_0 or P_1. The proposed algorithm is summarized in Fig. 3.2. An estimate of the number N of vectors needed in the attack is given by $N = 1/\varepsilon^2$. We define

$$\varepsilon^2 = 2^V \sum_{i=0}^{2^V - 1} \varepsilon_i^2 \tag{3.17}$$

where $\varepsilon_i = \Pr_{P_0}(X = i) - 2^{-V}$. This definition of ε^2 follows from an expansion of $D(P_0||P_1)$ similar to the expansion done in (2.60). It will satisfy the approximation (2.60) for any value of V and can thus be seen as a generalization of the ε^2 used in Section 2.4.

The performance of the algorithm depends on the polynomials, g_i, that are used. Fig. 3.3 shows an example of how the number of vectors required for a successful attack depends on the vector length for a certain combination of two polynomials. In the example, we have used $\Pr(e_t = 0) = 9/16$ ($\varepsilon = 2^{-3}$). Moreover, we assume that M is large enough so that noise variables from $g_1(x)$ and $g_2(x)$ in the vector are independent. The case $V = 1$ corresponds to the basic approach described in Section 3.2 and we see that increasing the vector length will decrease the number of vectors needed. Note that $g_1(x)$ and $g_2(x)$ are just two examples of what the polynomials might look like, they do not represent a multiple of any specific polynomial.

It is easy to generalize our reasoning with two groups to allow finding a multiple with arbitrarily many groups. Thus, we search for

$$a(x) = h(x)f(x) = g_1(x) + x^{M_1} g_2(x) + \dots + x^{M_{l-1}} g_l(x), \tag{3.18}$$

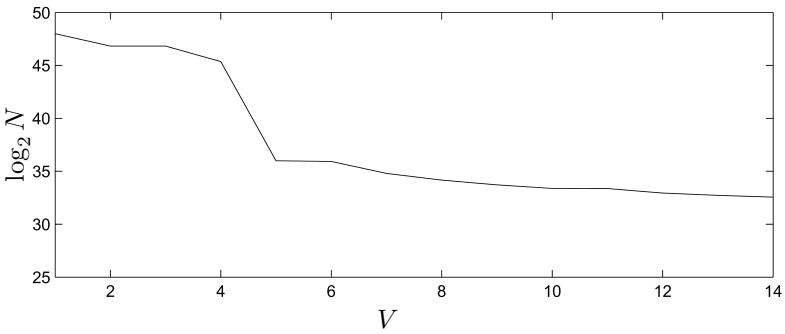

Figure 3.3: The number of vectors needed as a function of the vector length V. In this example $g_1(x) = 1 + x + x^5 + x^6$ and $g_2(x) = 1 + x + x^7 + x^8$.

where $g_i(x)$ is a polynomial of some small degree $\leq k$ and $M_1 < M_2 < \ldots < M_{l-1}$. It is clear that when l grows it is easier to find $a(x)$ with the desired properties. However, it is also clear that when l grows, the attack becomes weaker since variables stemming from different groups will be independent in the vector E_t.

3.4 Tweaking the Parameters in the Attack

The previous section described the details of the attack algorithm. In this section we take a closer look at the effect different parameters has on performance.

3.4.1 How $g_i(x)$ Affects the Results

The attack takes advantage of the fact that the same noise variable appears in several entries in the same vector. Exactly how the noise variables are located in the vectors depends on the polynomials $g_i(x)$ and different polynomials gives different distributions. Assume that $a(x)$ is of the form (3.9). We assume that the value M, i.e., the distance between the polynomials, is large. Thus, we can treat the polynomials $g_1(x)$ and $g_2(x)$ independently. In the preprocessing phase we can first simulate the distribution $P_0^{(1)}$ using noise variables determined by $g_1(x)$. Then we simulate $P_0^{(2)}$ using variables only determined by $g_2(x)$. The total cipher distribution P_0 is then given by

$$P_0 = P_0^{(1)} + P_0^{(2)}, \tag{3.19}$$

(100010001)	$\Gamma = \{5\}$	(101101011)	$\Gamma = \{3\}$	(101111001)	$\Gamma = \{4\}$
(100111111)	$\Gamma = \{2\}$	(100100101)	$\Gamma = \{\emptyset\}$	(110000011)	$\Gamma = \{7\}$
(101010001)	$\Gamma = \{7\}$	(101110111)	$\Gamma = \{2, 3, 4\}$	(111110101)	$\Gamma = \{2, 3, 6\}$
(110101111)	$\Gamma = \{2, 3, 5\}$	(111010101)	$\Gamma = \{3\}$	(110011111)	$\Gamma = \{2\}$
(101100111)	$\Gamma = \{4\}$	(110110111)	$\Gamma = \{2, 3\}$	(101010101)	$\Gamma = \{3\}$
(101000101)	$\Gamma = \{5\}$	(101011100)	$\Gamma = \{3, 4\}$	(100010110)	$\Gamma = \{9\}$
(101111101)	$\Gamma = \{2, 3\}$	(111100101)	$\Gamma = \{5\}$	(111101000)	$\Gamma = \{2, 4\}$

Table 3.1: Some examples of polynomials and corresponding set Γ.

where + is bitwise addition in $\mathbb{F}_2$ of the binary vectors. The distribution P_0 depends not only on the individual properties of $P_0^{(1)}$ and $P_0^{(2)}$, but also on how they relate to each other in terms of tap positions in $g_1(x)$ and $g_2(x)$. There is no simple rule to determine if two polynomials $g_1(x)$ and $g_2(x)$ results in an efficient attack, but we highlight a few important properties that will affect the result.

Let us start by looking at the distribution $P_0^{(i)}$ determined by $g_i(x)$. The distribution of vectors up to length 15 have been simulated for all possible polynomials $g(x)$ of degree $k \leq 8$. In the examples given below, the polynomial will be represented by a bit string as

$$g(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_8 x^8 \Longleftrightarrow (c_0, c_1, c_2, \ldots, c_8) \quad c_i \in \mathbb{F}_2. \tag{3.20}$$

Most polynomials have a certain vector length, denoted V_γ, at which the bias increases significantly compared to the previous vector lengths. Some polynomials have several such V_γ points. The set of all V_{γ_i} is denoted Γ. The efficiency of the attack is primarily determined by the following two properties of $g(x)$.

- *The weight of the polynomial.* The weight of $g(x)$ equals the amount of noise variables in each vector entry. Many noise variables usually means smaller bias and this is of course the case for $V = 1$. However, as V increases this is not always the case due to the correlation between neighbouring Q_i. As an example (101100111) has higher bias than (101100101) for $V > 3$ even though it has more noise variables.

- *Arrangement of the terms in $g(x)$.* This is equivalent to the arrangement of the taps in the LFSR. Many taps close together means that the same noise variable occurs more frequently in the noise vector E_t. The arrangement of the taps will also influence V_γ. Table 3.1 shows several examples of polynomials $g(x)$ and their corresponding set Γ.

The properties of the individual distributions $P_0^{(1)}$ and $P_0^{(2)}$ for increasing V gives a hint about the properties of the total distribution P_0 when V is increased. However, it is not possible to predict the properties for any pair

and it is usually necessary to simulate P_0 in order to know if the attack will succeed. Below we give a few rules of thumb for the *typical* behaviour when combining two distributions.

- Combining $g_1(x)$ and $g_2(x)$ with

$$\Gamma_{g_1} \cap \Gamma_{g_2} \neq \{\emptyset\} \tag{3.21}$$

  will often result in $\Gamma_a = \Gamma_{g_1} \cap \Gamma_{g_2}$. Sometimes we only get $\Gamma_a = \min\{\Gamma_{g_1} \cap \Gamma_{g_2}\}$ and if the intersecting values are large we can get $\Gamma_a = \{\emptyset\}$.

- Combining $g_1(x)$ and $g_2(x)$ with

$$\Gamma_{g_1} \cap \Gamma_{g_2} = \{\emptyset\} \tag{3.22}$$

  will often result in $\Gamma_a = \min\{\Gamma_{g_1} \cup \Gamma_{g_2}\}$. If the smallest value is large it is more probable to have $\Gamma_a = \{\emptyset\}$.

Fig. 3.4 gives several examples of the behaviour of P_0 when the two distributions $P_0^{(1)}$ and $P_0^{(2)}$ are combined. As before, we have used $\Pr(e_t = 0) = 9/16$ ($\varepsilon = 2^{-3}$).

3.4.2 Increasing Vector Length

The idea of using V larger than one is that we can get correlation between vector entries. Increasing V can never decrease the bias of the distribution P_1. This property will be proved in the cryptanalysis of Pomaranch in Chapter 6. Thus the number of vectors needed in the distinguishing attack will decrease (or remain the same) for each increase of V. However, increasing V also increases the computational complexity for simulating the distribution P_1 and also require more memory to save the distribution. Our simulations have showed that it is extremely rare to have a $V_\gamma > 10$ when we restrict $k \leq 8$. This suggest that V can be chosen to be in the order of k.

3.4.3 Increasing the Number of Groups l

As will be shown in Section 3.5.2, it is much easier to find a multiple $a(x)$ if we increase l. However, the resulting distribution

$$P_0 = \sum_{i=1}^{l} P_0^{(i)} \tag{3.23}$$

will become much more uniform for increasing l, resulting in a very high attack complexity. This is similar to the binary case in which more taps in the multiple will result in a more uniform distribution.

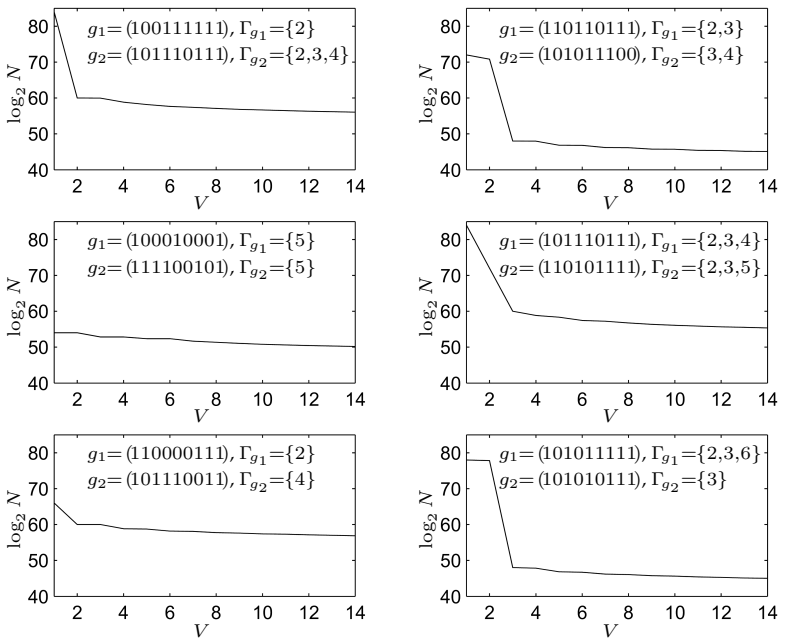

Figure 3.4: More examples of resulting distributions P_0. The number of vectors N needed in a distinguisher is given as a function of the vector length V

.

3.5 Finding a Multiple of the Form $a(x)$

We have until now assumed that a multiple of the characteristic polynomial has a certain form (3.18). This section considers the problem of finding multiples of the desired form.

3.5.1 Finding Low Weight Multiples

According to the piling-up lemma, see Section 2.4, the distribution P_0 becomes more uniform if the polynomial is of a high weight. If the feedback polynomial is not already of low weight, the first step in the correlation attack given in Section 3.2 is to find a low weight multiple. Obviously, the linear relation given by the multiple will satisfy the LFSR sequence produced by the feedback polynomial. The number of keystream bits needed to compute the first sample is given by the degree of the multiple. It turns out that there is a tradeoff between the weight and the degree of a multiple (in the

M_c	w												
	2	3	4	5	6	7	8	9	10	11	12	13	14
$L = 100$	2^{100}	$2^{50.5}$	$2^{34.2}$	$2^{26.1}$	$2^{21.4}$	$2^{18.2}$	$2^{16.0}$	$2^{14.4}$	$2^{13.2}$	$2^{12.2}$	$2^{11.4}$	$2^{10.7}$	$2^{10.2}$
$L = 500$	2^{500}	2^{250}	2^{168}	2^{126}	2^{101}	$2^{84.9}$	$2^{73.2}$	$2^{64.4}$	$2^{57.6}$	$2^{52.2}$	$2^{47.8}$	$2^{44.1}$	$2^{41.0}$
$L = 1500$	2^{1500}	2^{750}	2^{501}	2^{376}	2^{301}	2^{252}	2^{216}	2^{189}	2^{169}	2^{152}	2^{139}	2^{127}	2^{118}

Table 3.2: The critical degree M_c of the multiple as a function of L and w.

expected case). If we want to find a multiple of weight w of a polynomial of degree L it was shown by Golić in [Gol96] that the critical degree M_c at which multiples of weight w start to appear is

$$M_c = (w-1)!^{\frac{1}{w-1}} 2^{\frac{L}{w-1}}. \tag{3.24}$$

As a reference, Table 3.2 lists the expected degree of the lowest degree weight w multiple for $L = 100, 500$ and 1500.

3.5.2 Finding Multiples With Groups

Let us first consider the problem of finding a multiple of the form

$$a(x) = h(x)f(x) = g_1(x) + x^M g_2(x), \tag{3.25}$$

where $f(x)$ is the degree L characteristic polynomial of the cipher. This can be found by polynomial division. Assume that we have a polynomial $g_2(x)$ of degree $\leq k$. This polynomial is multiplied by x^i. The result is then divided by the LFSR characteristic polynomial $f(x)$. Thus, we can write

$$\frac{x^i g_2(x)}{f(x)} \Rightarrow x^i g_2(x) = f(x) \cdot q(x) + r(x), \tag{3.26}$$

where $q(x)$ and $r(x)$ are the quotient and the reminder respectively. There are 2^k ways to choose $g_2(x)$ and i can be chosen in M ways. Since $\deg f(x) = L$, we know that the remainder $r(x)$ satisfies $0 \leq \deg(r(x)) < L$. If $\deg r(x) \leq k$ we have found an acceptable $g_1(x)$. The probability that $\deg r(x) \leq k$ is given by

$$\Pr(\deg(r(x)) \leq k) = 2^{k-L}. \tag{3.27}$$

If we want it to be probable to find at least one such polynomial, we need

$$2^k \cdot M \cdot \frac{2^k}{2^L} \geq 1 \;\; \Rightarrow \;\; M \geq 2^{L-2k}. \tag{3.28}$$

M		l				
		2	3	4	5	6
k	3	2^{94}	2^{47}	2^{31}	2^{24}	2^{19}
	4	2^{92}	2^{46}	2^{31}	2^{23}	2^{18}
	5	2^{90}	2^{45}	2^{30}	2^{23}	2^{18}
	6	2^{88}	2^{44}	2^{29}	2^{22}	2^{18}
	7	2^{86}	2^{43}	2^{29}	2^{22}	2^{17}
	8	2^{84}	2^{42}	2^{28}	2^{21}	2^{17}

Table 3.3: The degree M of the multiple as a function of k and l for $L = 100$.

We see that for modest values of k the length of the multiple will become very large, comparable to (3.24). We can extend the reasoning to the case with arbitrarily many groups, i.e., we look for a multiple of the form

$$a(x) = h(x)f(x) = g_1(x) + x^{M_1}g_2(x) + \ldots + x^{M_{l-1}}g_l(x). \tag{3.29}$$

Using the same reasoning as above we get the new expression

$$2^k \cdot M_1 \cdot 2^k \cdot M_2 \cdot \ldots \cdot 2^k \cdot M_{l-1} \cdot \frac{2^k}{2^L} \geq 1 \;\Rightarrow\; M \geq 2^{\frac{L-lk}{l-1}}, \tag{3.30}$$

where we assume that $M_1, M_2, \ldots, M_{l-1} \leq M$. This gives us an upper bound on all M_i.

Comparing (3.24) and (3.30) we see that increasing l is the most efficient way to lower the degree of the multiple. However a larger l, as stated in Section 3.4.3, will significantly decrease the bias. Table 3.3 lists some values on M needed to find a multiple, for some chosen values of k and l.

3.6 Comparing the Proposed Attack With a Basic Distinguishing Attack

The applicability of the algorithm is twofold. The first case is if the characteristic polynomial is of the form $f(x) = g_1(x) + g_2(x)x^{M_1} + \ldots + g_l(x)x^{M_{l-1}}$. Applying the basic algorithm given in Section 3.2 to these LFSRs without first finding a multiple is equivalent to applying the new algorithm with $V = 1$. In this case, the new algorithm is a significant improvement since using vectors increases the bias. Of course, using the basic algorithm without first finding a multiple is very naive, but if the length L of the LFSR is

large the degree of a low weight multiple will also be large, see (3.24). Thus, if $f(x)$ is of high degree then our algorithm can be more effective.

Secondly, the new algorithm can be applied to arbitrary characteristic polynomials. In this case we first find a multiple of the polynomial that is of the form $a(x) = h(x)f(x) = g_1(x) + g_2(x)x^{M_1} + \ldots + g_l(x)x^{M_{l-1}}$ and then apply the algorithm. Comparing the two equations (3.24) and (3.30) we see that it is not much harder to find a polynomial of some weight w than it is to find a polynomial with the same number of groups. Although our algorithm takes advantage of the fact that the taps are close together, this is still not enough to compensate for the larger amount of noise variables. Thus, in this case the proposed attack will give improvements only for certain specific instances of characteristic polynomials, e.g., those having a surprisingly low degree multiple of the form $a(x)$ but no low weight multiples where the degree is surprisingly low. This attack may be viewed as a new design criterion. *One should avoid LFSRs where multiples of the form* (3.9) *are easily found.*

3.7 Summary

Through a new correlation attack, we have identified a new class of weak feedback polynomials, namely, polynomials of the form $f(x) = g_1(x) + g_2(x)x^{M_1} + \ldots + g_l(x)x^{M_{l-1}}$, where $g_1, g_2, \ldots, g_l$ are all polynomials of low degree. The correlation attack has been described in the form of a distinguishing attack. This was done mainly for simplicity, since the theoretical performance is easily calculated and we can compare with the basic attack based on low weight polynomials.

A next possible step in this direction would be to examine the possibility of turning these ideas into a key recovery attack. This could be done in a similar manner as in the Meier-Staffelbach approach. For example, we could try to derive many different relations (multiples) and apply some iterative decoding approach in vector form. The theoretical part of such an approach will probably be much more complicated.

4

Two New Attacks on the Self-Shrinking Generator

The *nonlinear combiner* and the *nonlinear filter generator* discussed in Section 2.2.6 introduces nonlinearity into linearly generated LFSR sequences by a nonlinear Boolean function, taking LFSR variables as inputs. Another way to introduce nonlinearity into an LFSR sequence is to *decimate* the sequence in an irregular way. This method is used in, among others, the shrinking generator proposed by Coppersmith, Krawczyk and Mansour [CKM93] in 1993 and in the self-shrinking generator introduced by Meier and Staffelbach [MS94] in 1994. Since the introduction of the shrinking and self-shrinking generators several attacks have been proposed on both. Though, despite the simplicity of the constructions, their security is still remarkably high and all attacks are exponential in the size of the LFSRs.

In this chapter we introduce two new key recovery attacks on the self-shrinking generator. One attack that requires a short keystream and one attack that requires a long keystream. The previously best known attack using short keystream has complexity $O(2^{0.6563L})$ [Kra02]. We show that our new attack using short keystream has approximately the same complexity as this attack. However, while the attack in [Kra02] needs an infeasible amount of memory, our attack works with very small memory complexity. The second attack uses a keystream length exponential in the length of the LFSR. The attack is first described in a basic variant, considering l consecutive keystream bits. Then the attack is extended to a more general version, considering two or more segments simultaneously. The computational complexity for this attack is significantly better than any previously known attack.

This chapter is based on [HJ06c] and is outlined as follows. Section 4.1 gives a brief description of the shrinking and self-shrinking generators. In

Section 4.2 we describe the previous attacks on the self-shrinking generator, both those using short keystream and those requiring a longer one. In Section 4.3 a description of the new attack that requires a short keystream sequence is given and in Section 4.4 we describe the attack requiring long keystream. This attack is extended to a more general version in Section 4.5. The chapter is summarized in Section 4.6.

4.1 Description of the Self-Shrinking Generator

In this section we give a description of the shrinking generator and the self-shrinking generator. Our attacks target the self-shrinking generator but the idea behind this generator is based on the idea behind the shrinking generator. These two generators probably represent the simplest stream ciphers imaginable. Still, their security is surprisingly high. The shrinking generator was introduced in 1993 by Coppersmith, Krawczyk and Mansour [CKM93]. It is based on two LFSRs, R_A and R_S. The sequence generated by R_S is used to select which bit to output from the sequence generated by R_A. If $s_t = 1$ then output a_t as a keystream bit, otherwise a_t is discarded. If the sequence generated by R_S is balanced, the rate of the shrinking generator is $1/2$, i.e., in average we need to clock the R_A and R_S two times in order to produce one keystream bit. We say that the sequence a_t is *irregularly decimated*.

The self-shrinking generator is a variant of the shrinking generator and it is even simpler. It consists of only one LFSR R of length L and was introduced in 1994 by Meier and Staffelbach [MS94]. The shift register R outputs a sequence $\mathbf{s} = (s_0, s_1, s_2 \ldots)$. The bits are divided into pairs, (s_{2t}, s_{2t+1}) and if $s_{2t} = 1$ the generator will output s_{2t+1} as a keystream bit. If $s_{2t} = 0$ no output is generated. The keystream sequence is denoted $\mathbf{z} = (z_0, z_1, z_2 \ldots)$. After two clockings, a keystream bit is output with probability $1/2$, after four clockings with probability $1/4$ etc. Let μ be the number of clockings needed to produce one keystream bit. The expected number of clockings, $E\{\mu\}$, needed for each keystream bit, z_t, is thus

$$E\{\mu\} = \sum_{i=0}^{\infty} 2i\frac{1}{2^i} = \sum_{i=0}^{\infty} \frac{i}{2^{i-1}} = 4. \tag{4.1}$$

Hence, the rate of the self-shrinking generator is $1/4$, half of that of the shrinking generator. The variance of the number of clockings, $V\{\mu\}$, per keystream bit is

$$V\{\mu\} = \sum_{i=0}^{\infty} (2i)^2\frac{1}{2^i} - 4^2 = 8. \tag{4.2}$$

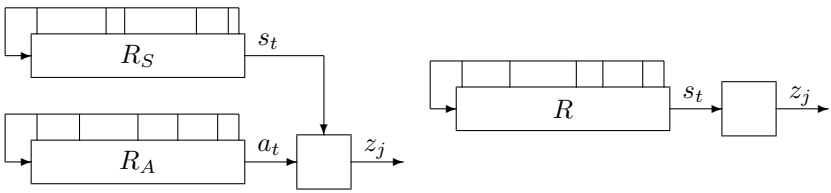

Figure 4.1: The shrinking generator (left) and the self-shrinking generator (right).

Decimation algorithm for the shrinking generator

```
i = 0; j = 0;
while (1)
    if (s[i] == 1)
        z[j] = a[i];
        j++;
    i++;
```

Decimation algorithm for the self-shrinking generator

```
i = 0; j = 0;
while (1)
    if (s[i] == 1)
        z[j] = s[i+1];
        j++;
    i += 2;
```

Figure 4.2: The algorithms used in the shrinking generator and the self-shrinking generator to irregularly decimate a binary sequence.

In [MS94] it was shown that the sequence produced has period $P \geq 2^{\lfloor L/2 \rfloor}$ and the linear complexity $L(\mathbf{z}) \geq 2^{\lfloor L/2 \rfloor - 1}$. The self-shrinking generator has the advantage over the shrinking generator that it only uses one LFSR, but on the other hand it has a lower rate. The two generators are shown in Fig. 4.1 and the algorithms to irregularly decimate the sequences are given in Fig. 4.2. The shrinking generator and the self-shrinking generator are no specific cipher proposals in the sense that all parameters are specified. They are instead just proposals of how to introduce nonlinearity into a linearly generated sequence. The length of the LFSR(s) as well as the feedback polynomial(s) are left to the designer to decide. In fact, the designs can operate on any kind of binary sequences. They do not have to be generated by an LFSR, though this is the most common and easy way since we know that LFSRs can produce sequences with several good statistical properties, see Section 2.2.1. Moreover, LFSRs are very efficient in hardware. In the following we will assume that the sequence used as input to the self-shrinking

algorithm is an m-sequence, generated by an LFSR with primitive feedback polynomial.

4.2 Previous Attacks on the Self-Shrinking Generator

Almost all proposed attacks are key recovery attacks and they can be divided into two categories, those that recover the key from a short known keystream, in the order of L, and those that need a longer keystream, exponential in L, to succeed. Of course, any attack using a short keystream sequence can be used on a long keystream sequence as well by just using a small part of it. Consequently, the long keystream attacks are only interesting if they require less computational complexity than the short keystream attacks.

This section will provide a brief overview of the previous key recovery attacks. The list of attacks covered might not be exhaustive, but all well-known and significant attacks prior to the results in this chapter will be mentioned.

4.2.1 Short Keystream Attacks

In the original paper [MS94], Meier and Staffelbach proposed an attack with complexity $2^{0.75L}$ that reconstructs the internal state with only a few known keystream bits. For each bit pair there are 3 different valid possibilities, namely $(0,0)$, $(0,1)$ and $(1,z_t)$. The probability for $(0,0)$ and $(0,1)$ are $1/4$ and the probability for $(1,z_t)$ is $1/2$. Thus, the entropy of a bit pair is

$$H = -(1/2)\log_2(1/2) - 2\cdot(1/4)\log_2(1/4) = 3/2. \tag{4.3}$$

The entropy per bit is $3/4$ so an exhaustive search among all different cases in the order of probability would require $2^{0.75L}$ time. In the same paper attacks are also shown that have a slightly lower complexity but they only apply if the feedback polynomial of the LFSR is of very low weight.

In 2001, Zenner, Krause and Lucks [ZKL01] showed how to determine the initial state in time $O(2^{0.694L})$. The attack is based on a backtracking approach. They build a binary search tree based on the bits s_{2t}. Each node will give one or two equations depending on the value of s_{2t}. When the number of equations are more than $L-1$, the feedback polynomial is used to represent the new equations as equations in the variables defining the starting state. If the new equation is linearly dependent of the earlier ones and this linear dependency leads to a contradiction that branch of the tree is ignored and backtracking starts. Whenever a branch is found that gives

L linearly independent equations, the system of equations is solved and a candidate initial state is obtained.

In 2002, Krause introduced the concept of BDD-based attacks [Kra02]. Attacks on several generators were presented, including A5/1, E0 and the self-shrinking generator. The attack on the self-shrinking generator was shown to have complexity $O(2^{0.656L})$. It is shown that approximately $4.82L$ free binary decision diagrams (FBDD) have to be computed. The size of an FBDD is given as the number of vertices and for each of the FBDDs the size is upper bounded by $L^{O(1)}2^{0.656L}$. The main problem with this attack is that the memory requirement to store one of these FBDDs is infeasible. Using rough estimates, even if the complexity suggests that it would be feasible to attack the self-shrinking generator with an LFSR size of about 100, the memory requirements limits the attack to an LFSR of size only about 60.
Remark. In all attacks above only the factor exponential in L has been given. The last two attacks also has a factor polynomial in L. The significance of this factor will decrease as L increases and asymptotically only the exponential factor is necessary. Similarly, in our attacks given in this chapter we focus on the exponential factor, though we will also discuss the size of the polynomial factor.

4.2.2 Long Keystream Attacks

In [Mih96] Mihaljević presented an attack that has better complexity than the attacks given in the original paper. However, this attack requires the attacker to know a keystream of length exponential in L. Assume that the attacker has

$$N \geq l2^{L/2}\binom{L/2}{l}^{-1} \tag{4.4}$$

keystream bits available. Then it is possible to mount an attack with time complexity $O(2^{L-l})$. In the attack, l is chosen to be the highest number such that (4.4) is still satisfied. Then the attacker guesses that in the current state of the LFSR, l out of the $L/2$ positions corresponding to s_{2t} are ones. Consequently, the remaining $L/2 - l$ positions corresponding to s_{2t} are zeros. If this is the case the current internal state will produce exactly l keystream bits. The attacker then looks at keystream segments of size l. There are $2^{L/2}$ ways of choosing bits for the positions corresponding to s_{2t} and l ones can be chosen in $\binom{L/2}{l}$ ways. Hence, the probability that the output segment of length l is generated by the internal state with l ones in positions corresponding to s_{2t} is $2^{-L/2}\binom{L/2}{l}$. From this it follows that the number of keystream segments that need to be checked before we can expect to find a segment giving us the key is given by $2^{L/2}\binom{L/2}{l}^{-1}$. The values s_{2t+1} when

$s_{2t} = 0$ are unknown to the attacker so these $2^{L/2-l}$ positions are exhaustively searched in the attack. It is shown that the attack is successful with high probability after 2^{L-l} steps. Choosing $l = 0.25L$ gives a computational complexity of $2^{0.75L}$ steps and choosing $l = 0.5L$ gives a computational complexity of $2^{0.5L}$. It is worth noting that the L bits recovered are consecutive and there is no polynomial factor that represents solving a system of equations in the attack complexity.

4.3 New Attack Using Short Keystream

In this section we will give a detailed description of our first attack. This attack recovers the initial state (key) of the LFSR if a short keystream sequence is available.

Assume that we know m keystream bits,

$$z_t, z_{t+1}, z_{t+2}, \ldots, z_{t+m-1}. \tag{4.5}$$

Each known keystream bit will give us 2 equations since each keystream bit has been preceded by a one in the LFSR-sequence. Hence, we have $2m$ equations. However, since the sequence produced by the LFSR has been irregularly decimated, we have no knowledge about the number of bits discarded between the known keystream bits. Instead, we only know that the observed keystream sequence (4.5) corresponds to the LFSR-sequence

$$1, z_t, X_0, 1, z_{t+1}, X_1, 1, z_{t+2}, X_2, \ldots, X_{m-2}, 1, z_{t+m-1}, \tag{4.6}$$

where each X_i corresponds to a sequence of zero or more 2-tuples $\in \{00, 01\}$. For each of these 2-tuples that we guess correct we will get one more equation since the first bit is always equal to zero. The total number of equations available is thus $2m + k$ where k is the total number of 2-tuples discarded. To get a complete system of equations in the initial state bits we require that

$$2m + k \geq L. \tag{4.7}$$

Using as few keystream bits as possible allows us to write

$$m = \left\lceil \frac{L-k}{2} \right\rceil. \tag{4.8}$$

The probability that there are no 2-tuples discarded before producing a new keystream bit is $1/2$. The probability that exactly one 2-tuple is discarded is $1/4$, etc. The probability that exactly i 2-tuples are discarded is

$$\Pr(\text{exactly } i \text{ discarded}) = \frac{1}{2^{i+1}}. \tag{4.9}$$

LFSR length	Complexity	k_{max}
128	$2^{83.3}$	46
256	$2^{168.8}$	90
512	$2^{340.3}$	178
1024	$2^{682.7}$	352

Table 4.1: The complexity of the attack for some LFSR lengths.

In the algorithm we try to guess the number of discarded bits, by guessing the most probable combinations first. There are $m-1$ gaps to guess, $X_0, X_1, \ldots, X_{m-2}$. The probability that no bits are discarded in any of the gaps is 2^{-m+1}. The probability that one 2-tuple is discarded in a given gap is $2^{-m+1} \cdot 2^{-1} = 2^{-m}$ etc. The probability that in total k 2-tuples are discarded is, for each possible assumption,

$$2^{-m-k+1}. \tag{4.10}$$

The number of ways to discard k 2-tuples in a total of $m-1$ gaps is a well-known combinatorial problem and given by $\binom{m-2+k}{k}$. We start by testing the case when 0 bits are discarded, then the case when 1 bit has been discarded, etc. The probability that we guess correct within

$$\sum_{k=0}^{k_{max}} \binom{m-2+k}{k} = \sum_{k=0}^{k_{max}} \binom{\lceil \frac{L+k}{2} \rceil - 2}{k} \tag{4.11}$$

guesses is

$$\sum_{k=0}^{k_{max}} \binom{m-2+k}{k} 2^{-m-k+1} = \sum_{k=0}^{k_{max}} \binom{\lceil \frac{L+k}{2} \rceil - 2}{k} 2^{-\lceil \frac{L+k}{2} \rceil + 1}. \tag{4.12}$$

Note that, using this algorithm, not the full probability space can be searched. As k_{max} approaches L, (4.12) goes towards $2/3$. We choose k_{max} such that the probability of success is > 0.5 and calculate the complexity of the attack for some different LFSR lengths. The result can be found in Table 4.1. For these cases we get a complexity of approximately $O(2^{0.66L})$ which is the same complexity as the BDD-based attack gives. The advantage of our algorithm is that there is hardly any memory usage. The amount of memory needed is limited to the memory required to solve the system of linear equations i.e., $O(L^2)$. The attack can be easily parallelized on several computers since solving the system of equations for one guess is independent of another guess. The estimated number of keystream bits required in the attack

is L. At most $L/2$ bits are needed to find L equations in the initial state but in order to verify that a candidate initial state is correct we need to compare the candidate keystream with the real keystream. This would require approximately L keystream bits in total since the entropy of the initial state is assumed to be L bits.

4.4 New Attack Using Long Keystream

In this section we will give a detailed description of an attack that requires the adversary to have a keystream sequence with length exponential in the size of the LFSR. We show that the asymptotic computational complexity needed to find the initial state is significantly lower than in the attack by Mihaljević when the same amount of keystream is known.

4.4.1 Main Ideas

The idea behind our attack is somewhat similar to Mihaljević's attack. However, this attack introduces the following new ideas.

- Instead of exhaustively searching the positions s_{2t+1} when the positions s_{2t} is zero, we choose to increase the size of the LFSR segment so that we still have adequate information to recover the key.

- Instead of considering only one segment, the attack is extended to a more general case. Several short keystream segments can be considered simultaneously.

These new ideas will result in a much more efficient attack.

4.4.2 Method for Cryptanalysis

We start by explaining the attack using a single segment. In the next section we generalize the attack by considering several segments simultaneously.

The attack considers a keystream segment of size l. Note that the *even* bits correspond to the s_{2t} values determining whether to output s_{2t+1} or not. We guess that this segment of size l is produced by an LFSR sequence of length S. This is the same as saying that l out of the $S/2$ even bits are one. This guess will be correct with probability

$$2^{-S/2}\binom{S/2}{l}, \tag{4.13}$$

since there are $\binom{S/2}{l}$ ways of choosing ones in the even positions. The number of segments that have to be tested before one that gives the correct guess is found, is geometrically distributed with an expected value of $2^{S/2}\binom{S/2}{l}^{-1}$.

For the segment, the number of equations in state bits that we get is $S/2 + l$ since the $S/2$ even bits are guessed and the l keystream bits will be located in the positions s_{2t+1}, when $s_{2t} = 1$. Since L equations are needed, the equality

$$S = 2(L - l) \tag{4.14}$$

must be satisfied.

The attack only makes sense for $l \leq S/2$, so l must be chosen such that $0 \leq l \leq L/2$. The amount of keystream, denoted N, needed in the attack is l times the number of non-overlapping keystream segments that we expect to try before a solution is found. This is given by

$$N = 2^{L-l}\binom{L-l}{l}^{-1} l. \tag{4.15}$$

For each keystream segment, $\binom{S/2}{l} = \binom{L-l}{l}$ combinations need to be tested, noting that (4.14) implies $S/2 = L - l$. Additionally, for each combination a system of linear equations needs to be solved. This system of equations can be solved in time $O(L^\omega)$. In theory $\omega \leq 2.376$, see [CW90], but the constant factor in this algorithm is expected to be very large. The fastest practical algorithm is Strassen's algorithm [Str69], which requires about $O(L^{\log_2 7}) \approx L^{2.8}$ operations. Often, for simplicity, the value $O(L^3)$ is used for the complexity of solving a system of L linear equations in L variables. Thus, the total computational complexity, denoted C, of the attack is

$$C = L^3 2^{L-l}. \tag{4.16}$$

For each candidate state that we recover, we need to clock the self-shrinking generator and produce approximately L candidate keystream bits and compare to real keystream. If the candidate keystream matches the real keystream we have found the correct state.

We will mainly focus on the asymptotic complexity, i.e., the complexity when L is large and only the factors exponential in L are considered. This case is thoroughly examined in the next subsection. However, before we move on to the asymptotic case we take a closer look at the non-asymptotic case. In the attack by Mihaljević the positions s_{2t+1} when the positions s_{2t} are zero are exhaustively searched. This has the advantage that a system of linear equations does not have to be solved. All recovered bits are consecutive. In our attack the recovered bits are not consecutive and when L is small the complexity of solving a system of equations will have noticeable impact on the computational complexity.

It is not possible to give an explicit value on L which determines when our attack becomes better than Mihaljević's attack since it will also depend on the amount of known keystream. As an example we can consider the case when $L = 512$, using the complexity $L^{2.8}$ for solving the system of linear equations. If the amount of known keystream is less than approximately $2^{56} = 2^{0.11L}$ our attack will give a better computational complexity, whereas if the amount of known keystream exceeds this bound, the attack by Mihaljević will give a better complexity. Increasing L will result in the behaviour that our attack will achieve better complexity for a wider interval of known keystream.

4.4.3 Asymptotic Complexity

Now we derive an asymptotic expression for the number of keystream bits needed and the complexity of the attack. Several previous attacks on the self-shrinking generator ignore the polynomial terms in the complexity and concentrate only on the exponential terms since these are dominant as the length of the LFSR increases. Taking only the exponential terms we can write the amount of keystream N needed and the computational complexity C as

$$N = 2^{L-l}\binom{L-l}{l}^{-1}, \tag{4.17}$$

$$C = 2^{L-l}. \tag{4.18}$$

Since l must be chosen such that $0 \leq l \leq L/2$, we write $l = \beta L/2$, $0 \leq \beta \leq 1$. Then the expressions for the keystream and complexity can be rewritten as

$$N = 2^{(1-\beta/2)L}\binom{L(1-\beta/2)}{L(\beta/2)}^{-1}, \tag{4.19}$$

$$C = 2^{(1-\beta/2)L}. \tag{4.20}$$

The following theorem is a standard bound for the binomial coefficient.

Theorem 4.1: For $0 < \lambda < 1$ and $\mu = 1 - \lambda$, we have

$$\frac{1}{\sqrt{8n\lambda\mu}}2^{nh(\lambda)} < \binom{n}{\lambda n} < \frac{1}{\sqrt{2\pi n\lambda\mu}}2^{nh(\lambda)}, \tag{4.21}$$

where $h(\lambda)$ is the binary entropy function, i.e.,

$$h(\lambda) = -\lambda \log_2 \lambda \; - \; (1-\lambda)\log_2(1-\lambda). \tag{4.22}$$

For a proof we refer to [Rom92]. For large n we can write $\binom{n}{\lambda n} \approx 2^{nh(\lambda)}$. Theorem 4.1 allows us to write the asymptotic keystream length as

$$N = 2^{\left(1-\beta/2-(1-\beta/2)h\left(\frac{\beta}{2-\beta}\right)\right)L}. \tag{4.23}$$

Similar asymptotic expressions can be found for the attack proposed by Mihaljević. They are

$$N_M = 2^{L/2}\binom{L/2}{l}^{-1} = 2^{(1/2-h(\beta)/2)L}, \tag{4.24}$$

$$C_M = 2^{L-l} = 2^{(1-\beta/2)L}. \tag{4.25}$$

In this case, $L/4 \leq l \leq L/2$, so we write again $l = \beta L/2$. Hence, β must be chosen such that $1/2 \leq \beta \leq 1$. In Fig. 4.3 the complexity and the amount of keystream required is plotted for both attacks by varying the value of β.

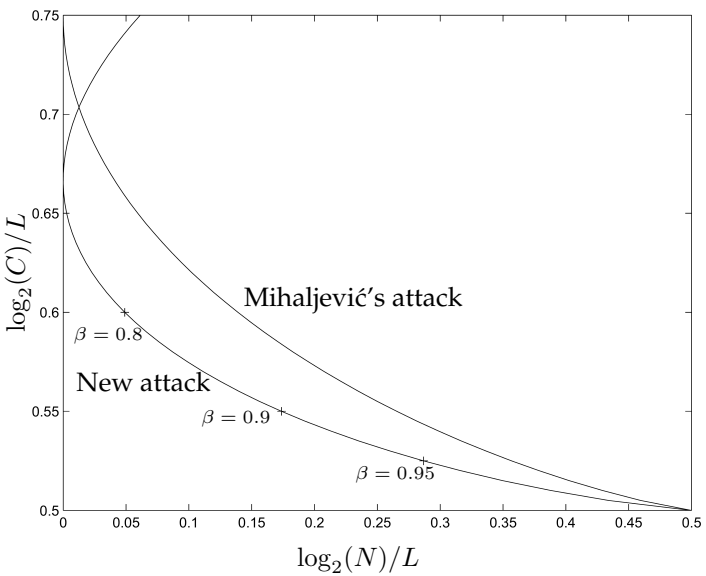

Figure 4.3: The attack complexity as a function of the keystream length. Comparison between the new attack and the attack by Mihaljević.

It is clear that the new attack has better asymptotic complexity. Though, as the amount of known keystream approaches $2^{0.5L}$ both attack complexities converge to $2^{0.5L}$.

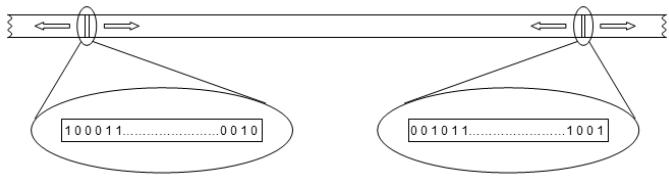

Figure 4.4: We consider several segments, each of l bits.

4.5 Improving the Attack

In the previous section only one block was considered. It is possible to extend the attack, looking at several sections at a time, to achieve better complexity for specific values of known keystream. In this section we examine the asymptotic behaviour of the complexity when the number of blocks is more than one.

In Section 4.1, we showed that the expected number of clockings $E\{\mu\}$ and the variance of the number of clockings $V\{\mu\}$ to generate a keystream bit are 4 and 8, respectively.

Consider B blocks, each block consisting of l keystream bits, see Fig. 4.4. Guess that the keystream bits in each block stem from an LFSR subsequence of length S. As before, this is the same as saying that l out of the $S/2$ even positions are ones. This guess will be correct for all segments simultaneously with probability

$$\left(2^{-S/2}\binom{S/2}{l}\right)^B = 2^{-BS/2}\binom{S/2}{l}^B. \tag{4.26}$$

The expected number of segment combinations that have to be tried before a combination is found that gives the correct guess, is

$$2^{BS/2}\binom{S/2}{l}^{-B}. \tag{4.27}$$

For each segment, we get $S/2+l$ equations so $L = BS/2 + Bl$, implying that $S = 2(L/B - l)$. Since $l \leq S/2$, l must be chosen such that $0 \leq l \leq L/2B$. Now we make the following approximations:

- In the observed keystream $\mathbf{z}$ of length N, for large L, there are approximately N^B ways of selecting segment combinations for the B blocks.
- The distance between two blocks will be approximately N.

Using the first approximation, the number of keystream bits needed to get $2^{BS/2}\binom{S/2}{l}^{-B}$ segment combinations, each of l bits, is

$$N = 2^{L/B-l}\binom{L/B-l}{l}^{-1}. \tag{4.28}$$

The problem with using several keystream segments is that we do not know the number of clocks between the blocks since only keystream bits are available. Consequently, this number has to be guessed. In order to guess the number of clocks we use the central limit theorem, which states that the sum of random variables is approximately normally distributed. More specifically, and applied to our case, the number of clocks μ_N needed to produce N keystream bits is normally distributed with expected value, $E\{\mu_N\}$, and standard deviation, σ_{μ_N},

$$E\{\mu_N\} = E\{\mu\} \cdot N = 4 \cdot 2^{L/B-l}\binom{L/B-l}{l}^{-1}, \tag{4.29}$$

$$\sigma_{\mu_N} = \sigma_\mu \cdot \sqrt{N} = \sqrt{8} \cdot 2^{L/2B-l/2}\binom{L/B-l}{l}^{-\frac{1}{2}}. \tag{4.30}$$

Assuming that the correct number of clockings will be within one standard deviation, we have to check $\binom{S/2}{l}^{B}\sigma_{\mu_N}^{B-1}$ different systems of equations for each of the $2^{BS/2}\binom{S/2}{l}^{-B}$ combinations we expect that we need. Hence, the total complexity of the attack will be

$$C = L^3 2^{L-Bl} 2^{\frac{B-1}{2}(L/B-l)}\binom{L/B-l}{l}^{-\frac{B-1}{2}}\sqrt{8}^{B-1}. \tag{4.31}$$

4.5.1 Asymptotic Complexity

In order to get an asymptotic expression for (4.31), only the terms exponential in L are considered. Moreover, since $l \leq L/2B$, we can write

$$l = \frac{\beta L}{2B} \tag{4.32}$$

and let $0 \leq \beta \leq 1$. Then we can write the amount of keystream needed and the complexity as

$$N = 2^{\left(\frac{1}{B}-\frac{\beta}{2B}\right)\left(1-h\left(\frac{\beta}{2-\beta}\right)\right)L}, \tag{4.33}$$

$$C = 2^{\left(1-\frac{\beta}{2}+\frac{B-1}{2}\left(\frac{1}{B}-\frac{\beta}{2B}\right)\left(1-h\left(\frac{\beta}{2-\beta}\right)\right)\right)L}. \tag{4.34}$$

C	N	M	P
$2^{0.50L}$	$2^{0.50L}$	—	—
$2^{0.52L}$	$2^{0.32L}$	—	—
$2^{0.54L}$	$2^{0.21L}$	$2^{0.52L}$	$2^{0.79L}$
$2^{0.56L}$	$2^{0.14L}$	$2^{0.58L}$	$2^{0.86L}$
$2^{0.58L}$	$2^{0.09L}$	$2^{0.62L}$	$2^{0.91L}$
$2^{0.60L}$	$2^{0.05L}$	$2^{0.65L}$	$2^{0.95L}$

Table 4.2: Our attack compared to a generic time-memory-data tradeoff attack.

These can be viewed as more general expressions for the keystream length and the computational complexity in the proposed attack. Indeed, letting $B = 1$ will result in (4.20) and (4.23). The complexity and the amount of keystream needed for different number of segments can be seen in Fig. 4.5. We see that for some small intervals of known keystream it is favourable to consider more that one segment.

4.5.2 Comparison to Time-Memory-Data Tradeoff Attacks

We have shown that our new attack compares favourably to the attack proposed by Mihaljević. Now we will compare our attack to the generic time-memory-data tradeoff attack on stream ciphers as given in section 2.3.3. Using the notation in this chapter, we have the tradeoff curve

$$CM^2N^2 = L^2, \quad \text{for any } N^2 \leq C \leq L, \tag{4.35}$$

where C is the time complexity in the real time phase, M the amount of memory needed, N the amount of keystream needed and L is the search space (the length of the shift register). The preprocessing time in the attack is given by $P = L/N$. To show that our attack is significantly better than this time-memory-data tradeoff attack we pick a few points on our curve and give the amount of memory and preprocessing time needed in order to have a better attack. This is given in Table 4.2. Also note that the first two points given in the table do not satisfy the condition $N^2 \leq C \leq L$.

From the table it is clear that using the same time complexity and amount of keystream the generic time-memory-data tradeoff attack requires an infeasible amount of memory and precomputation. A typical point on the curve, mentioned in [BS00], is $P = C = 2^{0.66L}$ and $M = N = 2^{0.33L}$. This point will give more realistic values, and comparing it to our attack we see

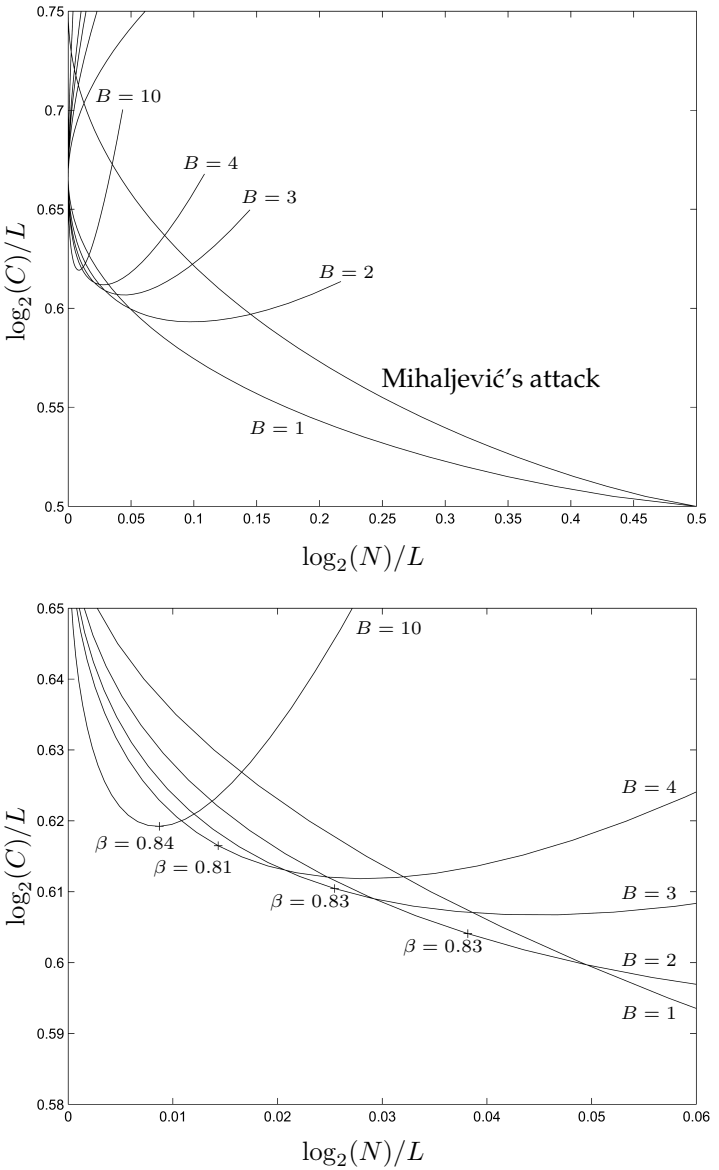

Figure 4.5: The attack complexity as a function of the keystream length. Comparison between the new attack and the attack by Mihaljević.

that it uses both more data and more computation than a typical point on our curve.

4.6 Summary

Since the introduction of the self-shrinking generator in 1994 several attacks have been proposed, some requiring only a small known keystream while others need longer sequence to succeed. In this chapter we presented two new attacks on the self-shrinking generator, one using a short keystream and one requiring a longer keystream. In the first attack, operating on a very short known keystream, we showed that the complexity is approximately the same as the best previously known attack (the BDD-based attack). However, our attack needs almost no memory whereas the BDD-based attack is unpractical due to the large memory required. In the second attack we assumed a longer known keystream. It was shown that the asymptotic computational complexity for this attack is significantly lower than in the previously best attack, for any amount of known keystream of length $2^{\alpha L}$ when $0 < \alpha < 0.5$.

5

Some Attacks on the Bit-Search Generator

In the previous chapter we considered the self-shrinking generator, an algorithm that decimated a pseudo random sequence in an irregular way. The bit-search generator (BSG) is a keystream generator intended to be used as a stream cipher, or perhaps as a part of a stream cipher. It was introduced in 2004 by Gouget and Sibert [GS04] and the idea behind the construction is similar to the self-shrinking generator. The output of the BSG is produced by a simple algorithm, taking a pseudo random sequence as input.

In this chapter we investigate some possible attacks on the bit-search generator. Similar to the self-shrinking generator, we will assume that the pseudo random sequence is an m-sequence, generated by an LFSR with primitive feedback polynomial and that the feedback polynomial is known to the attacker.

This chapter is based on [HJ05] and is outlined as follows. In Section 5.1 we describe the BSG and we compare the construction with similar generators. Then, in Section 5.2 we present an attack that reconstructs the input sequence to the BSG algorithm. By doing this we can recover the initial state of the LFSR. Section 5.3 gives the framework for a possible distinguishing attack and in Section 5.4 we discuss some related work and proposed improvements of the generator. In Section 5.5 we give our conclusions.

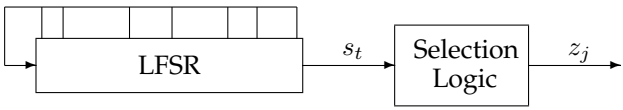

Figure 5.1: Block model of the bit-search generator.

5.1 Description of the Bit-Search Generator

In this section we describe the bit-search generator in two different but equivalent ways. First we give the original description that uses a sequence $\mathbf{s}$ as input, as presented in [GS04]. Then we give an alternative description that uses the differential sequence $\mathbf{d}$ of $\mathbf{s}$ as input. We also compare the construction to similar keystream generators.

The principle of the BSG is very simple. It consists only of an LFSR and some small selection logic, see Fig. 5.1. Consider a binary sequence $\mathbf{s} = (s_0, s_1, s_2 \ldots)$ generated by the LFSR. The output sequence, or keystream, $\mathbf{z} = (z_0, z_1, z_2 \ldots)$ is constructed from $\mathbf{s}$ by first letting $b = s_0$ be the first bit to search for. If the search ends immediately, i.e. $s_1 = b = s_0$ we output 0, otherwise we continue to search the sequence $\mathbf{s}$ until the bit we search for is found. When the correct bit is found we output 1 and we let the following bit be the next to search for. An output bit is produced after 2 input bits with probability $1/2$, after 3 input bits with probability $1/4$ etc. In general, an output bit is produced after $i + 1$ input bits with probability 2^{-i} so the average number of input bits needed to produce one output bit is

$$\sum_{i=0}^{\infty}(i+1)\cdot 2^{-i} = 3. \tag{5.1}$$

This shows that the rate of the BSG is asymptotically $1/3$.

To motivate why this generator is interesting we compare it to some other well-known generators based on the idea of only using LFSRs and some selection logic. We base the comparison on the number of LFSRs used and the rate of the cipher. As we can see in Table 5.1 the BSG has lower rate than the alternating step generator [Gün88] and the shrinking generator but it uses only one LFSR. The self-shrinking generator has also only one LFSR but it has lower rate.

We now consider the differential sequence $\mathbf{d}$ of $\mathbf{s}$. The differential sequence $\mathbf{d} = (d_0, d_1, d_2 \ldots)$ is defined as

$$d_t = s_t \oplus s_{t+1}. \tag{5.2}$$

If the sequence $\mathbf{s}$ is generated by an LFSR it is well known, see e.g. [McE87],

Generator	Number of LFSRs needed	Rate
Alternating Step	3	1
Shrinking	2	1/2
Self-Shrinking	1	1/4
BSG	**1**	**1/3**

Table 5.1: Comparison between the BSG and some well-known generators.

that the differential sequence can be generated by the same LFSR. The two sequences differ only by some shift. When reconstructing $\mathbf{s}$ from $\mathbf{d}$ we need to guess the first bit in $\mathbf{s}$, then the remaining bits are uniquely determined from $\mathbf{d}$.

The output of the BSG can be uniquely described by knowledge of the differential sequence. Hence, if we can reconstruct the differential sequence we can predict the future outputs uniquely and we can also recover the key used to initialize the LFSR. The BSG operates on the differential sequence in the following way. If $d_t = 1$ we know that $s_t \neq s_{t+1}$ so we will output 1. Then we search the sequence $\mathbf{d}$ until we find the next $d_j = 1$. If instead $d_t = 0$ we know that we have two consecutive bits which are the same, hence we output 0. In both cases we now know that we have found the bit we search for in the original BSG and we skip the next bit since it does not matter which value it has. It is clear that the output of the BSG can be generated from either the original LFSR sequence or from the differential sequence. The following is an example of a sequence $\mathbf{s}$ and the corresponding differential sequence $\mathbf{d}$. Applying the algorithms, we can see that they produce the same output.

$$\begin{aligned} s = &\quad 0101001001110111010110 10\ldots \quad \Rightarrow z = 110010101\ldots \\ d = &\quad 1111011010011001111011 1\ldots \quad \Rightarrow z = 110010101\ldots \end{aligned} \tag{5.3}$$

A summary of the two algorithms can be found in Fig. 5.2.

5.2 Reconstructing the Input Sequence

In this section we will describe a known plaintext attack that tries to reconstruct the differential sequence from the output sequence. In our attack we assume that we have an LFSR generating the pseudo random sequence and that the feedback polynomial of the LFSR is known to the attacker. If we have an LFSR of length L we need to guess L bits to be able to find a candidate initial state of the LFSR. Each bit can be written as a linear function of

Output generated from **s**

```
i = 0; j = 0;
while (1)
    b = s[i];
    i++;
    if (s[i] == b)  z[j] = 0;
    else  z[j] = 1;
    while  (s[i] ! = b)  i++;
    i++; j++;
```

Output generated from **d**

```
i = 0; j = 0;
while (1)
    z[j] = d[i];
    if (d[i] == 1)
        i++;
        while (d[i] == 0) i++;
    i += 2;
    j++;
```

Figure 5.2: The original BSG algorithm and an equivalent algorithm using the differential sequence **d** of **s** as input.

the initial state bits and by clocking the LFSR with a candidate initial state we can see if the candidate output equals the given output.

The attack in this section is very similar to the short keystream attack on the self-shrinking generator given in Chapter 4. It follows from the algorithm given in Fig. 5.2 that $z_t = 0$ corresponds to a 0 followed by an unknown value in the differential sequence. It is also clear that $z_t = 1$ corresponds to a 1 followed by $j \geq 0$ zeros followed by a single 1 and an unknown value. In short,

$$\begin{array}{lcl} z_t = 0 & \Rightarrow & (0, -) \\ z_t = 1 & \Rightarrow & (1, 0^j, 1, -) \end{array} \tag{5.4}$$

The probability of having j zeros is 2^{-j-1}, i.e. $z_t = 1$ corresponds to (1,1,–) with probability $1/2$, (1,0,1,–) with probability $1/4$, (1,0,0,1,–) with probability $1/8$ etc. The expected number of inserted zeros is

$$\sum_{i=0}^{\infty} i \cdot 2^{-i-1} = 1. \tag{5.5}$$

In the following we will denote by a the number of ones that we observe in an keystream sequence, b is the number of zeros in the keystream sequence and k is the number of zeros that are inserted in the candidate differential sequence, stemming from a set of a ones in the keystream sequence.

Now, assume that we have a set of a ones. There is one way to insert a total of $k = 0$ zeros and this happens with probability 2^{-a}. The number of ways to insert a total of $k = 1$ zero is $\binom{a}{1}$ and each has a probability of $2^{-a+1} \cdot 2^{-2} = 2^{-a-1}$. The number of ways to insert k zeros into a set

Search algorithm

Pick a part $\mathbf{z}'$ of $\mathbf{z}$ s.t. 2a+b=L;
k=0;
while (k <= k_{max})
    Try all ways to insert k zeros in $\mathbf{z}'$;
    Delete last bit in $\mathbf{z}'$;
    if (Deleted bit == 0) k = k + 1;
    else k = k + 2;

Figure 5.3: The algorithm used to find the correct differential sequence.

consisting of a ones is given by $\binom{a-1+k}{k}$. Hence, the probability of having a total of k zeros inserted will be

$$\binom{a-1+k}{k} 2^{-a-k}. \tag{5.6}$$

We construct a simple search algorithm based on these observations. The easiest way to find the correct differential sequence is to just guess the number of inserted zeros.

When we try to insert k zeros we need to look at an output sequence that satisfies $2a + b + k = L$. This is clear since every one in the output will give us two known bits and every zero will give us one known bit in the differential sequence. If we insert k extra zeros we will have a total of L bits which is enough to find the initial state. This leads us immediately to the algorithm in Fig. 5.3.

We start by just picking a part $\mathbf{z}'$ of the output sequence such that the bits of $\mathbf{z}'$ satisfies $2a + b = L$. Then we insert $k = 0$ zeros. If this candidate is not the correct differential sequence, we delete the last bit in $\mathbf{z}'$. If a zero is deleted we try $k = 1$ next time since $b \leftarrow b - 1$ and we still require $2a + b + k = L$ to hold. For the same reason, if a one is deleted we try $k = 2$ next time. Every time $k \leftarrow k + 2$ we will miss some possible combinations and, hence, not the full space will be searched.

5.2.1 Analysis of the Algorithm

The complexity of the algorithm and the probability of success will depend on two factors.

- The ratio between the number of zeros and the number of ones in the sequence. If we have found a $\mathbf{z}'$ which has many more zeros than ones, the complexity will be lower. This will also give us a higher success probability since we will delete a 0 more often than we will delete a 1.
- The maximum number of zeros we will try to insert into the sequence before we give up. This is the value k_{max} in the algorithm in Fig. 5.3. Choosing a high value for k_{max} will increase the success probability but it will also increase the complexity.

We consider the case when we choose a sequence $\mathbf{z}'$ at random. We expect the number of zeros in the sequence to be equal to the number of ones. We also expect that the deleted bit is 1 every second time. Moreover, when $\mathbf{z}'$ is of odd length, we consider the pessimistic case when $a = b+1$. We have the following equations

$$\left.\begin{array}{rcl} 2a+b+k & = & L \\ a & = & b \end{array}\right\} \Rightarrow a = \left\lceil \frac{L-k}{3} \right\rceil \tag{5.7}$$

The probability of success will be

$$\sum_{k=0}^{k_{max}} \binom{\lceil \frac{L-k}{3} \rceil - 1 + k}{k} 2^{-\lceil \frac{L-k}{3} \rceil - k} \tag{5.8}$$

and we have a total complexity of

$$\sum_{k=0}^{k_{max}} \binom{\lceil \frac{L-k}{3} \rceil - 1 + k}{k}. \tag{5.9}$$

Similar equations can easily be found also if $a \neq b$. We choose k_{max} as the smallest integer such that the probability of success is > 0.5. Focusing on the expected case when $a = b$, we summarize the complexity of an attack in Table 5.2 with respect to the length of the LFSR (keylength). It is clear that the complexity of the attack is very close to $2^{0.5L}$ tests for all cases.

Assuming that the entropy of the internal state is L bits, we can be confident that a candidate initial state is the correct state after comparing the produced sequence with about L keystream bits. Thus the amount of keystream needed in the attack is $O(L)$.

5.2.2 A Data-Time Tradeoff

As mentioned in the previous section, it is clear that the complexity of the attack depends on the number of ones that we observe in the keystream.

Keylength	k_{max}	Complexity
64	19	$2^{31.74}$
96	27	$2^{47.50}$
128	36	$2^{63.96}$
160	44	$2^{79.82}$
192	52	$2^{95.71}$
224	61	$2^{112.37}$
256	69	$2^{128.29}$

Table 5.2: The attack complexity when the number of zeros equals the number of ones in $\mathbf{z}'$.

With a large amount of keystream we can find sequences with few ones and, hence, the attack complexity is decreased. This provides a data-time tradeoff in the attack. Assume that we want to find a part $\mathbf{z}'$ of $\mathbf{z}$ that contains at most a ones and at least b zeros, where $b > a$. Looking at a random sequence of $a + b$ bits, the probability that we find a sequence with at most a ones is given by the binomial distribution,

$$\Pr(\#\text{ones} \leq a) = \frac{\sum_{i=0}^{a} \binom{a+b}{i}}{2^{a+b}}, \tag{5.10}$$

using the approximation that sequences are independent. The number of tries needed before a desired sequence is found is geometrically distributed with an expected value of

$$\frac{2^{a+b}}{\sum_{i=0}^{a} \binom{a+b}{i}} = \frac{2^{L-a}}{\sum_{i=0}^{a} \binom{L-a}{i}}. \tag{5.11}$$

In the equality we use $2a + b = L$. Table 5.3 demonstrates this data/time tradeoff for the case when $L = 128$, i.e., the keylength is 128 bits. Simulations show that the time complexity and the amount of keystream needed intersect at around $2^{0.27L}$ for all L between 64 and 1024 bits.

The complexities in Table 5.2 and Table 5.3 are given as the number of tests. To test if a candidate sequence is correct, a constant time is needed. This time can be divided into two parts. First we need to find the initial state of the LFSR by solving a system of L unknowns and L equations. For simplicity we write the time complexity for this step as L^3. We also need to clock the LFSR a sufficient number of times to compare our candidate output sequence with the observed output sequence. This second constant

Number of zeros (b) and ones (a) in $\mathbf{z}'$	k_{max}	Complexity	Keystream
b = 2a	29	$2^{51.09}$	$2^{10.46}$
b = 3a	24	$2^{42.21}$	$2^{21.59}$
b = 4a	21	$2^{36.31}$	$2^{31.32}$
b = 5a	19	$2^{32.14}$	$2^{39.75}$
b = 6a	17	$2^{28.46}$	$2^{48.70}$

Table 5.3: The data/time tradeoff based on the number of ones and zeros in $\mathbf{z}'$ using a 128 bit key.

would also be needed in an exhaustive key search. Thus, the total time complexity for our key recovery attack is $O(L^3 2^{0.5L})$ knowing $O(L)$ keystream bits. With the data/time tradeoff the complexity of the attack is $O(L^3 2^{0.27L})$ if we know $O(2^{0.27L})$ bits of the keystream. Note that these complexities are not formally derived but simulations show that they are valid (at least) up to LFSR lengths of 1024 bits. The memory complexity of the attack is limited to the memory needed to solve the system of linear equations, $O(L^2)$.

5.3 Distinguishing Attack

In this section we describe a possible distinguishing attack on the BSG. In the attack we assume that we have found a multiple of the feedback polynomial that is of weight w and degree M. Any multiple of a feedback polynomial will produce the same output sequence as the original polynomial. Recalling (3.24) we know that that the expected degree M when polynomial multiples of weight w start to appear is

$$M = (w-1)!^{1/(w-1)} 2^{L/(w-1)}, \tag{5.12}$$

where L is the degree of the polynomial. Hence, a feedback polynomial of degree L is expected to have a multiple of weight w that is of degree approximately $M = 2^{\frac{L}{w-1}}$. Now, assume that we have found a multiple of weight w that is of degree M.

The linear recurrence of the LFSR can be written as

$$0 = d_t \oplus d_{t+\tau_1} \oplus d_{t+\tau_2} \oplus \ldots \oplus d_{t+\tau_{w-1}}, \tag{5.13}$$

where $\tau_{w-1} = M$ and $\tau_j < \tau_k,\ j < k$. A zero in the output sequence $\mathbf{z}$ corresponds to a zero in the differential sequence and a one in the output

corresponds to a one in the differential sequence. Since the BSG has rate $1/3$ we can consider the following sums of symbols from the output sequence

$$B_t = z_t \oplus z_{t+\frac{\tau_1}{3}} \oplus z_{t+\frac{\tau_2}{3}} \oplus \ldots \oplus z_{t+\frac{\tau_{w-1}}{3}}. \tag{5.14}$$

We know that $B_t = 0$ if we have the correct synchronization ($d_{t+\tau_1}$ appears as $z_{t+\frac{\tau_1}{3}}$, $d_{t+\tau_2}$ appears as $z_{t+\frac{\tau_2}{3}}$ etc.) in the positions. We give an approximate value of the probability that we have synchronization in one position. With a multiple of low weight and high degree M the distance between z_t and any $z_{t+\frac{\tau_j}{3}}$ is in the order of M. Using the central limit theorem we say that the total number of inserted zeros after M outputs is normally distributed with standard deviation $\sigma \cdot \sqrt{M}$, where σ is the standard deviation for the number of inserted zeros after one output. Now, we approximate the probability that we have the correct synchronization as $M^{-\frac{1}{2}}$.

Hence, the probability that $z_{t+\frac{\tau_j}{3}}$, $1 \leq j \leq w-1$ is synchronized with z_t is approximately $M^{-\frac{1}{2}}$. The probability that all $w-1$ positions are synchronized, denoted $\Pr(\text{sync})$, is

$$\Pr(\text{sync}) = (M^{-\frac{1}{2}})^{w-1} = M^{-\frac{w-1}{2}} \tag{5.15}$$

and the probability that $B_t = 0$ can be calculated as

$$\begin{aligned} \Pr(B_t = 0) &= \Pr(B_t = 0 \mid \text{sync}) \cdot \Pr(\text{sync}) \\ &\quad + \Pr(B_t = 0 \mid \text{no sync}) \cdot \Pr(\text{no sync}) \\ &= 1 \cdot M^{-\frac{w-1}{2}} + 1/2 \cdot (1 - M^{-\frac{w-1}{2}}) \\ &= 1/2 + 1/2 \cdot M^{-\frac{w-1}{2}}. \end{aligned} \tag{5.16}$$

With a bias of $M^{-\frac{w-1}{2}}$ we will need, according to (2.62), about $N = M^{w-1}$ samples of the output sequence to distinguish it from random. The complexity of the distinguishing attack depends on the degree of the multiple and if the degree is the expected degree, $M = 2^{\frac{L}{w-1}}$, our distinguisher needs about $N = 2^L$ samples. However, if the feedback polynomial is not carefully chosen and we instead can find a multiple of low weight that is of much lower degree than expected, then the attack can be very efficient.

We can consider a feedback polynomial with $M \ll 2^{\frac{L}{w-1}}$ as being a weak polynomial and the BSG using a weak polynomial can be efficiently attacked.

The values in the previous attack are approximated but for large M they are quite accurate. In the case where the feedback polynomial itself is of low weight, the values are not very accurate. We now describe how this attack can be mounted if the LFSR uses a feedback polynomial of some low

weight w. Equation (5.13) will always hold for the differential sequence. To find the optimum guess for $z_{t+\frac{\tau_j}{3}}$, $1 \leq j \leq w-1$ in (5.14) we use the generating function for the probability of the number of clockings after λ outputs. Recall that the BSG will produce a keystream bit after two clockings with probability $1/2$, after 3 clockings with probability $1/4$ etc. The generating function can be written as

$$\left(\sum_{n=1}^{\infty} \frac{1}{2^n} z^{n+1} \right)^{\lambda}. \tag{5.17}$$

The coefficient of z^n is the probability that the LFSR has been clocked n times when the BSG has generated λ keystream bits.

By choosing the λ_j for which the coefficient of z^{τ_j} is highest we can determine which guess will give us the best probability of synchronization and we will also get the exact probability of a correct guess. We denote the probability that we guess λ_j correctly by p_{λ_j}. If p_{λ_j}, $1 \leq j \leq w-1$ are independent the probability that $B_t = 0$ can be written, similarly to (5.16), as

$$\begin{aligned} \Pr(B_t = 0) &= 1 \cdot \prod_{j=1}^{w-1} p_{\lambda_j} + 1/2 \cdot (1 - \prod_{j=1}^{w-1} p_{\lambda_j}) \\ &= 1/2 + 1/2 \cdot \prod_{j=1}^{w-1} p_{\lambda_j}. \end{aligned} \tag{5.18}$$

With a bias of $\prod_{j=1}^{w-1} p_{\lambda_j}$ we need about

$$N = \frac{1}{\prod_{j=1}^{w-1} p_{\lambda_j}^2} \tag{5.19}$$

samples for a successful distinguishing attack. We end this section with a small numerical example showing the performance of this distinguisher on a low weight feedback polynomial.

EXAMPLE *5.1:* Consider the weight 5 primitive feedback polynomial $1 + x^{29} + x^{66} + x^{95} + x^{128}$. Write the linear recurrence in the differential sequence as

$$0 = d_t \oplus d_{t+29} \oplus d_{t+66} \oplus d_{t+95} \oplus d_{t+128}. \tag{5.20}$$

Using (5.17) we find that the highest coefficient for z^{29}, z^{66}, z^{95} and z^{128} is achieved when we have $\lambda_1 = 10$, $\lambda_2 = 22$, $\lambda_3 = 32$ and $\lambda_4 = 43$ respectively. The best possible approximation of (5.14) is then $B_t = z_t + z_{t+10} + z_{t+22} +$

$z_{t+32} + z_{t+43}$. The probability that each of these terms are synchronized with z_t is the coefficient for each term in (5.17), i.e.,

$$p_{\lambda_1} = 2^{-3.43}, \quad p_{\lambda_2} = 2^{-4.06}, \quad p_{\lambda_3} = 2^{-4.31}, \quad p_{\lambda_4} = 2^{-4.53}. \tag{5.21}$$

This gives us a total bias of $\prod_{j=1}^{w-1} p_{\lambda_j} = 2^{-16.33}$ and, hence, our distinguisher needs approximately $2^{32.66}$ bits to succeed.

This shows that low weight feedback polynomials can be easily and efficiently attacked.

5.4 Related Work

The paper [GSB+05] independently presented cryptanalysis of the bit-search generator. Differently from the attacks in this chapter that paper instead assumed that each one in the keystream corresponded to the sequence $(\overline{b}, b, \overline{b})$. By picking a keystream window of size $2L/3$ and assuming the expected case that there is $L/3$ zeros and $L/3$ ones the probability that the assumption holds is $2^{-L/3}$. Thus, doing this for $2^{L/3}$ windows we expect to have found a window for which the assumption is true. Since each try involves solving a system of linear equations the computational complexity is $O(L^3 2^{L/3})$ and the required amount of keystream is $O(L2^{L/3})$. This approach was then improved by looking at pairs of windows of size l with w ones each. The assumption that every one in a window stems from the sequence $(\overline{b}, b, \overline{b})$ holds with probability 2^{-w}. By considering all pairs of 2^w windows and for each pair guessing the distance between the windows it is shown that the internal state can be recovered with computational complexity $L^3 2^{L/4}$ and $2^{L/4}$ keystream bits.

The security of the BSG is based on the uncertainty about the number of input bits needed to output a one. However, there is no uncertainty when a zero is output. In [GSB+05], the authors also proposed two new variants of the BSG, namely the MBSG and the ABSG. In these variants, there is uncertainty about the number of input bits no matter whether a zero or a one is produced as keystream bit. The algorithms for the MBSG and the ABSG are given in Fig. 5.4. Both these two new algorithms have a rate of $1/3$ bits/clock. The security of these two algorithms was also investigated in [GSB+05]. The best attack has computational complexity $O(2^{L/2})$ and requires $O(L2^{L/2})$ keystream bits. It was also deduced that FBDD-based cryptanalysis will succeed with computational and memory complexity of about $O(2^{0.53L})$. Thus, in the case of cryptanalysis using few keystream bits, the MBSG and the ABSG seems weaker than the self-shrinking generator. On the other hand, they have a higher rate.

MBSG Algorithm

```
i = 0; j = 0;
while (1)
    z[j] = s[i];
    i++;
    while (s[i] == 0)  i++;
    i++; j++
```

ABSG Algorithm

```
i = 0; j = 0;
while (1)
    b = s[i],  z[j] = s[i+1];
    i++;
    while (s[i] ! = b)  i++;
    i++; j++;
```

Figure 5.4: The MBSG and the ABSG algorithms, two variants of the BSG algorithm.

Finally we mention the stream cipher DECIM [BBC+05]. It is a stream cipher in the eSTREAM project and at the time of writing it has advanced to the third and last phase of the project. DECIM incorporates the ABSG algorithm as part of the construction.

5.5 Summary

The bit-search generator, proposed by Gouget and Sibert has been considered and an equivalent description based on the differential of the input sequence has been given. We propose an efficient attack that recovers the differential sequence, and hence, the key. The BSG is very similar to the self-shrinking generator and we find that the key recovery attacks presented here are more efficient than any known key recovery attack on the self-shrinking generator. The basis for a distinguishing attack is also described and we show that if the feedback polynomial is not carefully chosen, the BSG may be prone to efficient distinguishing attacks.

6

Cryptanalysis of the Pomaranch Family of Stream Ciphers

Pomaranch is one of the candidates in the eSTREAM [ECR] project. It is designed to be efficient both in hardware and software. The cipher is an interesting construction since it introduces a new approach to the design of LFSR based stream ciphers. The self-shrinking generator and the bit-search generator covered in chapters 4 and 5 introduce nonlinearity into a linearly generated sequence by applying an algorithm to the output of an LFSR. This results in an irregular decimation of the LFSR sequence for the self-shrinking generator and an irregular decimation of the differential of the LFSR sequence for the bit-search generator. In both cases, the number of bits discarded between consecutive output bits is unknown to an attacker. Another way to decimate the LFSR sequence is to clock the register irregularly. However, these approaches will introduce some problems. First, an output buffer is needed in hardware if the keystream has to be regularly produced. Second, the construction is likely to be vulnerable to timing and power attacks. The Pomaranch stream ciphers use a novel technique to avoid these problems. The registers are based on a new idea, called jump registers. In each update of the registers, the next state is one of two possible states. Which one it jumps to is key dependent and thus unknown to the attacker. The main advantage is that the update of the ciphers behaves like irregular clocking but the problems involved in irregular clocking are avoided.

There are 3 different versions of Pomaranch. Version 2 was designed in response to the attack on Version 1 and Version 3 was designed as a response to the attack on Version 2. This chapter is based on the two papers [HJ06b] and [EHJ06] and includes key recovery and distinguishing attacks on Versions 1 and 2. It also includes distinguishing attacks on Version 3.

The chapter is outlined as follows. Section 6.1 gives a theoretical background to the concept of jump registers. Section 6.2 describes the details of Pomaranch Version 1. This is followed by an attack on Version 1 in Section 6.3. Section 6.4 introduces Version 2 which is immune to the attack on Version 1. In Section 6.5 we give a new algorithm that shows how to easily find a biased linear approximation under certain circumstances. This algorithm is applied to Pomaranch Version 2 in Section 6.6. The details of Pomaranch Version 3 is given in Section 6.7. In Section 6.8 we give a distinguishing attack that can be mounted on all versions and variants of the cipher. The attack is described in a general way in order to provide theorems that can be used when designing future versions of the cipher. Section 6.9 describes another attack, using a special attack scenario, that can be applied to any stream cipher. It is shown to be successful on all versions and variants of Pomaranch. The chapter is summarized in Section 6.10.

6.1 Jump Registers

In this section, we will give an overview of the theory behind jump registers. These are the central building blocks in all Pomaranch stream ciphers. In order to describe the jump registers we need some background theory on matrices. Only the theory that is interesting for the jump registers is covered here. To every square matrix we can associate a *characteristic polynomial*.

Definition 6.1: Assume that we have an $n \times n$ matrix A over the finite field $\mathbb{F}_p$. The characteristic polynomial of A, denoted $p_A(x)$, is the polynomial

$$p_A(x) = \det(xI - A), \tag{6.1}$$

where I is the $n \times n$ identity matrix. □

The Cayley-Hamilton theorem states that $p_A(A) = 0$. Further, the transpose of A has the same characteristic polynomial as A itself.

Definition 6.2: Two $n \times n$ matrices A and B are similar if there exists an invertible matrix M such that

$$A = M^{-1}BM. \tag{6.2}$$

□

Similar matrices share many properties, e.g., determinant, eigenvalues, trace, rank and characteristic polynomial.

Definition 6.3: The companion matrix $C(f)$ to the degree n monic polynomial $f(x) = c_0 + c_1 x + \ldots + c_{n-1}x^{n-1} + x^n$ is the $n \times n$ matrix

$$C(f) = \begin{pmatrix} 0 & 1 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \\ -c_0 & -c_1 & -c_2 & -c_3 & \ldots & -c_{n-1} \end{pmatrix}. \tag{6.3}$$

□

The characteristic polynomial of $C(f)$ is $f(x)$. If $f(x)$ is a primitive polynomial over the finite field $\mathbb{F}_p$, then the companion matrix is a generator to an extension field isomorphic to $\mathbb{F}_{p^n}$. This follows from the Cayley-Hamilton theorem. We now tie Definition 6.1, 6.2 and 6.3 together with the following important fact. If the characteristic polynomial of a matrix A is irreducible over the finite field $\mathbb{F}_p$, then A is similar to the companion matrix of the characteristic polynomial of A

$$C(p_A(x)) = M^{-1}AM. \tag{6.4}$$

This is a special case of writing A in its *rational canonical form*, also known as the *Frobenius canonical form*. Now, let A be the transition matrix for a linear finite state machine (LFSM) of size L, not necessarily an LFSR and let $\mathbf{s}(t) = (s_L(t), s_{L-1}(t), \ldots, s_1(t))$ be the state at time t. Then

$$\mathbf{s}(t) = A^t\mathbf{s}(0) = MC(p_A(x))^tM^{-1}\mathbf{s}(0) \Rightarrow M^{-1}\mathbf{s}(t) = C(p_A(x))^tM^{-1}\mathbf{s}(0). \tag{6.5}$$

Thus, the LFSM defined by A is the same as the LFSR specified by the characteristic polynomial $p_A(x)$ except for a linear transformation. It is well known that an LFSR with primitive characteristic polynomial will produce a maximum length sequence. Thus, any transition matrix with primitive characteristic polynomial will make the state $\mathbf{s}(t)$ go through all non-zero states before returning to the initial state.

We are now ready to discuss jump registers. For simplicity, we restrict ourselves to the field $\mathbb{F}_2$. The linear transformation matrix A used for jump registers is of the form

$$A = \begin{pmatrix} d_L & 1 & 0 & 0 & \ldots & 0 \\ 0 & d_{L-1} & 1 & 0 & \ldots & 0 \\ 0 & 0 & d_{L-2} & 1 & \ldots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \ldots & 0 & d_2 & 1 \\ 1 & t_{L-1} & \ldots & \ldots & t_2 & d_1 + t_1 \end{pmatrix}. \tag{6.6}$$

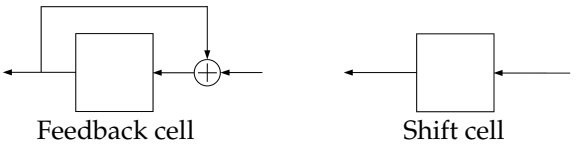

Figure 6.1: F-cell and S-cell used in jump registers.

The idea is to, at each time instance, multiply the state vector with one of two possible transition matrices. For jump registers, the state $\mathbf{s}(t)$ is multiplied by either A or $A+I$. The integer J satisfying $A^J = A+I$ is called the jump index. Clearly, multiplying by A^J and $A+I$ is equivalent and corresponds to jumping J steps in the state space. Moreover, if the characteristic polynomial $p_A(x)$ of A is primitive, there always exists an integer J such that $A^J = A+I$. We refer to [Jan05] for further mathematical background on jump registers.

Any matrix of the form (6.6) has characteristic polynomial

$$p_A(x) = 1 + \sum_{i=0}^{L-1} t_i \prod_{j=i+1}^{L} (d_j + x). \tag{6.7}$$

An important advantage of implementing the transition matrix in this way is the simplicity of implementation. Since only the diagonal elements are changed when going from A to $A+I$, and the rest of the matrix is equivalent to an LFSR update, it is very easy to implement the update of this LFSM in hardware. Each of the L state elements can be implemented in one of two ways, as a feedback cell (F-cell) or as a shift cell (S-cell), see Fig. 6.1. The values of d_i in (6.6) determines if a cell is currently implemented as an F-cell or an S-cell. When multiplying the state with $A+I$ instead of A, all F-cells are turned into S-cells and vice versa.

6.2 Pomaranch Version 1

The first version of Pomaranch was initially called Cascade Jump Controlled Sequence Generator (CJCSG). This name reflects the design principle in that it uses a cascade of 9 jump registers, $R_1, \ldots, R_9$, see Fig. 6.2. Pomaranch Version 1 supports a key size of 128 bits and an IV in the range of 64 to 112 bits. Each jump register has a length of $L = 14$ bits. All jump registers are identical, i.e., the state transition matrix A is chosen to be the same for all registers. The characteristic polynomial of the LFSM defined by A is

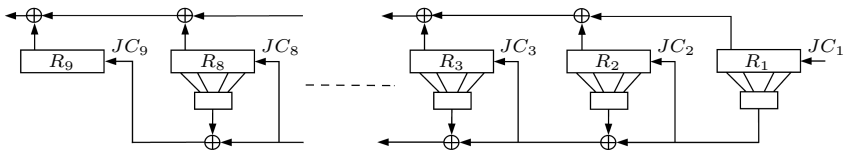

Figure 6.2: Overview of the Pomaranch design principle.

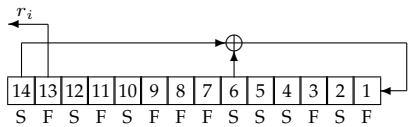

Figure 6.3: The jump register used in Pomaranch Version 1.

primitive. At all times, half of the register cells are S-cells and half are F-cells, see Fig. 6.3. The current implementation of each cell (S-cell or F-cell) is determined by a Jump Control bit, JC. If $JC = 0$, the state is updated corresponding to multiplication by A and if $JC = 1$, the update corresponds to multiplication by $A + I$. The state of the registers R_1 to R_8 are filtered through a nonlinear key dependent function, f_{K_i} producing output bits c_1 to c_8. The 128-bit key is divided into 8 parts of 16 bits each and part i is used in f_{K_i} The jump control bit for register i, denoted JC_i, is then calculated as

$$JC_i = c_1 \oplus \ldots \oplus c_{i-1}, \quad (i = 2, \ldots, 9), \tag{6.8}$$

as seen in Fig. 6.2. The jump control for the first register, JC_1, is always set to 0. The keystream at time t, denoted $z(t)$, is taken as the binary xor of all 9 bits at position 13, denoted $r_i(t)$, in the registers.

The initialization of the cipher will not affect our analysis. All attacks will be on the keystream generation phase. Instead, we refer to the design documents for a detailed description of the initialization.

6.3 Biased Linear Relations in Jump Register Outputs

The size of the registers is only 14. This suggests that it might be possible to mount a divide-and-conquer kind of attack on the cipher. Since the registers are updated linearly each new output bit will be linearly dependent on the initial state bits. Hence, for any given JC-sequence of length 14

there will be a linear relation in 15 output bits that will always hold, i.e., there is an array $\ell = (\ell_0, \ell_1, \ldots, \ell_{14})$, $\ell_i \in \{0, 1\}$ such that $\bigoplus_{i=0}^{14} \ell_i r(t+i) = 0$. Considering all possible JC-sequences, the possible values for ℓ may not be evenly distributed and thus, there might be linear relations that are more probable than others. This is the idea behind the attack on the keystream generation phase of Pomaranch Version 1. That attack was described in [Kha05] and the results in this section are taken from that paper. Before continuing we introduce the notation $a \stackrel{p}{=} b$ meaning that a and b are equal with probability p. By exhaustively searching all 2^{14} possible values for $JC(t), JC(t+1), \ldots, JC(t+13)$ it can be seen that the two linear relations

$$r_i(t) \oplus r_i(t+8) \oplus r_i(t+14) \stackrel{p}{=} 0, \tag{6.9}$$

$$r_i(t) \oplus r_i(t+8) \oplus r_i(t+13) \oplus r_i(t+14) \stackrel{p}{=} 0, \tag{6.10}$$

holds with probability $p = 1/2(1 + 840/2^{14})$. Thus, the bias is $\varepsilon = 840/2^{14} = 2^{-4.286}$. These biased relations will hold for the output of registers R_i for $2 \leq i \leq 9$. Using (6.9) we can, according to (2.58), write

$$\bigoplus_{i=2}^{9} r_i(t) \oplus r_i(t+8) \oplus r_i(t+14) \stackrel{p'}{=} 0 \tag{6.11}$$

which holds with probability $p' = 1/2(1 + \varepsilon_{tot})$ with $\varepsilon_{tot} = (840/2^{14})^8 = 2^{-34.286}$. Since $z(t) = \bigoplus_{i=1}^{9} r_i(t)$ we have

$$z(t) \oplus z(t+8) \oplus z(t+14) \stackrel{p'}{=} r_1(t) \oplus r_1(t+8) \oplus r_1(t+14). \tag{6.12}$$

Exhaustively searching the state of register R_1 we can, according to (2.61), distinguish the correct state using about

$$N = \frac{14 \cdot 2 \ln 2}{\varepsilon_{tot}^2} = 2^{72.9} \tag{6.13}$$

samples. Using both (6.9) and (6.10) we now have a distinguishing attack[1] requiring $2^{71.9}$ keystream bits and computational complexity $2^{14} 2^{72.9} = 2^{86.9}$.

When the state of R_1 has been recovered we guess the state of R_2 together with the 16 key bits used in f_{K_1}. Similarly as before we have

$$\bigoplus_{i=3}^{9} r_i(t) \oplus r_i(t+8) \oplus r_i(t+14) \stackrel{p''}{=} 0 \tag{6.14}$$

[1]The fact that we already here have a distinguishing attack is not mentioned in [Kha05].

Attack Complexities		
Type of Attack	Amount of Keystream Needed	Computational Complexity
Distinguishing Attack	$2^{71.9}$	$2^{86.9}$
Key recovery Attack	$2^{71.9}$	$2^{95.4}$

Table 6.1: Attack complexities for Pomaranch Version 1.

with $p'' = 1/2(1 + \varepsilon_{tot})$ with $\varepsilon_{tot} = (840/2^{14})^7 = 2^{-30.000}$. With R_1 known we need

$$N = \frac{30 \cdot 2 \ln 2}{\varepsilon_{tot}^2} = 2^{65.4} \tag{6.15}$$

samples and a computational complexity of $2^{30}2^{65.4} = 2^{95.4}$ in order to determine R_2 and the 16 bits of the key. After this the rest of the key can be recovered in the same way. It is easy to see that the following steps are computationally easier than the first two steps. The attack complexities are summarized in table 6.1.

6.4 Pomaranch Version 2 - Improving Jump Register Parameters

In [CGJ06], the authors presented an attack on Pomaranch targeting a weakness in the initialization procedure, not all IV bits are diffused into the whole state. This attack together with the attack in [Kha05] showed that the construction had to be improved.

In [JHK06] a theoretical analysis was done based on the attack in [Kha05]. Let $C(x)$ be the characteristic polynomial of A. Then for a given jump control sequence we have the equality

$$\sum_{i=0}^{L} \ell_i x^{i-k_i}(x+1)^{k_i} = C(x), \tag{6.16}$$

where k_i is the binary weight of the vector $(JC(t), \ldots, JC(t+i-1))$. Note that the coefficients in (6.16) are in $\mathbb{F}_2$. Using (6.16) a straightforward $O(L)$ approach to find the values of ℓ_i was described, giving the linear relation for this particular jump control sequence. Hence the most common linear relation among the 2^L sequences was found in $O(L2^L)$. Based on this, the parameters of the jump registers were tweaked such that there were no linear relations of length $L + 1$ that gave a bias large enough to mount the

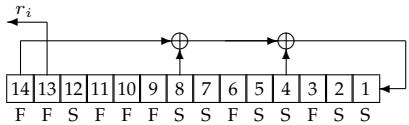

Figure 6.4: The jump register used in Pomaranch Version 2.

attack in Section 6.3 with complexity lower than exhaustive key search. The new jump registers were implemented as shown in Fig. 6.4. A variant of Pomaranch Version 2 that supports a key size of 80 bits was also introduced. The only difference between the 128-bit variant and the 80-bit variant is that the number of jump registers is reduced to 6 instead. Everything else in the design is left unchanged.

6.5 A New Algorithm That Can Find Linear Relations

Before we move on to the cryptanalysis of Pomaranch Version 2 we introduce a new algorithm that turns out to be helpful in our analysis. The idea is to not look at binary linear relations and their distance to the random distribution, but instead to consider vectors.

6.5.1 Vectorial Representation of a Linear Approximation

In this section we consider the advantage of representing a biased linear relation as a vector instead of as a binary relation. We give two propositions, which will lead us to a simple algorithm that helps us to find biased linear relations. Both propositions follow from basic information theory, see e.g., [CT91].

Proposition 6.1: Let $z_{t_1}, z_{t_2}, \ldots, z_{t_m}$ be binary random variables and let the vector $(z_{t_1}, z_{t_2}, \ldots, z_{t_m})$ follow the size 2^m distribution P_0. Let the sum of the variables $z = z_{t_1} \oplus z_{t_2} \oplus \ldots \oplus z_{t_m}$ follow the distribution P_0'. Then

$$D\left(P_0 \| P_1\right) \geq D\left(P_0' \| P_1'\right) \tag{6.17}$$

for any size 2^m distribution P_1 with corresponding distribution P_1' being the sum of the variables.

Proof: Denote by $a_i^{(e)}$ $(0 \leq i < 2^{m-1})$ the probabilities of the vectors in P_0 with even Hamming weight and correspondingly by $a_i^{(o)}$ $(0 \leq i < 2^{m-1})$ the

probabilities of the vectors with odd Hamming weight. Similarly, the probabilities of the vectors in distribution P_1 will be denoted $b_i^{(e)}$ and $b_i^{(o)}$ ($0 \leq i < 2^{m-1}$) respectively. Hence, we want to show that

$$\sum_i a_i^{(e)} \log \frac{a_i^{(e)}}{b_i^{(e)}} + \sum_i a_i^{(o)} \log \frac{a_i^{(o)}}{b_i^{(o)}} \geq \left(\sum_i a_i^{(e)}\right) \log \frac{\sum_i a_i^{(e)}}{\sum_i b_i^{(e)}} + \left(\sum_i a_i^{(o)}\right) \log \frac{\sum_i a_i^{(o)}}{\sum_i b_i^{(o)}}. \quad (6.18)$$

Let $E\{\cdot\}$ denote the expected value. Jensen's inequality states that for a convex function f and random variable X it holds that

$$E\{f(X)\} \geq f(E\{X\}). \quad (6.19)$$

If f is strictly convex, the equality in (6.19) implies that X is a constant. We use the fact that $t \log t$ is a strictly convex function and introduce $\alpha_i = \Pr(t_i \log t_i)$. If we consider only the first term on the left hand side and right hand side of (6.18) and putting $t_i = a_i^{(e)}/b_i^{(e)}$ we can write

$$\sum_i \alpha_i \frac{a_i^{(e)}}{b_i^{(e)}} \log \frac{a_i^{(e)}}{b_i^{(e)}} \geq \left(\sum_i \alpha_i \frac{a_i^{(e)}}{b_i^{(e)}}\right) \log \sum_i \alpha_i \frac{a_i^{(e)}}{b_i^{(e)}}. \quad (6.20)$$

This will hold for any choice of α_i as long as $\alpha_i \geq 0$ and $\sum_i \alpha_i = 1$. Hence we can put $\alpha_i = b_i^{(e)} / \sum_i b_i^{(e)}$ and it follows that

$$\sum_i a_i^{(e)} \log \frac{a_i^{(e)}}{b_i^{(e)}} \geq \left(\sum_i a_i^{(e)}\right) \log \frac{\sum_i a_i^{(e)}}{\sum_i b_i^{(e)}}. \quad (6.21)$$

Doing the same thing for the second terms in (6.18) will end the proof. ■

Proposition 6.1 implies that we can never lose anything by considering a linear approximation as a binary vector. Moreover, if the distribution of the vectors is such that for all vectors with even Hamming weight the probability is the same and for all vectors with odd Hamming weight the probability is the same, then there is no additional gain in using vectorial representation. We continue the analysis of vectorial representation with the following proposition.

Proposition 6.2: Assume that we have a binary vectorial distribution of size 2^m denoted P_0 and the size 2^m uniform distribution P_1. Adding a variable to the length m vector that is statistically independent with all other variables will not affect the relative entropy between two distributions. Furthermore, adding a variable that is correlated with other variables will increase the relative entropy.

Proof: Let $\mathbf{X} = (X_0, X_1, \ldots, X_{m-1})$. Using the chain rule for relative entropy we write

$$\begin{aligned} &D\left(P_0[\mathbf{x}, x_m] \parallel P_1[\mathbf{x}, x_m]\right) = \\ &\qquad D\left(P_0[\mathbf{x}] \parallel P_1[\mathbf{x}]\right) + D\left(P_0[x_m|\mathbf{x}] \parallel P_1[x_m|\mathbf{x}]\right) \end{aligned} \tag{6.22}$$

where the last term is zero if and only if $\Pr_{P_0}[x_m|\mathbf{x}] = 1/2,\ x_m \in \{0, 1\}$. ■

In the next section we show how these results can be used to find biased linear approximations of nonlinear blocks.

6.5.2 Finding a Biased Linear Approximation

In this section the theory developed in Section 6.5.1 is used to find biased linear approximations. If we have a linear approximation $x_{t_1} \oplus x_{t_2} \oplus \ldots \oplus x_{t_\mu}$ and add a variable, $x_{t_{\mu+1}}$, that is uniformly distributed and statistically independent with the other variables, the distribution of the resulting approximation is uniform and the approximation is useless. It can not be used in a distinguisher. If we instead write the approximation as a vector, $(x_{t_1}, x_{t_2}, \ldots, x_{t_\mu})$, then, according to Proposition 6.2, it does not matter how many uniformly distributed and independent variables we add to the vector. As long as all variables from the approximation are present, the relative entropy will never decrease and the vector can be used in a distinguisher.

Consider a cipher containing a relatively small building block B. If the distribution of the output bits, or the distribution of a linear equation of some output bits, can be found, then it is easy to search through all linear relations in order to determine which is most biased. However, as the amount of output bits to be considered increases, the number of possible equations increases exponentially. Considering m consecutive output bits there are 2^m possible equations. If no biased equation is found among these equations, one more output bit has to be considered and an additional 2^m equations has to be checked. If checking one equation requires 2^k computation steps, checking all equations involving m bits will require a computational complexity of 2^{k+m}. Instead, by considering the output bits as a vector, the computational complexity will be limited to 2^k. On the other hand, the memory complexity will be 2^m since the distribution of a length m vector needs to be kept in memory. Assume that the building block B has a biased linear relation involving some of the output bits $x_0, x_1, \ldots, x_{m-1}$ with a significantly larger bias than any linear relation involving less than m consecutive output bits. Let $\mathbf{x_m} = (x_0, x_1, \ldots, x_{m-1})$. Then

$$D\left(P_0[\mathbf{x_m}] \parallel P_1[\mathbf{x_m}]\right) \gg D\left(P_0[\mathbf{x_{m-1}}] \parallel P_1[\mathbf{x_{m-1}}]\right), \tag{6.23}$$

where P_0 is the cipher distribution and P_1 is the uniform distribution. Hence, we can start with the vector $\mathbf{x_1} = (x_0)$ and add one extra variable at

a time. If, at step m, considering the vector $\mathbf{x_m} = (x_0, x_1, \ldots, x_{m-1})$, the relative entropy between the distribution of $\mathbf{x_m}$ and the uniform distribution increases significantly, there might be a biased linear relation involving the variables x_0 and x_{m-1} and zero or more variables x_i ($1 \leq i \leq m-2$). In order to find the other variables in the biased linear relation we consider the vector $\mathbf{x_m^{(i)}}$ which we define as the vector $\mathbf{x_m}$ with the variable x_i removed. If

$$D\left(P_0[\mathbf{x_m^{(i)}}] \parallel P_1[\mathbf{x_m^{(i)}}]\right) \approx D\left(P_0[\mathbf{x_m}] \parallel P_1[\mathbf{x_m}]\right) \tag{6.24}$$

then the variable x_i is *not* present in the linear approximation. Note that we do not use equality in (6.24) since the variable i can be present in some biased linear relation but still not in the most biased relation. We are only interested in the most biased relation and if the variable x_i is present in this relation the decrease in relative entropy will be significant, not just approximate. Of course we can continue increasing the length of the vector, hoping to find even better linear approximations. Perhaps the algorithm is best explained using a concrete example and Pomaranch Version 2 turns out to be very suitable.

6.6 Algorithm Applied to Pomaranch Version 2

In this section we show how we can use the algorithm described in Section 6.5.2 to find a heavily biased linear relation in the stream cipher Pomaranch Version 2. We also show that the existence of this linear relation makes it possible to mount both distinguishing and key recovery attacks on the cipher.

6.6.1 New Attack on Pomaranch Version 2

In this section we show that it is still possible, using our proposed algorithm, to find linear relations in the output bits that can be used in an attack. The problem with the theoretical analysis in [JHK06], using (6.16), is that it only considers relations of length $L+1$. The same analysis is not applicable to relations involving bits further apart since the characteristic polynomial of A is of degree L. Consequently, the design parameters for Pomaranch Version 2 are optimized for the cipher to resist correlation attacks based on relations of length $L+1$ and it successfully does so.

Unfortunately it is not enough to only consider these relations. It is possible that there are relations involving bits further apart that are more biased than any relation involving only bits $L+1$ positions apart. The algorithm proposed in Section 6.5.2 is very suitable for Pomaranch. The shift registers are of length 14 which is easy to search exhaustively. Moreover, all registers

```
for all 2^14 − 1 initial states
    for all 2^(i−1) possible JC-sequences
        clock the jump register i−1 times;
        P_0[(z_0, ..., z_(i−1))] + +;
    end for
end for
```

Figure 6.5: Algorithm to find the distribution for i consecutive output bits. The actual implementation can be recursive to make it faster.

are identical. Register R_1 will have $JC_1(t) = 0, \forall t$ and will thus behave like a regularly clocked shift register with primitive feedback polynomial. Hence, the register R_1 will not be considered using our algorithm. Instead, it will be exhaustively searched. Registers R_i ($2 \leq i \leq 9$) will have JC_i determined by a key dependent function. Since these registers are identical, finding a biased linear approximation in the output bits of one register means that all other registers (except R_1) will have this biased approximation. In order to find a good approximation, we look at vectors of consecutive output bits. When calculating the bias of these vectors, the following three assumptions will be used:

(*i*) All states of the registers will have the same probability, except the all-zero state, which has probability 0.

(*ii*) All JC-sequences will have the same probability.

(*iii*) All jump sequences, JC_i ($2 \leq i \leq 9$), are independent.

Since the shift registers are of length 14, the first vector length we check is 15. The algorithm used to find the distribution for i consecutive output bits is given in Fig. 6.5. The output (keystream) of Pomaranch is given as the xor of the output bits of the registers. Hence, we need to find the bias of the xor of all 8 distributions. For k distributions, this can easily be done in $O(k2^{2n})$ time, where n is the size in bits for each random variable. This can be a bottleneck if the vectors are very large. A much more efficient algorithm for finding the distribution of a sum of random variables was given in [MJ05]. They show that the distribution for $\Pr(X_1 \oplus X_2 \oplus \ldots \oplus X_k)$ can be found in $O(kn2^n)$ time where all X_i are n-bit random variables. This algorithm was adopted in our implementation. The relative entropies between the vectors of length 15 to 23 and the random distribution have been given in Table 6.2.

We see that $D(P_0\|P_1)$ increases significantly when the vector reaches length 19. The fact that $D(P_0\|P_1)$ is much higher for $\mathbf{x_i}$ ($i \geq 19$) tells us that there might exist a heavily biased linear approximation involving

Vector Length	$D(P_0 \| P_1)$
15	$2^{-111.914}$
16	$2^{-108.603}$
17	$2^{-107.671}$
18	$2^{-107.108}$
19	$2^{-75.849}$
20	$2^{-74.849}$
21	$2^{-74.264}$
22	$2^{-73.849}$
23	$2^{-73.527}$

Table 6.2: Relative entropy between output vectors and the random distribution.

i	$\mathbf{x_{19}^{(i)}}$	i	$\mathbf{x_{19}^{(i)}}$
1	$2^{-75.851}$	10	$2^{-75.849}$
2	$2^{-105.383}$	11	$2^{-75.849}$
3	$2^{-75.849}$	12	$2^{-75.849}$
4	$2^{-75.849}$	13	$2^{-75.849}$
5	$2^{-76.077}$	14	$2^{-75.849}$
6	$2^{-105.264}$	15	$2^{-75.849}$
7	$2^{-75.849}$	16	$2^{-75.849}$
8	$2^{-75.849}$	17	$2^{-76.849}$
9	$2^{-75.849}$		

Table 6.3: Relative entropy when bit i is excluded from the vector $\mathbf{x_{19}}$.

the bits x_0, x_{18} and zero or more bits x_i ($1 \leq i \leq 17$). To find the particular linear approximation that allows us to attack the cipher we look at $D\left(P_0[\mathbf{x_{19}^{(i)}}] \parallel P_1[\mathbf{x_{19}^{(i)}}]\right)$, as suggested by our proposed algorithm. The result is given in Table 6.3.

We see that when x_2 or x_6 are removed, the relative entropy is almost as when $\mathbf{x_{18}}$ was considered. This implies that the heavily biased linear relation is

$$z(t) \oplus z(t+2) \oplus z(t+6) \oplus z(t+18) = 0. \tag{6.25}$$

It is possible that the figures in Table 6.3 stems from the fact that the vector $(z(t), z(t+2), z(t+6), z(t+18))$ is heavily biased but the xor of the variables has a much smaller bias or even no bias at all. However, checking the relative entropy between (6.25) and random, which is $2^{-77.080}$, confirms that the figures in Table 6.3 stems from the biased linear relation. In any cipher, constructed using jump registers together with some other function, this relation would have been an important part of linear cryptanalysis. In the specific case of Pomaranch, which uses the output bits from the jump registers immediately in the output function it is even better to consider the total vector of as many bits as possible, since this will give us more information. Though, most of this information stems immediately from this relation. The discovery of the biased linear relation given by our algorithm may help the designers to get additional theoretical knowledge about the design principle of the Pomaranch family of stream ciphers.

6.6.2 Distinguishing and Key Recovery Attacks

In this section we give the details and complexities of the attacks on Pomaranch Version 2. Using output vectors of length 23, which is the largest vectors we were able to find the distribution for, we get the relative entropy

$$D(P_0[\mathbf{x_{23}}] \| P_1[\mathbf{x_{23}}]) = 2^{-73.527}. \tag{6.26}$$

From here, the attack follows the same steps as the attack on Pomaranch Version 1 given in Section 6.3. For the 128-bit variant, the amount of keystream needed in order to recover the state of R_1 is, according to (2.61), about

$$N = \frac{14}{D(P_0[\mathbf{x_{23}}] \| P_1[\mathbf{x_{23}}])} = 2^{77.33}. \tag{6.27}$$

The computational complexity is then $2^{14}2^{77.33} = 2^{91.33}$. As with Pomaranch Version 1, this distinguisher can be extended to a key recovery attack by guessing R_2 and the 16 key bits in f_{K_1}.

The sum of the stream generated by $R_3, \ldots, R_9$ has relative entropy $2^{-63.897}$ compared to the random distribution. The corresponding distinguisher needs

$$N = \frac{30}{2^{-63.897}} = 2^{68.80} \tag{6.28}$$

samples. The computational complexity for finding 16 bits of the key will then be $2^{68.80}2^{30} = 2^{98.80}$. Finding the remaining bits of the key will now be a relatively fast procedure. The 80-bit hardware oriented variant of Pomaranch Version 2 has the same structure as the 128-bit variant with the exception that only 6 registers are used. The relative entropy using 5 and 4 registers is $2^{-44.59}$ and $2^{-34.88}$ respectively. The corresponding distinguishing

Attack Complexities			
Type of Attack	Variant	Amount of keystream needed	Computational Complexity
Distinguishing Attack	80 bits	$2^{48.39}$	$2^{62.39}$
	128 bits	$2^{77.33}$	$2^{91.33}$
Key recovery Attack	80 bits	$2^{48.39}$	$2^{69.79}$
	128 bits	$2^{77.33}$	$2^{98.80}$

Table 6.4: Complexities of the attacks proposed on Pomaranch Version 2.

attack on the 80-bit version will require $2^{48.39}$ samples and a computational complexity of $2^{62.39}$. The key recovery attack will require $2^{48.39}$ keystream bits and has a computational complexity of $2^{69.79}$. Table 6.4 summarizes all proposed attacks on Pomaranch Version 2 and the corresponding complexities.

6.6.3 Simulation Results

When the distribution for the output vectors was theoretically calculated, we assumed that the initial states had the same probability, that all jump control sequences had the same probability and that all jump control sequences JC_i ($2 \leq i \leq 9$) were independent. To verify these assumptions, the real distribution was simulated for scaled down variants of the cipher. An interesting question here is the amount of samples that is needed in the simulation. Let the simulated distribution of the output from the jump registers be denoted P_0^*. As before, the *theoretical* distribution using the assumptions above is denoted P_0 and the uniform distribution is denoted P_1. The size of the distributions is denoted $|\mathcal{X}|$. We use the following theorem taken from [CT91].

Theorem 6.3: Let $X_1, X_2, \ldots, X_n$ be independent and identically distributed according to P_0. Then

$$\Pr\left(D(P_0^* \| P_0) > \epsilon\right) \leq 2^{-n\left(\epsilon - |\mathcal{X}| \frac{\log(n+1)}{n}\right)}. \tag{6.29}$$

If $D(P_0 \| P_1) = \mu$, then $\epsilon \leq \mu/2$. To reach the amount of samples where the error probability in (6.29) is less than or equal to 1, n should satisfy

$$n\left(\mu/2 - |\mathcal{X}| \frac{\log(n+1)}{n}\right) \geq 0 \Rightarrow \frac{n}{\log(n+1)} \geq \frac{2|\mathcal{X}|}{\mu}. \tag{6.30}$$

Jump Registers Used	$D(P_0 \| P_1)$	
	Theoretical	Simulated
1	$2^{-10.05}$	$2^{-9.82}$
2	$2^{-19.62}$	$2^{-19.36}$
3	$2^{-29.20}$	$2^{-29.27}$

Table 6.5: Simulated values for the distributions.

It is clear that the amount of samples needed increases exponentially with the vector length and in order to be able to simulate more than one register, we chose to simulate the distribution for the linear relation (6.25) instead of vectors. (Note that in the hypothesis test used in the cryptanalysis, this is not the case. The amount of samples needed then is still in the order of $1/D(P_0\|P_1)$. Stein's lemma is still applicable.) We found the distribution for 1, 2 and 3 registers. Using a random key, register R_1 was fed with the all zero JC-sequence. The output was taken as the xor of the output of the other registers, i.e. R_2, $R_2 \oplus R_3$ and $R_2 \oplus R_3 \oplus R_4$ respectively. In all cases, the amount of samples used was 2^{39} which, according to (6.30), should be enough. The simulated values for the relative entropy can be found in Table 6.5. From the simulated values, we conclude that the assumptions made in the calculation of the theoretical distributions are valid at least up to 3 registers. From this it should be safe to assume that the theoretical distributions are valid also up to 8 registers. Thus, the theoretical distributions can be used in the hypothesis test.

6.7 Pomaranch Version 3 - New Jump Registers

Following the attack given in Section 6.6, it is clear that Pomaranch has to be changed again. The new jump registers must not only resist attacks based on linear relations of size $L + 1$ bits, they also need to resist attacks when other linear relations are considered. A new design approach that made the attacks on Version 1 and Version 2 infeasible was to have 2 different kinds of jump registers, type 1 and type 2. A linear relation that has a high bias in jump registers of type 1 has probably a very low bias in jump registers of type 2. In addition, the jump registers used in Pomaranch Version 3 have $L = 18$. The new registers are given in Fig. 6.6. All odd numbered jump registers use type 1 registers and all even numbered registers are of type 2. Similar to Version 2, Version 3 also supports a variant with 80-bit key. As before, the reduced variant only uses 6 jump registers. Additionally, the output functions, from now on denoted H, of the 128-bit and the 80-bit variants

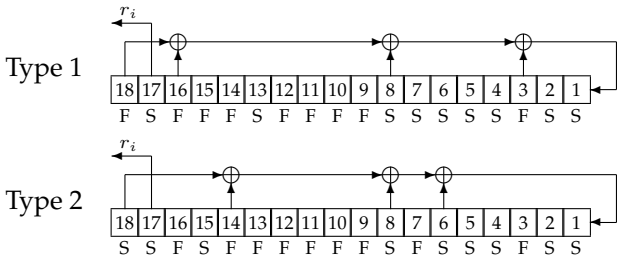

Figure 6.6: The jump registers used in Pomaranch Version 3.

are different. The output function for the 128-bit variant is a simple xor of the bit in position 17 of the jump registers, similar to the output function for the earlier versions. In the following, *linear output function* means an output function which is the xor of the output bits from *all* registers. For the 80-bit variant the output is given as

$$z(t) = G(r_1(t), \ldots, r_5(t)) \oplus r_6(t), \tag{6.31}$$

where

$$G(r_1(t), \ldots, r_5(t)) = r_1 \oplus r_2 \oplus r_5 \oplus r_1 r_3 \oplus r_2 r_4 \oplus r_1 r_3 r_4 \oplus r_2 r_3 r_4 \oplus r_3 r_4 r_5 \tag{6.32}$$

is a 1-resilient degree 3 Boolean function with nonlinearity 12. A keystream limitation has also been introduced in Version 3. The maximum number of keystream bits that can be produced for one key/IV pair is limited to 2^{64} bits.

6.8 General Distinguising Attacks on All Versions

In this section we propose another way of attacking Pomaranch. In addition to linear approximations, the relatively short period produced by some registers will be used. It turns out that this attack will succeed on all versions and variants of Pomaranch. Because of this, the cryptanalysis method will be given in a form as general as possible. We hope that the general theorems for attack complexities will be helpful in the design of future versions of the cipher. The attack given in this section is a distinguishing attack and is thus less powerful than the key recovery attacks on Version 1 and Version 2 given in previous sections.

6.8.1 Period of Registers

We start by looking at the period of the sequence produced by each register. The first jump register, denoted R_1 in Fig. 6.2, is during keystream generation mode fed by the jump sequence only containing zeros. The period of this register is denoted by T_1, hence $r_1(t) = r_1(t + T_1)$ with $T_1 = 2^L - 1$.

From R_1 a jump control sequence is calculated which controls the jumping of R_2. Assume that after T_1 clocks of R_1, register R_2 has jumped C steps. Then after T_1^2 clocks R_2 has jumped CT_1 steps, a multiple of T_1 and is thus back to its initial state. Using the same arguments, after T_1^3 clocks R_3 is back in its initial state. In general, if primitive characteristic polynomials are used for the registers, the period T_i for register R_i is

$$T_i = T_1^i, \tag{6.33}$$

and hence

$$r_i(t) = r_i(t + T_1^i). \tag{6.34}$$

Consequently, at time t and $t + T_1^p$, the output function H has p inputs with exactly the same value, namely the contribution from registers $R_1, \ldots, R_p$. This observation will be used in our attack.

6.8.2 Output Function

The output function used in Pomaranch can be a nonlinear Boolean function or just the linear xor of the output bits of each jump register. Our attack can be applied to both variants. The keystream bit at time t, denoted by $z(t)$, can be described as

$$z(t) = H\big(r_1(t), \ldots, r_n(t)\big). \tag{6.35}$$

Using the results from Section 6.8.1 and taking our samples as $z(t)+z(t+T_1^p)$ we can write the expression for the samples as

$$z(t) \oplus z(t+T_1^p) = H\big(r_1(t), \ldots, r_n(t)\big) \oplus H\big(r_1(t+T_1^p), \ldots, r_n(t+T_1^p)\big). \tag{6.36}$$

6.8.2.1 Linear Output Function

When the output function H is linear, i.e., $H(r_1, \ldots, r_n) = \bigoplus_{i=1}^{n} r_i$, and our samples are taken as $z(t) \oplus z(t + T_1^p)$, we know from Section 6.8.1 that p inputs to the output function are the same. Hence we can rewrite (6.36) as

$$z(t) \oplus z(t + T_1^p) = \bigoplus_{i=p+1}^{n} r_i(t) \oplus r_i(t + T_1^p). \tag{6.37}$$

6.8.2.2 Nonlinear Output Function

When the output function H is nonlinear, $r_i(t)$ and $r_i(t+T_1^p)$ will not cancel out in the keystream with probability one, as in Section 6.8.2.1. But, the input to H at time t and $t+T_1^p$ have p inputs $r_1, \ldots, r_p$ with the exact same value. This might lead to a biased distribution,

$$\begin{aligned}&\Pr\left(H\big(r_1(t),\ldots,r_n(t)\big) \oplus H\big(r_1(t+T_1^p),\ldots,r_n(t+T_1^p)\big) = 0\right) = \\ &\Pr\left(z(t) \oplus z(t+T_i^p) = 0\right) = \frac{1}{2}(1+\epsilon),\end{aligned} \tag{6.38}$$

where ϵ denotes the bias and $0 < |\epsilon| \leq 1$.

6.8.3 Linear Approximations of Jump Registers

In the attack, we need to find a linear approximation for the output bits of the jump registers that holds with probability different from one half. This is the same property as was used in the attacks on Pomaranch Version 1 and Version 2. In the search for good linear approximations we use the same assumptions as before regarding the states and the jump sequences. We search for a set $\mathcal{A}$ of size w such that

$$\Pr\left(\bigoplus_{i\in\mathcal{A}} r(t+i) = 0\right) = \frac{1}{2}(1+\varepsilon), \quad 0 < |\varepsilon| \leq 1, \tag{6.39}$$

i.e., the weight of the approximation is w and the terms are given by the set $\mathcal{A}$. For our attack to work it is important that the bias of this approximation is sufficiently high.

Since jump register R_1 will always have the all zero jump control sequence, the linear approximation will never apply for this register.

Recall that Pomaranch Version 3 uses two types of registers. In this case we are not interested in the most biased linear approximation of single registers. Instead we have to search for a linear approximation that has a good bias for both types of registers at the same time. This is much harder to find than a single approximation for one register. In the general case there can of course be even more types of registers.

6.8.4 Attacking Different Versions of Pomaranch

In general, a Pomaranch stream cipher can be designed using one or several types of jump registers. It can also use a linear or a nonlinear output function. In this section we recall the different design possibilities that has been used and give an expression for the number of samples needed in a distinguisher for each possibility.

The general expressions for the amount of keystream needed in an attack can be seen as a new design criterion for Pomaranch-like stream ciphers.

6.8.4.1 One Type of Registers with Linear Output Function

In this family of Pomaranch stream ciphers we assume that all jump register sections use the same type of register and that the output function H is linear, i.e., $H(r_1, \ldots, r_n) = \bigoplus_{i=1}^{n} r_i$. Pomaranch Version 1 and Pomaranch Version 2 are both included in this family.

Assume that we have found a linear approximation, as described in Section 6.8.3, of weight w of the register used. We consider samples at time t and $t + T_1^p$ such that p positions into H are the same according to Section 6.8.1. Our samples will be taken as

$$\bigoplus_{i\in\mathcal{A}} z(t+i) + \bigoplus_{i\in\mathcal{A}} z(t+i+T_1^p) = \bigoplus_{j=p+1}^{n} \bigoplus_{i\in\mathcal{A}} \big(r_j(t+i) + r_j(t+i+T_1^p)\big). \tag{6.40}$$

Since the bias of $\sum_{i\in\mathcal{A}} r_i(t+i)$ is ε and we have $2(n-p)$ such relations the total bias of the samples is given by

$$\varepsilon_{tot} = \varepsilon^{2(n-p)}. \tag{6.41}$$

We use the approximation (2.62) and write the number of samples needed in our distinguishing attack as

$$N = \frac{1}{\varepsilon_{tot}^2}. \tag{6.42}$$

This gives us the following theorem.

Theorem 6.4: The computational complexity and the number N of keystream bits needed to reliably distinguish the Pomaranch family of stream ciphers using a linear output function and n jump registers of the same type is upper bounded by

$$N \leq T_1^p + \frac{1}{\varepsilon^{4(n-p)}}, \quad p > 0, \tag{6.43}$$

where ε is the bias of the best linear approximation of the jump register.

6.8.4.2 Different Registers with Linear Output Function

In this family of generators different types of jump registers are used and the output function is assumed to be $H(r_1, \ldots, r_n) = \bigoplus_{i=1}^{n} r_i$.

This case is very similar to the case when all registers are of the same type. The difference is that, in this case, we are not looking for the best linear approximation of the registers separately. Instead, we have to find a linear approximation that have a bias for all the registers $R_{p+1}, \ldots, R_n$. This can be difficult if there are several types of registers. Approximations with a large bias for one type might have a very small bias for other types. Anyway, assume that we have found such a linear approximation. Our samples will still be taken as in (6.40). If we denote the bias for the approximation of register i by ε_i, then the total bias will be given as

$$\varepsilon_{tot} = \prod_{i=p+1}^{n} \varepsilon_i^2. \tag{6.44}$$

according to (2.58). Note that there are two approximations from each register.

Theorem 6.5: Assuming there is a linear relation that is biased in all registers. The computational complexity and the number N of keystream bits needed to reliably distinguish the Pomaranch family of stream ciphers using a linear output function and n jump registers of different types is upper bounded by

$$N \leq T_1^p + \frac{1}{\prod_{i=p+1}^{n} \varepsilon_i^4}, \quad p > 0, \tag{6.45}$$

where ε_i is the bias of jump register R_i.

The 128-bit variant of Pomaranch Version 3 belongs to a special subclass of this family, namely all registers in odd positions are of type 1 and registers in even positions are of type 2. In this case we only have to search for a linear approximation that is biased for type 1 and type 2 registers at the same time. The bias of $\bigoplus_{i \in \mathcal{A}} r_i(t+i)$ is denoted $\varepsilon_{type\ 1}$ and $\varepsilon_{type\ 2}$ respectively for the different registers. In total we have $2\lceil \frac{n-p}{2} \rceil$ type 1 relations and $2\lfloor \frac{n-p}{2} \rfloor$ type 2 relations when n is odd. Hence, the total bias of the samples is given by

$$\varepsilon_{tot} = \varepsilon_{type\ 1}^{2\lceil \frac{n-p}{2} \rceil} \varepsilon_{type\ 2}^{2\lfloor \frac{n-p}{2} \rfloor}. \tag{6.46}$$

If we apply Theorem 6.5 to the 128-bit variant of Pomaranch Version 3, the number of samples in the distinguisher is given by

$$N = T_1^p + \frac{1}{\varepsilon_{type\ 1}^{4\lceil \frac{n-p}{2} \rceil} \varepsilon_{type\ 2}^{4\lfloor \frac{n-p}{2} \rfloor}}. \tag{6.47}$$

6.8.4.3 Nonlinear Output Function

Now we consider the case when the output function is nonlinear. We restrict ourselves to the case when the output function H can be written on the form

$$H(r_1, \ldots, r_n) = G(r_1, \ldots, r_{n-1}) \oplus r_n. \tag{6.48}$$

The attack can easily be extended to output functions with more (or less) linear terms but to simplify the presentation, and the fact that the 80-bit variant of Pomaranch Version 3 is on this form, we only consider this special case here.

Our attacks on this family uses a biased linear approximation of R_n, see Section 6.8.3, together with the fact that the input to G at time t and $t + T_1^p$ have p inputs in common and hence in some cases a biased distribution, see Section 6.8.2.2.

Let ϵ denote the bias of $G\big(r_1(t), \ldots, r_{n-1}(t)\big) \oplus G\big(r_1(t+T_1^p), \ldots, r_{n-1}(t+T_i^p)$, and ε the bias of our linear approximation for R_n, which is $\bigoplus_{i\in\mathcal{A}} r_n(t+i)$. The samples are taken as

$$\begin{aligned}\bigoplus_{i\in\mathcal{A}} z(t+i) \oplus \bigoplus_{i\in\mathcal{A}} z(t+i+T_1^p) &= \bigoplus_{i\in\mathcal{A}} r_n(t+i) \oplus \bigoplus_{i\in\mathcal{A}} r_n(t+i+T_1^p) \\ \oplus \bigoplus_{i\in\mathcal{A}} \big(G\big(r_1(t+i), \ldots, r_{n-1}(t+i)\big) &\oplus G(r_1(t+i+T_1^p), \ldots, r_{n-1}(t+i+T_1^p))\big),\end{aligned} \tag{6.49}$$

and the bias of the samples is given by

$$\varepsilon_{tot} = \varepsilon^2 \epsilon^w. \tag{6.50}$$

This relation tells us that we need to keep the weight of the linear approximation of R_n as low as possible, i.e., the most biased relation may not be the most suitable.

Theorem 6.6: The computational complexity and the number N of keystream bits needed to reliably distinguish the Pomaranch family of stream ciphers using an output function of the form (6.48) is upper bounded by

$$N \leq T_1^p + \frac{1}{(\varepsilon^2\epsilon^w)^2}. \tag{6.51}$$

where ε is the a biased approximation of weight w of register R_n and ϵ is the bias of $G\big(r_1(t), \ldots, r_{n-1}(t)\big) \oplus G\big(r_1(t+T_1^p), \ldots, r_{n-1}(t+T_1^p)\big)$.

Note that in this presentation it does not matter if all registers are of the same type or if they are of different types. Since only register R_n is completely linear in the output function H, we only need to have an approximation of this register.

p	1	2	3	4	5	6	7
N	$2^{137.15}$	$2^{120.01}$	$2^{102.86}$	$2^{85.72}$	$2^{70.46}$	$2^{84.00}$	$2^{98.00}$

Table 6.6: Number of samples and computational complexity needed to distinguish Pomaranch Version 1 from random.

6.8.5 Attack Complexities for the Existing Versions of the Pomaranch Family

In this section, we look at the existing versions and variants of Pomaranch that have been proposed so far. These are Pomaranch Version 1, the 80-bit and 128-bit variants of Pomaranch Version 2 and the 80-and 128-bit variants of Pomaranch Version 3. Applying the distinguishing attack proposed in this section, we show that we can find distinguishers with better complexity than previously known for *all* 5 ciphers.

6.8.5.1 Pomaranch Version 1

In Pomaranch Version 1 all registers are the same, so the attack will be according to Section 6.8.4.1. The bias of the best known linear approximation for this register, as given in Section 6.3 and originally in [Kha05], is

$$\varepsilon = |2\Pr(r(t) + r(t+8) + r(t+14) = 0) - 1| = 2^{-4.286}. \tag{6.52}$$

Using Theorem 6.4 for different values of p we get Table 6.6. We see that the best attack is achieved when $p = 5$. The computational complexity and the amount of keystream needed is then $2^{70.46}$. It is interesting to note that the amount of keystream needed is basically the same as the one given in Table 6.1. The difference is that in the first attack we multiply by a factor $14 \cdot 2\ln 2$ since there are 2^{14} candidate sequences. Moreover, in the first attack we do not have the term T_1^5. In Table 6.1 we also counted the fact that we have two relations with the same bias reducing the amount of keystream by a factor 2. That can of course also be done in this case. However, it should be noted that the distance between the first and the last keystream bit in each sample is $T_1^5 \approx 2^{70}$ so this will still be a lower bound for the attack. Using vector representation can also help reduce the amount of keystream needed, but we still have a lower bound given by T_1^5. Using vector representation and $p = 4$ does not give enough information to have better complexity than $p = 5$ either. The bias will be too small. Finally, we see that the computational complexity of this distinguishing attack is far smaller than the attack given in Table 6.1 since we do not have to exhaustively search R_1.

p	1	2	3	4	5	6
N (80-bit)	$2^{95.76}$	$2^{76.61}$	$2^{57.46}$	$2^{56.00}$	$2^{70.00}$	$2^{84.00}$
N (128-bit)	$2^{153.22}$	$2^{134.06}$	$2^{114.91}$	$2^{95.76}$	$2^{76.62}$	$2^{84.00}$

Table 6.7: Number of samples needed to distinguish Pomaranch Version 2 according to Theorem 6.4.

6.8.5.2 Pomaranch Version 2

Similarly as in Pomaranch Version 1, in Pomaranch Version 2 all registers are the same and the attack will be performed according to Section 6.8.4.1. The bias of the best linear approximation for the register used is given by

$$\varepsilon = |2\Pr(r(t) + r(t+2) + r(t+6) + r(t+18) = 0) - 1| = 2^{-4.788}. \quad (6.53)$$

Using Theorem 6.4 for different values of p gives Table 6.7. For the 80-bit variant the computational complexity and the number of keystream bits needed is $2^{56.00}$. This is determined by T_1^4 since the number of samples needed in the distinguisher is only about 2^{38}. Comparing with Table 6.4 we see that we need more keystream bits in this attack. The attack in Table 6.4 combines 5 registers and exhaustively searches R_6. In this attack we do not have the possibility to combine 5 registers. Choosing $p = 4$ combines 4 registers[2] and choosing $p = 3$ combines 6 registers[3]. It is possible to slightly reduce the complexity of this attack by choosing $p = 3$ and use a vector representation as in Section 6.6. For the 128-bit variant the computational complexity and the number of keystream bits needed is $2^{76.62}$ when $p = 5$. As suggested by Table 6.2 the complexity of this attack can also be slightly reduced by considering vectors.

6.8.5.3 Pomaranch Version 3

There is a significant difference between the 80-bit and the 128-bit variants of Pomaranch Version 3, so this section will be divided into two parts.

80-bit Variant. The 80-bit variant of Pomaranch Version 3 uses a nonlinear output function, the attack will hence follow the procedure described in Section 6.8.4.3. The bias of

$$G\big(r_1(t), \ldots, r_5(t)\big) \oplus G\big(r_1(t + T_1^p), \ldots, r_5(t + T_1^p)\big). \quad (6.54)$$

was first estimated for different choices of p. The results are summarized in Table 6.8 The keystream per IV/key pair of Pomaranch Version 3 is limited

[2] R_5 and R_6 at two different time instances.
[3] R_4, R_5 and R_6 at two different time instances.

p	1	2	3	4	5
ϵ	0	2^{-4}	2^{-3}	2^{-2}	1

Table 6.8: The bias of $G\big(r_1(t),\ldots,r_5(t)\big) \oplus G\big(r_1(t+T_1^p),\ldots,r_5(t+T_1^p)\big)$ in the 80 bit variant of Pomaranch Version 3 for different values of p.

to 2^{64}. Because of this we limit p to $p \in \{1,2,3\}$, otherwise $T_1^p > 2^{64}$. We looked for a linear relation of R_6 that, together with a value of $p \in \{1,2,3\}$, minimizes the amount of keystream needed as given by Theorem 6.6. The best approximation found was

$$r_6(t) \oplus r_6(t+5) \oplus r_6(t+7) \oplus r_6(t+9) \oplus r_6(t+12) \oplus r_6(t+18) \quad (6.55)$$

which has weight $w = 6$ and bias $\varepsilon = 2^{-8.774}$. Using $p = 3$, the total bias of our samples using (6.55) is

$$\varepsilon_{tot} = (2^{-8.774})^2 \cdot (2^{-3})^6 = 2^{-35.548} \quad (6.56)$$

according to (6.50). The samples used in the attack are taken according to

$$\bigoplus_{i\in\mathcal{A}} z(t+i) \oplus \bigoplus_{i\in\mathcal{A}} z(t+i+T_1^3), \quad (6.57)$$

where $\mathcal{A} = \{0,5,7,9,12,18\}$. According to Theorem 6.6, the amount of keystream needed is $2^{54} + 2^{71.096} = 2^{71.096}$. This is also the computational complexity of the attack. In the specification of Pomaranch Version 3 the frame length (keystream per IV/key pair) is limited to 2^{64}. This does not prevent our attack since all samples will have this bias regardless of the key and IV used. We only need to consider 2^{64} keystream bits from $\lceil 2^{7.096} \rceil = 137$ different key/IV pairs.

Using vectors in this case does not seem useful. The attack uses one bias ϵ that stems from bits very far apart in the sequence produced by the registers. It also uses another bias ε stemming from a linear approximation in the output of one register. Thus, the bits in a vector are likely to be (almost) independent.

128-bit Variant. In Pomaranch Version 3 two different registers are used, so we start by searching for a linear approximation that is good for both types of registers. The best approximation we found was

$$r(t)\oplus r(t+1)\oplus r(t+2)\oplus r(t+5)\oplus r(t+7)\oplus r(t+11)\oplus r(t+12)\oplus r(t+15)\oplus r(t+21), \quad (6.58)$$

p	1	2	3	4	5	6	7	8
N	$2^{349.89}$	$2^{306.15}$	$2^{262.42}$	$2^{218.68}$	$2^{174.94}$	$2^{131.21}$	$2^{126.00}$	$2^{144.00}$

Table 6.9: Number of samples needed to distinguish the 128-bit variant of Pomaranch Version 3 according to (6.47).

which has the same bias for both types of registers, namely

$$\varepsilon_{even} = \varepsilon_{odd} = 2^{-10.934}. \tag{6.59}$$

Using (6.47) for different values of p we get Table 6.9. Our best distinguishing attack needs $2^{126.00}$ keystream bits. This figure is determined by $T_1^7 = 2^{126.00}$ so it is not possible to look at different key/IV pairs in this case since the distance between the bits in each sample has to be $2^{126.00}$. If the frame length is limited to 2^{64} it will not be possible to get any biased samples at all with $p = 7$.

We do not exclude the possibility to get a better attack by taking $p = 6$ and using a vector representation. However, our simulations show that the additional gain is very small when considering vectors. With $p = 6$ and vectors of length 23 we need $2^{128.40}$ keystream bits in an attack.

Nevertheless, a distinguishing attack requiring 2^{126} keystream bits suggests that the 128-bit variant offers less security than many other 128-bit ciphers that has not been successfully cryptanalyzed and that do not restrict the keystream length.

6.9 A Resynchronization Collision Attack

In this section we will give an attack that works for all currently existing Pomaranch ciphers. The size of the state in Pomaranch is always larger than twice the key size, e.g., the 128-bit variant of Pomaranch Version 3 has a state size of 290 bits. Thus, the generic time-memory tradeoff attacks will not be applicable in general. The attack in this section is a variant of the time-memory tradeoff attack and is generic for all stream ciphers. Let us divide the internal state of the cipher into two parts,

$$State = (State_K, State_{K,IV}), \tag{6.60}$$

where $State_K$ is a part of the state that statically holds the key and $State_{K,IV}$ is a part of the state that is updated, depending on both the key and the IV. If the key size $|K| > |State_{K,IV}|/2$, then the attack will always succeed with complexity below exhaustive key search. In Pomaranch, $State_K$ will consist of the $|K|$ key bits and $State_{K,IV}$ will consist of the register cells.

The attack scenario and the goal of the attack is somewhat different from usual. The scenario is given as follows. We assume that the key is fixed and that the cipher is initialized with many different IVs. Further, we assume that we have access to one long keystream sequence produced from one of the IVs, denoted IV_0. We intercept the ciphertext corresponding to many other IVs and we know the first l plaintext bits corresponding to every ciphertext. Our goal is to recover the rest of the plaintext for one of the messages.

We apply this attack scenario to Pomaranch. The key map used to produce the jump control bits is key dependent but independent of the IV. Hence, a fixed key will define a state graph of size $(2^L-1)^n \approx 2^{nL}$ states, where L is the register length and n the number of registers. Let a sample, $S_{IV_i}(t)$, generated from IV_i at time t, be a sequence of l consecutive keystream bits, i.e., $S(t) = \big(z(t), z(t+1), \ldots, z(t+l-1)\big)$. We first store a large amount of samples from IV_0 in a table. We would like to find another IV, denoted IV_c, that results in a sample such that

$$S_{IV_c}(t_c) = S_{IV_0}(t_0). \tag{6.61}$$

If a collision is found, then with high probability the following keystream of IV_0 and IV_c will also be identical. That means that if we just know the first l keystream bits generated by IV_c, we can predict future keystream bits from IV_c. The attack is visualized in Figure 6.7.

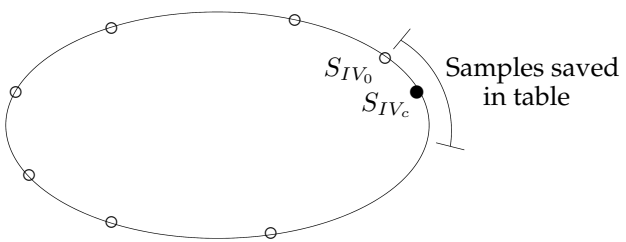

Figure 6.7: State graph for a fixed key, a sample is visualized by a small ring.

Assume that $2^{\beta nL}$ $(0 < \beta < 1)$ samples of length l from a keystream sequence of $2^{\beta nL}+l$ bits, originating from IV_0 and key K, is saved in a table. The table is then sorted with complexity $O(\beta nL \cdot 2^{\beta nL})$. This table covers a fraction of $2^{-(1-\beta)nL}$ of the entire cycle. The number of samples (IVs with l known keystream bits) we need to test to find a collision is geometrically distributed with expected value $2^{(1-\beta)nL}$. For each sample, a logarithmic

search with complexity $O(\beta nL)$ in the table is performed to see if there is a collision. To be sure that a collision in the table actually means that we have found a collision in the state cycle, l must be $l \approx nL$. The attack complexities are then given by

$$\begin{aligned} Keystream: &\quad 2^{\beta nL} + nL, \text{ from one IV and} \\ &\quad nL, \text{ from } 2^{(1-\beta)nL} \text{ IVs,} \\ Time: &\quad \beta nL 2^{\beta nL} + \beta nL 2^{(1-\beta)nL}, \\ Memory: &\quad nL 2^{\beta nL}, \end{aligned}$$

where $0 < \beta < 1$. By decreasing β it is possible to achieve smaller memory complexity at the expense of more IVs and higher time complexity. We can also see that the best time complexity is achieved when $\beta = 0.5$ for large nL.

6.9.1 Attack Complexities for Pomaranch

In this section we will look at the existing versions of Pomaranch and show that the resynchronization collision attack can be mounted with complexity significantly less than exhaustive key search. We assume $\beta = 0.5$ so the time complexity and the memory complexity in bits are equal. The 128-bit variants of Pomaranch Version 1 and Version 2 can be attacked using a table of size $2^{67.0}$ bytes together with keystream from $2^{63.0}$ different IVs. The 80-bit variant of Pomaranch Version 2 can be attacked using a table of $2^{45.4}$ bytes and $2^{42.0}$ different IVs. Pomaranch Version 3 uses larger registers, and the complexity of the attack on the 80-bit variant is a table of size $2^{57.8}$ bytes and $2^{54.0}$ IVs. The 128-bit variant needs a table of $2^{85.3}$ bytes and 2^{81} IVs. However, if we respect the maximum frame length of 2^{64} bits, we need to choose $\beta = 0.395$. Then we need a table of $2^{71.3}$ bytes and 2^{98} IVs. The time complexity is in this case 2^{104}.

The success probability of the attack has been simulated on a reduced version of the 128-bit variant of Pomaranch Version 3, using two registers. Choosing $\beta = 0.5$ implies that we know 2^{18} keystream bits from IV_0, we store all samples of length $nL = 36$ in a table, and that we need samples from 2^{18} different IVs in order to find a collision. The simulation results are summarized in Table 6.10. We also verified the attack using 3 registers. The attack given in this section suggests a new design criterion for the Pomaranch family of stream ciphers, namely that the total register length must be twice the keysize.

Success Rate		Number of IVs				
		2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
Table size	2^{17}	28	45	64	87	98
	2^{18}	38	**67**	91	98	100
	2^{19}	71	86	98	100	100

Table 6.10: Simulation results using 2 register Pomaranch Version 3 with linear output function, the table summarizes how many times the attack succeeds out of 100 attacks for a specific table size and number of IVs.

6.10 Summary

In this chapter we have seen several attacks on the stream cipher Pomaranch. The first attacks use linear approximations of the output sequences from the jump registers. Pomaranch Version 3 was designed such that there were no approximations that could be used in an attack. However, by also using the fact that the first few registers have a relatively short period, it was possible to eliminate the influence of these registers on the considered output. This made it possible to mount distinguishing attacks also on Pomaranch Version 3 by considering linear approximations.

Using more different registers is the most obvious way to increase the strength of the cipher. If all registers are unique it seems very unlikely that there is an approximation that is highly biased in all registers. The cipher description will not be as simple but the size in hardware and speed in software will remain more or less the same.

To resist the resynchronization collision attack the part of the state that is updated needs to be increased. This can be done in one of two ways.

- Update the part of the state that holds the key and is used in the filter function that determines the jump sequence.
- Increase the size of the registers.

Both alternatives seem to increase the hardware complexity of the algorithm and slow it down in software. It can be noted that the resynchronization collision attack may not be seen as a devastating weakness. Any block cipher in OFB mode would also be vulnerable to this attack if the size of the key is larger than half the block size. More discussion on this can be found in [EHJ07].

7

Cryptanalysis of the Achterbahn Family of Stream Ciphers

Like Pomaranch, the Achterbahn stream cipher was submitted to the eSTREAM [ECR] project. It is to be considered as a hardware efficient cipher, using a key size of 80 bits. However, the latest version of the cipher, Achterbahn-128/80 supports both 80 and 128 bit keys. There is a number of different versions and variants of the Achterbahn family and an exhaustive treatment of all variants and attacks will not be given here. This is motivated by the fact that the design was not considered secure enough to move to the third phase of eSTREAM. We focus on the latest version, Achterbahn-128/80, and the attack given in [HJ07]. This attack is an improved version of the previous attacks on older versions and variants and can also be used to break these. This chapter is outlined as follows. Section 7.1 will give the history of the Achterbahn cipher prior to the results in this chapter. In Section 7.2 we give a description of the cipher and in Section 7.3 we give an analysis showing how the construction can be cryptanalyzed. Section 7.4 will discuss the fact that not all variables used will be independent, resulting in important consequences for the attack complexity. Sections 7.5 and 7.6 show how the attack can be applied to Achterbahn-80 and Achterbahn-128 respectively and in Section 7.7 we show how the full key can be recovered. Further improvements and observations by other researchers are given in Section 7.8 and the history following our results are given in Section 7.9. The Chapter is summarized in Section 7.10.

7.1 History of Achterbahn, Part I

The reader not familiar with all the steps in the evolution of the Achterbahn family of stream ciphers might get confused when reading the papers concerning this family. Different names are mixed with different keystream limitations. The aim of this section is to clear this confusion. In this first part, the history prior to the result in this chapter will be given. The history following these results will be covered in Part II in Section 7.9.

The Achterbahn stream cipher [GGK05a] was submitted to the eSTREAM project in April 2005. The first cryptanalysis result was given in [JMM05], a paper submitted to the eSTREAM webpage in Sept 2005. A few weeks later, the designers submitted a note [GGK05b] in which they gave two new variants of the Boolean combining function which would both counter the attack. On the Achterbahn web page [Gam07] it was stated that Achterbahn using one of these two new Boolean functions was denoted Version 2[1]. When the original attack was published in FSE 2006 [JMM06], the paper included also attacks on the two new Boolean functions, denoting these by Achterbahn-v2 and Achterbahn-v3 respectively. Here, Achterbahn-v2 was the same as Version 2 as given on the web page. In [GGK06b], presented at SASC 2006, a new version of Achterbahn was introduced using more shift registers. This version is denoted Achterbahn-Version 2 by the designers (obviously, at this point the web page was changed). To avoid attacks, the amount of keystream used in the encryption process was limited to 2^{63} bits, also denoted the maximum frame length. An attack on Achterbahn-Version 2 was presented at SAC 2006 [HJ06a]. As a response to this attack, the latest version Achterbahn-128/80 was introduced for the second phase of eSTREAM to which Achterbahn qualified. The amount of keystream was now limited to 2^{64} bits. This chapter presents cryptanalysis of Achterbahn-128/80 and is based on [HJ07].

7.2 Description of Achterbahn-128/80

Despite the existence of previous versions, we will only explicitly describe the latest version, denoted Achterbahn-128/80. However, some differences between the current and previous versions will be mentioned to highlight the measures taken to resist attacks.

Achterbahn-128/80 is based on the idea behind the nonlinear combiner discussed in Section 2.2.6. The security, or insecurity, of this construction is quite well understood by now, see e.g., [BL05] and the references in that paper. Many attacks on nonlinear combiners use the fact that the registers have

[1]This version of the web page has been removed and the fact that it ever existed will probably be denied by the designers. However, a mirror is available from the author.

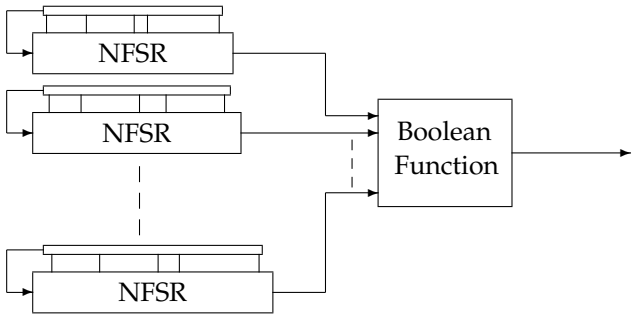

Figure 7.1: Overview of the Achterbahn design idea.

linear feedback. In order to protect against these attacks, the Achterbahn stream ciphers use nonlinear feedback shift registers (NFSRs), see Fig. 7.1. Thus, the ciphers consist only of a set of shift registers and a nonlinear Boolean output function. This makes it very simple and also very small in hardware.

7.2.1 Notation

The number of shift registers used is denoted n, shift register i will be denoted by R_i and the size of register i is denoted L_i. All registers are primitive, which in this context means that the period of register R_i is $2^{L_i} - 1$. We denote this period by T_i. Hence,

$$T_i = 2^{L_i} - 1. \tag{7.1}$$

The input bit to the Boolean function F (or G) from register R_i at time t will be denoted $x_i(t)$ and if the time instance t is fixed the simplified notation x_i will sometimes be used. The keystream is denoted $z(t)$ and the number of keystream bits that we need in an attack is denoted N. We will distinguish between the two variants by the names Achterbahn-80 and Achterbahn-128 though Achterbahn-80 is just a substructure of Achterbahn-128. Sometimes we also use Achterbahn as a collecting name for all Achterbahn ciphers.

7.2.2 Design Parameters

The design of Achterbahn-128/80 makes it possible to use either a 80-bit key or a 128-bit key. The number of registers used in Achterbahn-128 is $n = 13$ and their sizes are all numbers between 21 and 33 i.e.,

$$L_i = 21 + i, \quad 0 \leq i \leq 12. \tag{7.2}$$

The nonlinear feedback functions used in the registers are not used in our cryptanalysis and will not be given. All registers are autonomously driven and it suffices to say that all feedback functions are chosen such that the registers will enter all possible nonzero states before returning to the initial state. At each time instance t, the output Boolean function takes one bit x_i from each register, R_i $(0 \leq i \leq 12)$ as input and outputs the keystream bit z as $z = F(x_0, x_1, \ldots, x_{12})$ with

$$
\begin{aligned}
F(x_0, x_1, \ldots, x_{12}) = \; & x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_{12} \oplus x_0x_5 \oplus x_2x_{10} \\
& \oplus x_2x_{11} \oplus x_4x_8 \oplus x_4x_{12} \oplus x_5x_6 \oplus x_6x_8 \oplus x_6x_{10} \oplus x_6x_{11} \oplus x_6x_{12} \\
& \oplus x_7x_8 \oplus x_7x_{12} \oplus x_8x_9 \oplus x_8x_{10} \oplus x_9x_{10} \oplus x_9x_{11} \oplus x_9x_{12} \oplus x_{10}x_{12} \\
& \oplus x_0x_5x_8 \oplus x_0x_5x_{10} \oplus x_0x_5x_{11} \oplus x_0x_5x_{12} \oplus x_1x_2x_8 \oplus x_1x_2x_{12} \\
& \oplus x_1x_4x_{10} \oplus x_1x_4x_{11} \oplus x_1x_8x_9 \oplus x_1x_9x_{10} \oplus x_1x_9x_{11} \oplus x_1x_9x_{12} \\
& \oplus x_2x_3x_8 \oplus x_2x_3x_{12} \oplus x_2x_4x_8 \oplus x_2x_4x_{10} \oplus x_2x_4x_{11} \oplus x_2x_4x_{12} \\
& \oplus x_2x_7x_8 \oplus x_2x_7x_{12} \oplus x_2x_8x_{10} \oplus x_2x_8x_{11} \oplus x_2x_9x_{10} \oplus x_2x_9x_{11} \\
& \oplus x_2x_{10}x_{12} \oplus x_2x_{11}x_{12} \oplus x_3x_4x_8 \oplus x_3x_4x_{12} \oplus x_3x_8x_9 \oplus x_3x_9x_{12} \\
& \oplus x_4x_7x_8 \oplus x_4x_7x_{12} \oplus x_4x_8x_9 \oplus x_4x_9x_{12} \oplus x_5x_6x_8 \oplus x_5x_6x_{10} \\
& \oplus x_5x_6x_{11} \oplus x_5x_6x_{12} \oplus x_6x_8x_{10} \oplus x_6x_8x_{11} \oplus x_6x_{10}x_{12} \oplus x_6x_{11}x_{12} \\
& \oplus x_7x_8x_9 \oplus x_7x_9x_{12} \oplus x_8x_9x_{10} \oplus x_8x_9x_{11} \oplus x_9x_{10}x_{12} \oplus x_9x_{11}x_{12} \\
& \oplus x_0x_5x_8x_{10} \oplus x_0x_5x_8x_{11} \oplus x_0x_5x_{10}x_{12} \oplus x_0x_5x_{11}x_{12} \oplus x_1x_2x_3x_8 \\
& \oplus x_1x_2x_3x_{12} \oplus x_1x_2x_7x_8 \oplus x_1x_2x_7x_{12} \oplus x_1x_3x_5x_8 \oplus x_1x_3x_5x_{12} \\
& \oplus x_1x_3x_8x_9 \oplus x_1x_3x_9x_{12} \oplus x_1x_4x_8x_{10} \oplus x_1x_4x_8x_{11} \oplus x_1x_4x_{10}x_{12} \\
& \oplus x_1x_4x_{11}x_{12} \oplus x_1x_5x_7x_8 \oplus x_1x_5x_7x_{12} \oplus x_1x_7x_8x_9 \oplus x_1x_7x_9x_{12} \\
& \oplus x_1x_8x_9x_{10} \oplus x_1x_8x_9x_{11} \oplus x_1x_9x_{10}x_{12} \oplus x_1x_9x_{11}x_{12} \oplus x_2x_3x_4x_8 \\
& \oplus x_2x_3x_4x_{12} \oplus x_2x_3x_5x_8 \oplus x_2x_3x_5x_{12} \oplus x_2x_4x_7x_8 \oplus x_2x_4x_7x_{12} \\
& \oplus x_2x_4x_8x_{10} \oplus x_2x_4x_8x_{11} \oplus x_2x_4x_{10}x_{12} \oplus x_2x_4x_{11}x_{12} \oplus x_2x_5x_7x_8 \\
& \oplus x_2x_5x_7x_{12} \oplus x_2x_8x_9x_{10} \oplus x_2x_8x_9x_{11} \oplus x_2x_9x_{10}x_{12} \oplus x_2x_9x_{11}x_{12} \\
& \oplus x_3x_4x_8x_9 \oplus x_3x_4x_9x_{12} \oplus x_4x_7x_8x_9 \oplus x_4x_7x_9x_{12} \oplus x_5x_6x_8x_{10} \\
& \oplus x_5x_6x_8x_{11} \oplus x_5x_6x_{10}x_{12} \oplus x_5x_6x_{11}x_{12}.
\end{aligned}
$$

The Boolean function F has the following properties [GGK06a]:

- It is balanced.
- Algebraic degree $\deg(F) = 4$.
- Correlation immune of order $m = 8$ (8-resilient).
- Nonlinearity $nl(F) = 3584$.
- Algebraic immunity $AI(F) = 4$.

It is also claimed that F can be implemented with gate count 68, which can be considered as quite small considering the large ANF. Achterbahn-80 can be constructed by removing the registers R_0 and R_{12} i.e., letting $x_0 = 0$ and $x_{12} = 0$ at all time instances. Thus, it consists of $n = 11$ registers. The keystream bit z for Achterbahn-80 is given by the Boolean function

$$z = G(x_1, \ldots, x_{11}) = F(0, x_1, \ldots, x_{11}, 0), \tag{7.3}$$

which has ANF

$$
\begin{aligned}
G(x_1,\ldots,x_{11}) = \; & x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_2x_{10} \oplus x_2x_{11} \oplus x_4x_8 \\
& \oplus x_5x_6 \oplus x_6x_8 \oplus x_6x_{10} \oplus x_6x_{11} \oplus x_7x_8 \oplus x_8x_9 \oplus x_8x_{10} \oplus x_9x_{10} \\
& \oplus x_9x_{11} \oplus x_1x_2x_8 \oplus x_1x_4x_{10} \oplus x_1x_4x_{11} \oplus x_1x_8x_9 \oplus x_1x_9x_{10} \\
& \oplus x_1x_9x_{11} \oplus x_2x_3x_8 \oplus x_2x_4x_8 \oplus x_2x_4x_{10} \oplus x_2x_4x_{11} \oplus x_2x_7x_8 \\
& \oplus x_2x_8x_{10} \oplus x_2x_8x_{11} \oplus x_2x_9x_{10} \oplus x_2x_9x_{11} \oplus x_3x_4x_8 \oplus x_3x_8x_9 \\
& \oplus x_4x_7x_8 \oplus x_4x_8x_9 \oplus x_5x_6x_8 \oplus x_5x_6x_{10} \oplus x_5x_6x_{11} \oplus x_6x_8x_{10} \\
& \oplus x_6x_8x_{11} \oplus x_7x_8x_9 \oplus x_8x_9x_{10} \oplus x_8x_9x_{11} \oplus x_1x_2x_3x_8 \oplus x_1x_2x_7x_8 \\
& \oplus x_1x_3x_5x_8 \oplus x_1x_3x_8x_9 \oplus x_1x_4x_8x_{10} \oplus x_1x_4x_8x_{11} \oplus x_1x_5x_7x_8 \\
& \oplus x_1x_7x_8x_9 \oplus x_1x_8x_9x_{10} \oplus x_1x_8x_9x_{11} \oplus x_2x_3x_4x_8 \oplus x_2x_3x_5x_8 \\
& \oplus x_2x_4x_7x_8 \oplus x_2x_4x_8x_{10} \oplus x_2x_4x_8x_{11} \oplus x_2x_5x_7x_8 \oplus x_2x_8x_9x_{10} \\
& \oplus x_2x_8x_9x_{11} \oplus x_3x_4x_8x_9 \oplus x_4x_7x_8x_9 \oplus x_5x_6x_8x_{10} \oplus x_5x_6x_8x_{11}.
\end{aligned}
$$

The Boolean function G has the following properties [GGK06a]:

- It is balanced.
- Algebraic degree $\deg(G) = 4$.
- Correlation immune of order $m = 6$ (6-resilient).
- Nonlinearity $nl(G) = 896$.
- Algebraic immunity $AI(G) = 4$.

The first two versions of Achterbahn came in one full and one reduced variant. In the reduced variant the inputs to the Boolean combining function were taken simply as the value of the rightmost cell in Fig. 7.1. In the full variant the input was a key dependent linear combination of a few shift register cells. In Achterbahn-128/80, the method used in the previous reduced variants has been adopted. The input to the Boolean function is just taken from a fixed shift register cell. However, the bit x_i taken from register R_i is not taken from the rightmost cell. Instead it is taken from position 16.

7.2.3 Initialization

The initialization of Achterbahn has been slightly changed in Achterbahn-128/80 compared to the previous two versions. Only the new initialization procedure will be explained here. Let K denote the key, $|K|$ the key size in bits and $k_0 \ldots k_{|K|-1}$ be the individual bits in the key. Also, let IV denote the initialization vector and $|IV|$ its size in bits. The initialization is divided into 6 steps as follows.

(*i*) Each register R_i is loaded in parallel with the first L_i bits of the key, i.e., $k_0 \ldots k_{L_i-1}$.

(*ii*) Each register R_i is updated $|K| - L_i$ times, xoring the remaining $|K| - L_i$ key bits with the feedback bit to the register.

(*iii*) Each register R_i is updated $|IV|$ times, xoring the $|IV|$ bits of the IV with the feedback bit to the register.

(*iv*) The output of the registers are compressed into one bit by the Boolean combining function and this bit is fed back and xored with the feedback bit to each register. This is done 32 times.

(*v*) The least significant bit of each NFSR is set to 1. This prevents the NFSRs to be initialized with all zeros.

(*vi*) Warm up phase. The registers are clocked 64 times.

Because of step (v) above, the entropy of the content of register R_i at the beginning of step (vi) is only $L_i - 1$. No unknowns (key bits) are used in phase (vi) and thus, exhaustively searching register R_i requires 2^{L_i-1} tries. We now move on to an analysis of the Achterbahn design.

7.3 Analysis of Achterbahn

This section will give an analysis of the security of the Achterbahn stream cipher. We show several ideas how to cryptanalyze the construction. The analysis does not use the fact that the shift registers have nonlinear feedback. The main property that is used in the analysis is the fact that all registers are very short and thus, have a very short period. This will be shown to be the most serious weakness of the construction. Consequently, the analysis given here will be valid also for a nonlinear combiner using short registers with linear feedback.

The proposed attacks on Achterbahn consists of two steps, namely

(*i*) Recover the state of a subset of the NFSRs. This step is equivalent to a distinguishing attack.

(*ii*) Recover the key K. For Achterbahn, the complexity of this step is, for all known attacks, much smaller than the complexity of the first step.

7.3.1 Attacking the Achterbahn Family of Stream Ciphers

A possible approach to attack Achterbahn is to consider correlations between input and output bits in the Boolean combining function. Recalling Section 2.3.4, let the resiliency of the Boolean function be m. Then there is a linear relation between at least $m + 1$ input bits and the output bit that is unbalanced, i.e.,

$$\Pr\left(\bigoplus_{i\in\mathcal{A}} x_i(t) = z(t)\right) = \frac{1}{2}(1+\varepsilon) \quad 0 < |\varepsilon| \leq 1, \tag{7.4}$$

for some subset $\mathcal{A}$, of size $|\mathcal{A}| > m$, of registers. If we can find a parity check equation for the sequence produced by $\bigoplus_{i\in\mathcal{A}} x_i(t)$, then, according to (2.62) we can distinguish the keystream from random considering about

$$N \approx \frac{1}{\varepsilon_{tot}^2} \tag{7.5}$$

keystream bits, where ε_{tot} will depend on ε and the number of terms in the parity check equation. Since the shift registers R_i used in Achterbahn are small it is very easy to find parity check equations. The sequence generated by register R_i has characteristic polynomial $1 - x^{T_i}$ resulting in the parity check equation

$$x_i(t) = x_i(t + T_i). \tag{7.6}$$

More generally, the sequence produced by $\bigoplus_{i\in\mathcal{A}} x_i(t)$ has characteristic polynomial

$$\prod_{i\in\mathcal{A}} \left(1 - x^{T_i}\right). \tag{7.7}$$

Clearly, the number of terms in the corresponding parity check equation is $2^{|\mathcal{A}|}$. Assuming that all variables in the parity check equation are independent we can use the piling-up lemma (2.58) and write the total bias as

$$\varepsilon_{tot} = \varepsilon^{2^{|\mathcal{A}|}} \tag{7.8}$$

and the number of samples needed in the distinguisher is about

$$N \approx \frac{1}{\varepsilon^{2^{(|\mathcal{A}|+1)}}}. \tag{7.9}$$

The distance between the first and the last keystream bit in each sample is $\sum_{i\in\mathcal{A}} T_i$, i.e., the largest exponent in (7.7).

The resiliency of the Boolean combining function has increased for every new version of Achterbahn, since the attacks have showed that the resiliency has not been enough. If the resiliency m of the function increases, it is clear that the number of required samples increases since $|\mathcal{A}| > m$. To compensate for this, we present three approaches.

- *Guess the state of one or several registers.* This will automatically give one or several internal states and the attack is not just a distinguishing attack, it will give the attacker the possibility to recover the key. Guessing λ registers will decrease the number of factors in the product (7.7) by λ. Thus, the number of terms in the parity check equation will decrease by a factor 2^λ. Now we are left with only $2^{|\mathcal{A}|-\lambda}$ terms in the parity check equation and according to (2.58) the total bias is

$$\varepsilon_{tot} = \varepsilon^{2^{|\mathcal{A}|-\lambda}} \tag{7.10}$$

if the variables are independent. On the other hand, since we are guessing the internal state of one or several registers, we have to find the biased sequence among a set of approximately $2^{\sum_{i\in\Gamma}(L_i-1)}$ sequences, where Γ is the set of registers which internal state is guessed. The number of samples needed in the distinguisher is now, according to (2.61), given by

$$N \approx \frac{\sum_{i\in\Gamma}(L_i-1)\cdot 2\ln 2}{\varepsilon_{tot}^2}. \tag{7.11}$$

- *Consider nonlinear approximations of the combining function.* The characteristic polynomial of the sequence produced by $x_i x_j$ is given as $1-x^{T_iT_j}$ and the characteristic polynomial of the sequence produced by $x_i x_j x_k$ is $1-x^{T_iT_jT_k}$. Using a quadratic or cubic approximation will reduce the amount of required samples since the number of terms in the parity check equation will be reduced. On the other hand, the distance between the first and the last keystream bit in each sample will increase and give an upper limit on the degree of the nonlinear approximation.

- *Jump T_i steps for each new sample*. A third approach to compensate for high resiliency is to not consider samples given by consecutive keystream bits. Instead, to remove the dependency of the linear term x_i in the approximation, we can jump forward T_i steps every time we take a new sample. The contribution to the Boolean combining function from register R_i will then be constant, though we do not know if this constant is 0 or 1. This will double the error probability. However, it is not crucial for our attack that only the correct guess will report a detected bias. It is possible to have a few probable initial states and recover the corresponding key from each state and then decide which key is correct. This will not influence the attack complexity. Taking every T_ith sample will increase the amount of keystream needed in the attack but it will decrease the computational complexity since the state of register R_i does not have to be guessed.

The attacks on Achterbahn-128/80 given in this chapter will use all three approaches given above.

7.3.2 Summary of Attack Procedure

An example of the procedure is given as follows. Assume that we have a biased relation in Achterbahn-80 of the form

$$Q(x_1,\ldots,x_{11}) = x_a \oplus x_b \oplus x_c \oplus x_d x_e \oplus x_f x_g. \tag{7.12}$$

Instead of using a parity check equation for all 5 terms, we only use the parity check equation for the terms $x_d x_e \oplus x_f x_g$. Thus, we know that

$$\begin{aligned} 0 \;=\; & x_d(t)x_e(t) \oplus x_d(t+T_dT_e)x_e(t+T_dT_e) \oplus x_d(t+T_fT_g)x_e(t+T_fT_g) \\ & \oplus x_d(t+T_dT_e+T_fT_g)x_e(t+T_dT_e+T_fT_g) \\ & \oplus x_f(t)x_g(t) \oplus x_f(t+T_dT_e)x_g(t+T_dT_e) \oplus x_f(t+T_fT_g)x_g(t+T_fT_g) \\ & \oplus x_f(t+T_dT_e+T_fT_g)x_g(t+T_dT_e+T_fT_g), \end{aligned} \tag{7.13}$$

and we can write

$$\begin{aligned} & x_a(t) \oplus x_a(t+T_dT_e) \oplus x_a(t+T_fT_g) \oplus x_a(t+T_dT_e+T_fT_g) \\ \oplus \;& x_b(t) \oplus x_b(t+T_dT_e) \oplus x_b(t+T_fT_g) \oplus x_b(t+T_dT_e+T_fT_g) \\ \oplus \;& x_c(t) \oplus x_c(t+T_dT_e) \oplus x_c(t+T_fT_g) \oplus x_c(t+T_dT_e+T_fT_g) \\ \overset{p}{=} \;& z(t) \oplus z(t+T_dT_e) \oplus z(t+T_fT_g) \oplus z(t+T_dT_e+T_fT_g), \end{aligned} \tag{7.14}$$

with $p = 0.5(1+\varepsilon_{tot})$. The distance between the first and the last keystream bit in each sample is $T_dT_e + T_fT_g$. When the correct initial states of R_a, R_b and R_c are guessed we detect the bias. Instead of exhaustively searching the states of all registers we jump T_c steps for each new sample. Then we can write

$$\begin{aligned} & x_a(tT_c) \oplus x_a(tT_c+T_dT_e) \oplus x_a(tT_c+T_fT_g) \oplus x_a(tT_c+T_dT_e+T_fT_g) \\ \oplus \;& x_b(tT_c) \oplus x_b(tT_c+T_dT_e) \oplus x_b(tT_c+T_fT_g) \oplus x_b(tT_c+T_dT_e+T_fT_g) \\ \oplus \;& \gamma(t) \\ \overset{p}{=} \;& z(tT_c) \oplus z(tT_c+T_dT_e) \oplus z(tT_c+T_fT_g) \oplus z(tT_c+T_dT_e+T_fT_g), \end{aligned} \tag{7.15}$$

where

$$\gamma(t) = x_c(tT_c) \oplus x_c(tT_c+T_dT_e) \oplus x_c(tT_c+T_fT_g) \oplus x_c(tT_c+T_dT_e+T_fT_g) \tag{7.16}$$

is constant $\forall t$ and $\gamma(t) \in \{0, 1\}$.

7.4 The Sum of Dependent Variables

When the piling-up lemma was used in the previous section it was assumed that all variables are independent. However, if the variables are *dependent* the piling-up lemma is not applicable[2]. This is demonstrated by the two examples given in Fig. 7.2. In linear cryptanalysis, dependency between variables is often assumed to be small and the piling-up lemma is used.

[2]The author would like to acknowledge Lennart Brynielsson for pointing out an error in a previous version of this section.

X_1	X_2	$\Pr(X_1, X_2)$
0	0	1/2
0	1	1/4
1	0	1/4
1	1	0

$$\begin{aligned}\Pr(X_1 = 0) &= 0.5(1 + 0.5) \quad (\varepsilon = 0.5)\\ \Pr(X_2 = 0) &= 0.5(1 + 0.5) \quad (\varepsilon = 0.5)\\ \Pr(X_1 \oplus X_2 = 0) &= 0.5(1 + 0) \quad (\varepsilon = 0)\end{aligned}$$

X_1	X_2	$\Pr(X_1, X_2)$
0	0	1/2
0	1	1/4
1	0	0
1	1	1/4

$$\begin{aligned}\Pr(X_1 = 0) &= 0.5(1 + 0.5) \quad (\varepsilon = 0.5)\\ \Pr(X_2 = 0) &= 0.5(1 + 0) \quad (\varepsilon = 0)\\ \Pr(X_1 \oplus X_2 = 0) &= 0.5(1 + 0.5) \quad (\varepsilon = 0.5)\end{aligned}$$

Figure 7.2: Examples showing that the piling-up lemma is not applicable if the variables are dependent.

We now move on to determine the *real* bias of the samples used in the cryptanalysis of Achterbahn-128/80. In Achterbahn-80, if we consider a quadratic approximation of the form

$$Q(x_1, \ldots, x_{11}) = x_a \oplus x_b \oplus x_c \oplus x_d x_e \oplus x_f x_g. \tag{7.17}$$

This approximation has bias $\varepsilon = 2^{-5}$ (or $\varepsilon = 0$ but this case is not interesting here). According to the piling-up lemma, the bias of (7.15) is $\varepsilon_{tot} = (2^{-5})^4 = 2^{-20}$. However, it is easy to see that the variables are quite dependent, e.g., for the first two terms, the variables x_d and x_e will always be the same. Instead of $4n = 44$ variables, there is only $11 + 9 + 9 + 7 = 36$ different variables used in (7.15). Computing the exact bias by exhaustively searching all 2^{36} possibilities we can see that it will not be $\varepsilon_{tot} = 2^{-20}$ but instead $\varepsilon_{tot} = 2^{-12}$. This difference has a huge effect on the attack complexities. Instead of 2^{40} samples we only need about 2^{24} samples to detect this bias.

Looking at cubic approximations of Achterbahn-128 of the form

$$C(x_0, \ldots, x_{12}) = x_a \oplus x_b \oplus x_c \oplus x_d \oplus x_e x_f \oplus x_g x_h x_i, \tag{7.18}$$

this will hold with bias $\varepsilon = 2^{-6}$ (or $\varepsilon = 0$). The sequence generated by the nonlinear terms will have a parity check equation given by

$$z(t) \oplus z(t + T_e T_f) \oplus z(t + T_g T_h T_i) \oplus z(t + T_e T_f + T_g T_h T_i). \tag{7.19}$$

Using the piling-up lemma, the total bias will be $\varepsilon_{tot} = (2^{-6})^4 = 2^{-24}$. However, the real bias is also in this case 2^{-12} because of the dependency between variables. For quadratic approximations of Achterbahn-128 the corresponding bias is $\varepsilon_{tot} = 2^{-12}$, the same as for the 80-bit variant, instead of $\varepsilon_{tot} = 2^{-20}$ as given by the piling-up lemma.

7.5 Attack on Achterbahn-80

The Boolean combining function used in Achterbahn-80 has resiliency 6 so every biased approximation must consider at least 7 variables. In this sec-

tion, the attack parameters for a successful attack on Achterbahn-80 will be given.

7.5.1 Generalization of the Attack Using Quadratic Approximations

The quadratic approximations used in the attack on Achterbahn-80 have the form

$$Q(x_1, \ldots, x_{11}) = x_a \oplus x_b \oplus x_c \oplus x_d x_e \oplus x_f x_g, \tag{7.20}$$

where all indices are distinct. Assume that, in the attack, the initial states of registers R_a and R_b are guessed, i.e., we perform an exhaustive search over all possible initial states of these two registers. Further, every T_cth sample is taken. This corresponds to the procedure given in Section 7.3.2. The total amount of keystream needed is then

$$T_d T_e + T_f T_g + \frac{(L_a + L_b - 2) \cdot 2 \ln 2}{\varepsilon_{tot}^2} T_c. \tag{7.21}$$

The computational complexity of the attack is given as

$$2^{L_a + L_b - 2} \cdot \frac{(L_a + L_b - 2) \cdot 2 \ln 2}{\varepsilon_{tot}^2}. \tag{7.22}$$

For all approximations of the form (7.20) that we use, we have $|\varepsilon_{tot}| = 2^{-12}$.

7.5.2 Attack Complexities for Achterbahn-80

We search through all approximations of the form (7.20). Table 7.1 shows the amount of keystream needed and the corresponding time complexity of an attack using a certain approximation. The best time complexity, $2^{72.90}$, is achieved if $2^{58.24}$ keystream bits are available. The attack also allows a tradeoff between the amount of required keystream and the time complexity. Having less keystream bits results in higher complexity. In the table, we only consider approximations resulting in lower computational complexity than 2^{79}.

7.6 Attack on Achterbahn-128

The Boolean combining function used in Achterbahn-128 has resiliency 8 so every biased approximation must consider at least 9 variables. In this section we give the attacks on Achterbahn-128 using quadratic and cubic approximations.

Approximation	Keystream	Time
$x_1 \oplus x_2 \oplus x_7 \oplus x_3x_{10} \oplus x_4x_9$	$2^{58.24}$	$2^{72.90}$
$x_1 \oplus x_3 \oplus x_5 \oplus x_4x_{10} \oplus x_6x_7$	$2^{57.29}$	$2^{73.93}$
$x_1 \oplus x_4 \oplus x_5 \oplus x_3x_{10} \oplus x_6x_7$	$2^{56.98}$	$2^{74.96}$
$x_1 \oplus x_5 \oplus x_4 \oplus x_3x_{10} \oplus x_6x_7$	$2^{56.58}$	$2^{75.99}$
$x_1 \oplus x_6 \oplus x_4 \oplus x_3x_{10} \oplus x_5x_7$	$2^{56.33}$	$2^{77.03}$
$x_1 \oplus x_6 \oplus x_3 \oplus x_2x_{10} \oplus x_4x_9$	$2^{56.01}$	$2^{78.06}$

Table 7.1: Attack complexities for Achterbahn-80.

7.6.1 Generalization of the Attack Using Quadratic Approximations

The quadratic approximation used in cryptanalysis of Achterbahn-128 can be written as

$$Q(x_0, \ldots, x_{12}) = x_a \oplus x_b \oplus x_c \oplus x_d \oplus x_e \oplus x_f x_g \oplus x_h x_i. \tag{7.23}$$

Exhaustively searching registers R_a, R_b, R_c and R_d and taking every T_eth sample will result in an attack requiring

$$T_f T_g + T_h T_i + \frac{(L_a + L_b + L_c + L_d - 4) \cdot 2 \ln 2}{\varepsilon_{tot}^2} T_e \tag{7.24}$$

keystream bits and the computational complexity is

$$2^{L_a+L_b+L_c+L_d-4} \cdot \frac{(L_a + L_b + L_c + L_d - 4) \cdot 2 \ln 2}{\varepsilon_{tot}^2}, \tag{7.25}$$

with $|\varepsilon_{tot}| = 2^{-12}$.

7.6.2 Generalization of the Attack Using Cubic Approximations

For Achterbahn-128 it is possible, not only to look at quadratic approximations, but also to consider cubic approximations. The cubic approximations in our attack will have the form

$$C(x_0, \ldots, x_{12}) = x_a \oplus x_b \oplus x_c \oplus x_d \oplus x_e x_f \oplus x_g x_h x_i. \tag{7.26}$$

It is not possible to find a product of the periods of 3 registers in Achterbahn-128 less than the maximum frame length, 2^{64}. However, since periods of all registers are not relatively prime, it is possible to find 3 registers for which the least common multiple (lcm) of the periods is less than 2^{64}. This will happen for

$$\begin{aligned} lcm(T_0, T_1, T_3) &= 2^{62.60} \\ lcm(T_0, T_1, T_7) &= 2^{62.42} \\ lcm(T_0, T_1, T_{12}) &= 2^{62.19} \\ lcm(T_1, T_3, T_{12}) &= 2^{63.60}. \end{aligned}$$

Hence, for the attack to be successful the cubic term in the approximation must be one of $x_0x_1x_3$, $x_0x_1x_7$, $x_0x_1x_{12}$ and $x_1x_3x_{12}$. In the attack, the registers R_a, R_b and R_c are exhaustively searched and we take every T_dth sample. The amount of keystream needed in the attack is

$$T_eT_f + lcm(T_g, T_h, T_i) + \frac{(L_a + L_b + L_c - 3) \cdot 2 \ln 2}{\varepsilon_{tot}^2} T_d \tag{7.27}$$

keystream bits and the computational complexity is

$$2^{L_a+L_b+L_c-3} \cdot \frac{(L_a + L_b + L_c - 3) \cdot 2 \ln 2}{\varepsilon_{tot}^2}, \tag{7.28}$$

with $|\varepsilon_{tot}| = 2^{-12}$.

7.6.3 Attack Complexities for Achterbahn-128

In Table 7.2 the attack complexities we have found for Achterbahn-128 is shown. The best computational complexity is achieved using the cubic approximations, but if less keystream is available, the quadratic approximations can be used as well.

7.7 Recovering the Key

The figures given in Tables 7.1 and 7.2 are the complexities for recovering a subset of the states of the NFSRs. This would be equivalent to a distinguishing attack on the cipher. In this section we show that the key also can be recovered and the complexity for this step is smaller than for the distinguishing attack. Thus, the figures in the tables are valid also for a key recovery attack. Because of the introduction of step (iv) in the initialization process, it is not possible to recover the key without knowing the state of all registers. In the attacks described in Section 7.5 and 7.6 we only recover a subset of the states. The output bit of each register is taken as a bit in the middle of the register and not the last bit. This makes the initialization process invertible and if we know all initial states it will be easy to recover the key. After recovering a subset of the states in the first step, it is possible, in a second step, to recover all other states with much less keystream and complexity than in the first step. As an example, when the approximation

$$x_1 \oplus x_2 \oplus x_7 \oplus x_3x_{10} \oplus x_4x_9 \tag{7.29}$$

Approximation	Keystream	Time
$x_0 \oplus x_2 \oplus x_4 \oplus x_9 \oplus x_7x_{10} \oplus x_1x_3x_{12}$	$2^{63.81}$	$2^{96.52}$
$x_2 \oplus x_3 \oplus x_4 \oplus x_9 \oplus x_7x_{10} \oplus x_0x_1x_{12}$	$2^{62.71}$	$2^{99.58}$
$x_3 \oplus x_4 \oplus x_5 \oplus x_7 \oplus x_6x_{10} \oplus x_0x_1x_{12}$	$2^{62.38}$	$2^{102.64}$
$x_3 \oplus x_4 \oplus x_6 \oplus x_7 \oplus x_5x_{10} \oplus x_0x_1x_{12}$	$2^{62.35}$	$2^{103.66}$
$x_3 \oplus x_4 \oplus x_7 \oplus x_6 \oplus x_5x_{10} \oplus x_0x_1x_{12}$	$2^{62.29}$	$2^{104.68}$
$x_3 \oplus x_5 \oplus x_7 \oplus x_6 \oplus x_4x_{10} \oplus x_0x_1x_{12}$	$2^{62.27}$	$2^{105.70}$
$\vdots$	$\vdots$	$\vdots$
$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_7 \oplus x_4x_{10} \oplus x_8x_9$	$2^{60.04}$	$2^{116.90}$
$x_0 \oplus x_1 \oplus x_2 \oplus x_4 \oplus x_7 \oplus x_3x_{10} \oplus x_8x_9$	$2^{60.00}$	$2^{117.91}$
$x_0 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_6 \oplus x_5x_{10} \oplus x_7x_8$	$2^{58.97}$	$2^{118.93}$
$x_0 \oplus x_1 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_4x_{10} \oplus x_7x_8$	$2^{58.78}$	$2^{119.95}$
$x_0 \oplus x_1 \oplus x_3 \oplus x_6 \oplus x_5 \oplus x_4x_{10} \oplus x_7x_8$	$2^{58.31}$	$2^{120.96}$
$x_0 \oplus x_1 \oplus x_3 \oplus x_7 \oplus x_5 \oplus x_4x_{10} \oplus x_6x_8$	$2^{57.99}$	$2^{121.98}$
$x_0 \oplus x_1 \oplus x_4 \oplus x_7 \oplus x_5 \oplus x_3x_{10} \oplus x_6x_8$	$2^{57.80}$	$2^{122.99}$

Table 7.2: Attack complexities for Achterbahn-128.

is used for Achterbahn-80, x_1 and x_2 are recovered in the first step. When these are known it is easy to recover x_7 by just guessing the state of this. In the next step we can use the linear approximation

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_7 \oplus x_9 \oplus x_{10}. \tag{7.30}$$

This approximation has bias $\varepsilon = 2^{-3}$ and we can use a characteristic polynomial for two of the 4 unknown variables, decimate the sequence for one unknown and guess the last. Doing this for all 4 unknowns will give us in total 7 recovered registers. The last 4 can be found using other linear approximations. A similar method can be used to recover all states of Achterbahn-128.

When the initial states of all registers are recovered, they are clocked backwards until the beginning of step 3 in the initialization step. To make it simple, the last $80 - L_1$ bits of the key are guessed. All registers are clocked backwards according to the guessed key bits until the beginning of step 2 of the initialization is reached. When the first L_1 bits of the states are the same for all registers, the correct key has been found. This last step has computational complexity 2^{58} and is thus not a bottleneck. For the 128-bit variant we can use the time-memory tradeoff. Divide the key into two parts K_1 and K_2

with sizes $|K_1|$ and $|K_2| = 128 - |K_1|$. Do step (i) and (ii) in the initialization process using the first $|K_1|$ bits and save the resulting states in memory. Then, do step (ii) backwards for the last $|K_2|$ bits. When the resulting state collides with a saved state, the correct key has been found. This will require $2^{|K_1|}$ in memory and $2^{|K_1|} + 2^{|K_2|}$ in time complexity. Picking e.g., $|K_1| = 40$ and $|K_2| = 88$ will make this complexity faster than recovering the initial states.

7.8 Further Improvements

In [NP07a], further improvements and considerations regarding the cryptanalysis ideas given in Section 7.3.1 was presented. It was observed that the parity check equations used in order to remove the dependency of input variables did not exactly have to correspond to the approximation of the nonlinear function. In Achterbahn-80, when considering samples of the form

$$z(t) \oplus z(t + T_d T_e) \oplus z(t + T_f T_g) \oplus z(t + T_d T_e + T_f T_g) \tag{7.31}$$

we eliminate the dependency of x_d, x_e, x_f and x_g. These variables can then occur in any form in the approximation. In [CT00] it was shown that the highest bias is achieved by a linear approximation. Thus it is possible to instead look at approximations of the form

$$L(x_1, \ldots, x_{11}) = x_a \oplus x_b \oplus x_c \oplus x_d \oplus x_e \oplus x_f \oplus x_g, \tag{7.32}$$

which will give a higher bias than any of the quadratic approximations used in Section 7.5. In [GG07, Theorem 1], it was shown that the piling-up lemma can be used if the approximations are affine and if the number of terms is $m + 1$. Thus, the bias achieved in Section 7.4 and in [NP07a] are the same.

The paper [NP07a] also improved the computational complexity of the attacks by presenting a faster algorithm to exhaustively search a subset of the registers. This algorithm will not be given here and we refer to [NP07a] for details. The improved algorithm reduced the time complexity of the attack to 2^{55} for Achterbahn-80 and $2^{80.58}$ for Achterbahn-128.

7.9 History of Achterbahn, Part II

In an effort to save Achterbahn, the designers proposed new keystream limitations to Achterbahn-128/80. At SASC 2007 the new keystream limitations was defined to be 2^{52} and 2^{56} for Achterbahn-80 and Achterbahn-128 respectively. The cipher using these new limitations was broken on the

rump session of the same workshop. These results are also given in [NP07a] and [NP07b]. Again, the designers changed the keystream limitation. This was done in [GG07] and the new limit is 2^{44} for both variants.

7.10 Summary

This chapter showed a key recovery attack on the stream cipher Achterbahn-128/80. Both the 80-bit and the 128-bit variants of the cipher are vulnerable to the attack. The attack takes advantage of the short periods of the nonlinear feedback shift registers used in the cipher. Using the short periods, several nontrivial and novel techniques are combined in order to decrease the time complexity and the amount of keystream needed in order to recover a subset of the initial states of the NFSRs. Knowing these states it was relatively easy to recover the secret key used in the cipher. The Achterbahn stream cipher reached the second phase of eSTREAM. Due to the findings given in this section together with the attacks presented in [NP07a], the cipher was not considered interesting or secure enough to qualify for the third phase of eSTREAM. At the time of writing there is no attack on Achterbahn respecting the latest keystream restriction 2^{44} bits. However, a cipher requiring such a low limitation should not be considered for usage in any environment unless it has other very important advantages. Though, this does not seem to be the case for Achterbahn.

8

The Grain Family of Stream Ciphers

The previous five chapters have considered cryptanalysis of stream ciphers. This chapter will describe a new stream cipher primitive. When designing a cryptographic primitive there are many different properties that have to be addressed. These include e.g., speed and security. Comparing several ciphers, it is likely that one is faster on a 32-bit processor, another is faster on an 8 bit processor and yet another one is faster in hardware. The simplicity of the design is another factor that has to be taken into account. While the software implementation can be very simple, the hardware implementation might be quite complex.

There is a need for cryptographic primitives that have very low hardware complexity. A radio-frequency identification (RFID) tag is a typical example of a product where the amount of memory and power is very limited. These are microchips capable of transmitting an identifying sequence upon a request from a reader. Forging an RFID tag can have devastating consequences if the tag is used e.g., in electronic payments and hence, there is a need for cryptographic primitives implemented in these tags. Today, a hardware implementation of e.g., AES on an RFID tag is not feasible due to the large number of gates needed. The Grain family of stream ciphers, introduced in this chapter, is a stream cipher family that is designed to be very easy and small to implement in hardware.

Several recent LFSR based stream cipher proposals, see e.g., [EJ02,HR03] and their predecessors, are based on word oriented LFSRs. This allows them to be efficiently implemented in software but it also allows them to increase the throughput since words instead of bits are output. In hardware, a word oriented cipher is likely to be more complex than a bit oriented one. In the Grain ciphers, this issue has been addressed by basing the design on

bit oriented shift registers with the extra feature of allowing an increase in speed at the expense of more hardware. The user can decide the speed of the cipher depending on the amount of hardware available. This property is not explicitly found in most other stream ciphers.

The proposed designs, denoted Grain and Grain-128, are bit oriented synchronous stream ciphers. The designs are based on two shift registers, one with linear feedback (LFSR) and one with nonlinear feedback (NFSR). The LFSR guarantees a minimum period for the keystream and it also provides balancedness in the output. The NFSR, together with a nonlinear output function introduces nonlinearity to the cipher. The input to the NFSR is masked with the output of the LFSR so that the state of the NFSR is balanced. Hence, we use the notation NFSR even though this is actually a filter. What is known about cycle structures of nonlinear feedback shift registers cannot immediately be applied here.

Grain is, like Pomaranch discussed in Chapter 6 and Achterbahn discussed in Chapter 7, a stream cipher submitted to the eSTREAM project [ECR]. The first, unpublished, version of the cipher is denoted version 0. This version was cryptanalyzed in [Max06, BGM06, KHK05]. The design of version 0 will not be given in this chapter but the attack will be discussed in Section 8.3.1. The two variants of Grain given in this chapter are denoted Grain Version 1 (or Grain V1) and Grain-128.

The chapter is based on [HJM06] and [HJMM06] and is organized as follows. Section 8.1 provides a detailed description of the Grain and Grain-128 designs. The possibility to easily increase the throughput is discussed in Section 8.2. The security of Grain is discussed in Section 8.3 together with a motivation for the different design parameters. In Section 8.4 the hardware performance of the cipher is considered. Section 8.5 summarizes the chapter.

8.1 Design specifications

This section specifies the details of the designs of both Grain and Grain-128. Both ciphers follow the same design principle. They consist of three main building blocks, namely an LFSR, an NFSR and an output function. The contents of the two shift registers represent the state of the cipher and their sizes are $|K|$ bits each, where K is the key. The content of the LFSR is denoted $S_t = s_t, s_{t+1}, \ldots, s_{t+|K|-1}$ and the content of the NFSR is denoted $B_t = b_t, b_{t+1}, \ldots, b_{t+|K|-1}$. The output function, denoted $H(B_t, S_t)$ consists of two parts. A nonlinear Boolean function $h(x)$ and a set of linear terms added to $h(x)$. The output of $H(B_t, S_t)$ is the keystream bit z_t. A general overview of the design is given in Fig. 8.1.

8.1.1 Grain - Design Parameters

The keysize of Grain is $|K| = 80$ bits and the cipher supports an IV of size $|IV| = 64$ bits. The feedback polynomial of the LFSR, denoted $f(x)$ is a primitive polynomial of degree 80. It is defined as

$$f(x) = 1 + x^{18} + x^{29} + x^{42} + x^{57} + x^{67} + x^{80}. \tag{8.1}$$

To remove any possible ambiguity we also define the update function of the LFSR as

$$s_{t+80} = s_{t+62} \oplus s_{t+51} \oplus s_{t+38} \oplus s_{t+23} \oplus s_{t+13} \oplus s_t. \tag{8.2}$$

The feedback polynomial of the NFSR, $g(x)$, is defined as

$$\begin{aligned} g(x) \quad = \quad & 1 + x^{18} + x^{20} + x^{28} + x^{35} + x^{43} + x^{47} + x^{52} + x^{59} + x^{66} + \\ & + x^{71} + x^{80} + x^{17}x^{20} + x^{43}x^{47} + x^{65}x^{71} + x^{20}x^{28}x^{35} + \\ & + x^{47}x^{52}x^{59} + x^{17}x^{35}x^{52}x^{71} + x^{20}x^{28}x^{43}x^{47} + x^{17}x^{20}x^{59}x^{65} + \\ & + x^{17}x^{20}x^{28}x^{35}x^{43} + x^{47}x^{52}x^{59}x^{65}x^{71} + x^{28}x^{35}x^{43}x^{47}x^{52}x^{59}. \end{aligned} \tag{8.3}$$

Again, to remove any possible ambiguity we also write the update function of the NFSR. Note that the bit s_t which is masked with the input is included in the update function below.

$$\begin{aligned} b_{t+80} \quad = \quad & s_t \oplus b_{t+62} \oplus b_{t+60} \oplus b_{t+52} \oplus b_{t+45} \oplus b_{t+37} \oplus b_{t+33} \oplus b_{t+28} \oplus \\ & \oplus b_{t+21} \oplus b_{t+14} \oplus b_{t+9} \oplus b_t \oplus b_{t+63}b_{t+60} \oplus b_{t+37}b_{t+33} \oplus \\ & \oplus b_{t+15}b_{t+9} \oplus b_{t+60}b_{t+52}b_{t+45} \oplus b_{t+33}b_{t+28}b_{t+21} \oplus \\ & \oplus b_{t+63}b_{t+45}b_{t+28}b_{t+9} \oplus b_{t+60}b_{t+52}b_{t+37}b_{t+33} \oplus \\ & \oplus b_{t+63}b_{t+60}b_{t+21}b_{t+15} \oplus b_{t+63}b_{t+60}b_{t+52}b_{t+45}b_{t+37} \oplus \\ & \oplus b_{t+33}b_{t+28}b_{t+21}b_{t+15}b_{t+9} \oplus b_{t+52}b_{t+45}b_{t+37}b_{t+33}b_{t+28}b_{t+21}. \end{aligned} \tag{8.4}$$

From the two registers, 5 variables are taken as input to a Boolean function, $h(x)$. This filter function is chosen to be balanced, correlation immune of the first order and has algebraic degree 3. The nonlinearity is the highest possible for these functions, namely 12. The function is defined as

$$\begin{aligned} & h(x) = h(x_0, x_1, \ldots, x_4) = \\ & = x_1 \oplus x_4 \oplus x_0x_3 \oplus x_2x_3 \oplus x_3x_4 \oplus x_0x_1x_2 \oplus x_0x_2x_3 \oplus x_0x_2x_4 \oplus x_1x_2x_4 \oplus x_2x_3x_4 \end{aligned} \tag{8.5}$$

where the variables x_0, x_1, x_2, x_3 and x_4 correspond to the tap positions s_{t+3}, s_{t+25}, s_{t+46}, s_{t+64} and b_{t+63} respectively. The output function $H(B_t, S_t)$ is given by

$$z_t = H(B_t, S_t) = \bigoplus_{j \in \mathcal{A}} b_{t+j} \oplus h(s_{t+3}, s_{t+25}, s_{t+46}, s_{t+64}, b_{t+63}) \tag{8.6}$$

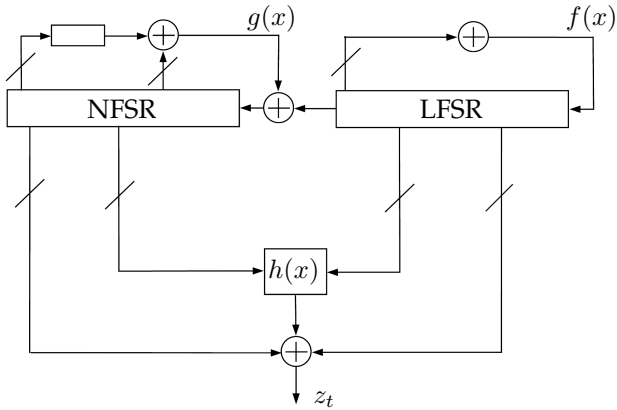

Figure 8.1: Overview of the different design blocks in the Grain family of stream ciphers.

where $\mathcal{A} = \{1, 2, 4, 10, 31, 43, 56\}$.

Cipher Initialization Before any keystream is generated the cipher must be initialized with the key and the IV. Let the bits of the key, K, be denoted $k_i,\ 0 \leq i \leq 79$ and the bits of the IV be denoted $IV_i,\ 0 \leq i \leq 63$. The initialization of the key is done as follows. First the NFSR and LFSR are loaded with key and IV bits as

$$\begin{cases} b_i = k_i, & 0 \leq i \leq 79 \\ s_i = IV_i, & 0 \leq i \leq 63 \end{cases} \tag{8.7}$$

The remaining bits of the LFSR are filled with ones, $s_i = 1,\ 64 \leq i \leq 79$. Then the cipher is clocked 160 times without producing any keystream. Instead the output function is fed back and xored with the input, both to the LFSR and to the NFSR, see Fig. 8.2.

8.1.2 Grain-128 - Design Parameters

Grain-128 supports a key size of $|K| = 128$ bits, as suggested by the name. The size of the IV is specified to be $|IV| = 96$ bits. The feedback polynomial of the LFSR, $f(x)$, is a primitive polynomial of degree 128. It is defined as

$$f(x) = 1 + x^{32} + x^{47} + x^{58} + x^{90} + x^{121} + x^{128}. \tag{8.8}$$

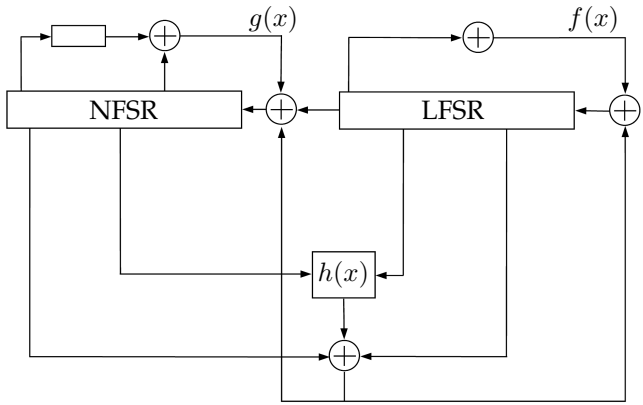

Figure 8.2: Overview of the key initialization.

To remove any possible ambiguity we also give the corresponding update function of the LFSR as

$$s_{t+128} = s_t \oplus s_{t+7} \oplus s_{t+38} \oplus s_{t+70} \oplus s_{t+81} \oplus s_{t+96}. \tag{8.9}$$

The nonlinear feedback polynomial of the NFSR, $g(x)$, is the sum of one linear and one bent function. It is defined as

$$\begin{aligned} g(x) \quad = \quad & 1 + x^{32} + x^{37} + x^{72} + x^{102} + x^{128} + x^{44}x^{60} + x^{61}x^{125} + x^{63}x^{67} + \\ & + x^{69}x^{101} + x^{80}x^{88} + x^{110}x^{111} + x^{115}x^{117}. \end{aligned} \tag{8.10}$$

Again, to remove any possible ambiguity we also write the corresponding update function of the NFSR. In the update function below, note that the bit s_t which is masked with the input to the NFSR is included, while omitted in the feedback polynomial.

$$\begin{aligned} b_{t+128} \quad = \quad & s_t \oplus b_t \oplus b_{t+26} \oplus b_{t+56} \oplus b_{t+91} \oplus b_{t+96} \oplus b_{t+3}b_{t+67} \oplus \\ & \oplus b_{t+11}b_{t+13} \oplus b_{t+17}b_{t+18} \oplus b_{t+27}b_{t+59} \oplus \\ & \oplus b_{t+40}b_{t+48} \oplus b_{t+61}b_{t+65} \oplus b_{t+68}b_{t+84}. \end{aligned} \tag{8.11}$$

From the state, nine variables are taken as input to a Boolean function, $h(x)$. Two inputs to $h(x)$ are taken from the NFSR and seven are taken from the LFSR. This function is of degree $\deg(h(x)) = 3$ and very simple. It is defined as

$$h(x) = h(x_0, x_1, \ldots, x_8) = x_0x_1 \oplus x_2x_3 \oplus x_4x_5 \oplus x_6x_7 \oplus x_0x_4x_8 \tag{8.12}$$

where the variables x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 and x_8 correspond to the tap positions b_{t+12}, s_{t+8}, s_{t+13}, s_{t+20}, b_{t+95}, s_{t+42}, s_{t+60}, s_{t+79} and s_{t+95} respectively. The output $H(B_t, S_t)$ function is defined as

$$z_t = H(B_t, S_t) = \bigoplus_{j \in \mathcal{A}} b_{t+j} \oplus h(x) \oplus s_{t+93}, \tag{8.13}$$

where $\mathcal{A} = \{2, 15, 36, 45, 64, 73, 89\}$.

Cipher Initialization The initialization is very similar to the initialization of the 80-bit variant of the cipher. The bits of the key K, denoted $k_i,\ 0 \leq i \leq 127$, and the bits of the IV, denoted $IV_i,\ 0 \leq i \leq 95$, are loaded into the NFSR and LFSR respectively as

$$\begin{cases} b_i = k_i, & 0 \leq i \leq 127 \\ s_i = IV_i, & 0 \leq i \leq 95 \end{cases} \tag{8.14}$$

The last 32 bits of the LFSR are filled with ones, $s_i = 1,\ 96 \leq i \leq 127$. After loading key and IV bits, the cipher is clocked 256 times without producing any keystream. The output function is fed back and xored with the input, both to the LFSR and to the NFSR.

8.2 Throughput Rate

Both the LFSR and the NFSR are clocked regularly. Thus, in its simplest implementation, 1 bit/clock is output. This can be compared to the shrinking generator, self-shrinking generator, alternating step generator and the bit-search generator discussed in chapters 4 and 5, which all outputs less than 1 bit/clock. If the keystream needs to be regularly produced these constructions require an output buffer.

In addition, it is possible to increase the throughput rate of the Grain ciphers by adding some additional hardware. This is an important feature of the Grain family of stream ciphers compared to many other stream ciphers. Increasing the speed can very easily be done by just implementing the feedback functions, $f(x)$ and $g(x)$, and the output function several times. In order to simplify this implementation, the last 15 bits in Grain and the last 31 bits in Grain-128 of the shift registers are not used in the feedback functions or in the input to the output function. I.e., $s_i,\ 65 \leq i \leq 79$ and $b_i,\ 65 \leq i \leq 79$ in Grain and $s_i,\ 97 \leq i \leq 127$ and $b_i,\ 97 \leq i \leq 127$ in Grain-128 are not used in the three functions. This allows the speed to be easily multiplied by up to 16 for Grain and 32 for Grain-128 if a sufficient amount of hardware is available. An overview of the implementation when the speed is doubled

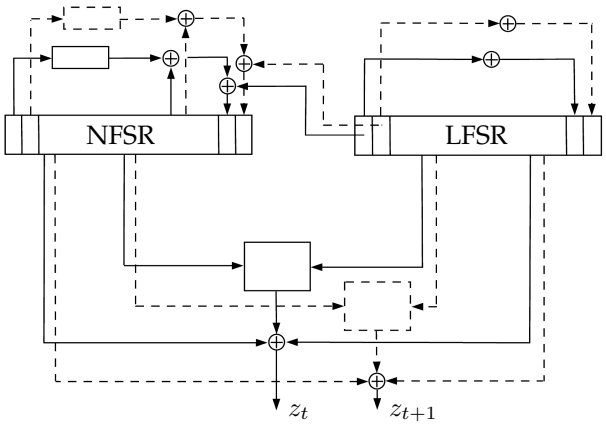

Figure 8.3: Implementation of Grain which outputs 2 bits/clock.

can be seen in Fig. 8.3. Naturally, the shift registers also need to be implemented such that each bit is shifted δ steps instead of one when the speed is increased by a factor δ. Since, in the key initialization, the cipher is clocked 160 times (Grain) or 256 times (Grain-128), the possibilities to increase the speed is limited to factors that are divisors of 160 or 256 respectively. The number of clockings needed in the key initialization phase is then $160/\delta$ or $256/\delta$. Since the output and feedback functions are small, it is quite feasible to increase the throughput in this way.

8.3 Security and Design Choices

In this section we give a security analysis of the construction and motivate the different design choices.

8.3.1 Linear Approximations

Attacking Grain using linear approximations of the two nonlinear functions turned out to be successful on the first version of Grain, denoted version 0. This attack was discovered by several independent researchers and the details can be found in [Max06, BGM06, KHK05]. The design choices in the current versions are influenced by this attack. In this subsection, we temporarily switch to the notation $s(t)$ instead of s_t as previously used to denote a value at time t.

With a slight abuse of notation, let us rewrite the update function of the NFSR as

$$0 = g(B_t) \oplus s(t). \tag{8.15}$$

Let the weight of a binary linear function ℓ, denoted $w(\ell)$, be the number of variables in the function. I.e., let $\ell(t) = \bigoplus_{i=0}^{n} c_i x(t+i)$, then

$$w(\ell) = |\{i \in 0..n : c_i = 1\}|. \tag{8.16}$$

Here, ℓ uses variables $s(t)$ and $b(t)$ and $w(\ell)$ is thus the number of shift register taps used in ℓ. Further, let $w_N(\ell)$ and $w_L(\ell)$ be the number of terms from the NFSR and from the LFSR respectively. Assume that we have found a linear approximation $\ell_g(t)$ of (8.15) i.e.,

$$\ell_g(t) = \bigoplus_{i=0}^{w(\ell_g)-1} b(t+\phi_i) \oplus s(t) \tag{8.17}$$

where $\phi_0, \phi_1, \ldots, \phi_{w(\ell_g)-1}$ denote the positions in the NFSR that are present in the linear approximation. The bias of $\ell_g(t)$ is denoted ε_g, i.e.,

$$\Pr(\ell_g(t) = 0) = \frac{1}{2}(1+\varepsilon_g), \quad 0 < |\varepsilon_g| \leq 1. \tag{8.18}$$

Similarly, a linear approximation $\ell_H(t)$ of the output function $H(B_t, S_t)$ can be written as

$$\ell_H(t) = \bigoplus_{i=0}^{w_N(\ell_H)-1} b(t+\xi_i) \oplus \bigoplus_{i=0}^{w_L(\ell_H)-1} s(t+\psi_i) \tag{8.19}$$

where $\xi_0, \xi_1, \ldots, \xi_{w_N(\ell_H)-1}$ and $\psi_0, \psi_1, \ldots, \psi_{w_L(\ell_H)-1}$ determines the location of the taps in the NFSR and LSFR used in the linear approximation. The bias of (8.19) is denoted ε_H, i.e.,

$$\Pr(\ell_H(t) = z(t)) = \frac{1}{2}(1+\varepsilon_H), \quad 0 < |\varepsilon_H| \leq 1. \tag{8.20}$$

Now, sum up the keystream bits determined by ϕ_i in (8.17),

$$\begin{aligned} z(t+\phi_0) \oplus z(t+\phi_1) \oplus \ldots \oplus z(t+\phi_{w(\ell_g)-1}) &\stackrel{p}{=} \\ \ell_H(t+\phi_0) \oplus \ell_H(t+\phi_1) \oplus \ldots \oplus \ell_H(t+\phi_{w(\ell_g)-1}). & \end{aligned} \tag{8.21}$$

Using the piling-up lemma (2.58), the relation (8.21) holds with probability $p = 1/2(1+\varepsilon_H^{w(\ell_g)})$. The terms on the right hand side of (8.21) will consist of $w_N(\ell_H) \cdot w(\ell_g)$ terms from the NFSR and $w_L(\ell_H) \cdot w(\ell_g)$ terms from the LFSR.

All terms from the NFSR can now be approximated using (8.17) resulting in a relation involving only keystream bits and LFSR bits as

$$\bigoplus_{i=0}^{w(\ell_g)-1} z(t+\phi_i) \stackrel{p'}{=} \bigoplus_{i=0}^{w(\ell_g)-1} \bigoplus_{j=0}^{w_L(\ell_H)-1} s(t+\phi_i+\psi_j) \oplus \bigoplus_{i=0}^{w_N(\ell_H)-1} s(t+\xi_i), \quad (8.22)$$

which holds with probability $p' = 1/2(1+\varepsilon_{tot})$ with

$$\varepsilon_{tot} = \varepsilon_g^{w_N(\ell_H)} \cdot \varepsilon_H^{w(\ell_g)}. \quad (8.23)$$

From this point there are several possibilities for attacks. By finding a multiple of the LFSR feedback polynomial of weight 3, a distinguishing attack can be mounted. The expected degree of this multiple would be around $2^{|K|/2}$. Combining the keystream bits given by the multiple and using the approximation that $1/\varepsilon^2$ samples are needed in the distinguisher (see (2.62)), about

$$N = 2^{|K|/2} + \frac{1}{\varepsilon_{tot}^6} \quad (8.24)$$

keystream bits are required in the attack.

Another approach is to try to recover the state of the LFSR. An obvious way of doing this is to exhaustively search the state and determine which state gives the bias in (8.23). In this case, only about

$$N = \frac{|K| \cdot 2\ln 2}{\varepsilon_{tot}^2} \quad (8.25)$$

keystream bits are needed according to (2.61). Since the size of the LFSR is the same as the key size, this method is obviously more expensive than exhaustive key search. A faster algorithm was given in [BGM06], where they generate more equations of the form (8.22). By only using the most favourable equations, and by using the Fast Walsh Transform, the attack complexity could be made significantly lower. We refer to [BGM06] for more details on this attack.

Due to this attack, the parameters of Grain Version 0 were changed. A higher resiliency was added to the NFSR feedback function, increasing $w(\ell_g)$ and several linear terms from the NFSR was added to the output function, increasing $w_N(\ell_H)$. The result was Grain Version 1, referred to as Grain, in this thesis.

The design of Grain-128 is inspired by the analysis in this section. Thus, the NFSR feedback function should satisfy the following three criteria

- High resiliency $\Rightarrow$ many terms in linear approximation (high $w(\ell_g)$). This can be achieved by adding several linear terms to the function. Each linear term will increase the resiliency by one.

- High nonlinearity $\Rightarrow$ small bias of linear approximations (small ε_g). This can be achieved by using a bent function, i.e., a function with maximum nonlinearity.
- Small hardware implementation $\Rightarrow$ attractive in low-cost implementations.

A well-known n-variable bent function is the function $x_1x_2 \oplus x_3x_4 \oplus \ldots \oplus x_{n-1}x_n$. This function is also very small in hardware. Using $n = 14$ and adding 5 linear terms gives a 4-resilient Boolean function with nonlinearity 260096. The best linear approximations have bias $\varepsilon_g = 2^{-7}$ and $w(\ell_g) \geq 5$.

The output function has the same design criteria as the NFSR feedback function. However, to increase the algebraic degree it has a term of degree 3. It has nonlinearity 61440 and resiliency 7. The best linear approximations have bias $\varepsilon_H = 2^{-4}$ and $w_N(\ell_H) \geq 7$.

8.3.2 Time-Memory Tradeoff Attacks

In the review of time-memory tradeoff attacks in Section 2.3.3 it was concluded that the state must be at least twice the key size in order to prevent these attacks. Both the LFSR and NFSR are of size $|K|$ bits, and thus the state is exactly twice the key size. Since Grain is designed to be as small as possible in hardware, no extra state bits are added to the design. The state is relatively expensive to implement in hardware and it is important to keep it as small as possible. In [HS05] it was noted that the initialization process of a stream cipher could be seen as a one-way function i.e., the function taking the key K and the IV IV as input and outputs the first $|K| + |IV|$ bits of the keystream. In this case the search space is $2^{|K|+|IV|}$ and new data is generated by repeated initializations of the cipher. If we allow a preprocessing time P that is higher than exhaustive key search $2^{|K|}$, then it is possible to have an attack with real time complexity lower than exhaustive key search. Table 8.1 gives attack complexities for Grain and Grain-128 in the time-memory tradeoff setting of [HS05]. If $|IV| < \frac{1}{2}|K|$ then it is possible to have the preprocessing time also smaller than exhaustive key search. In this case we need to initialize with several different keys and we will only retrieve one of these keys in the real time phase. In the Grain ciphers $|IV| > \frac{1}{2}|K|$ so this is not applicable here.

8.3.3 Algebraic Attacks

As mentioned in Section 2.3.5, algebraic attacks can be very successful on nonlinear filter generators. Especially if the output function is of very low degree. Grain is very similar to a nonlinear combiner. However, the introduction of the NFSR in the design will defeat all algebraic attacks known

Attack Complexities				
	T	D	M	P
Grain	2^{80}	2^{40}	2^{64}	2^{104}
	2^{72}	2^{36}	2^{72}	2^{108}
Grain-128	2^{128}	2^{64}	2^{96}	2^{160}
	2^{112}	2^{56}	2^{112}	2^{168}

Table 8.1: Time-Memory tradeoff attack with real time complexity T, D initializations, M memory words and preprocessing time P.

today. Since the update function of the NFSR is nonlinear, the later state bits of the NFSR as a function of the initial state bits will have varying but large algebraic degree. As the output function has several inputs from the NFSR, the algebraic degree of the keystream bits expressed as functions of key bits will be large in general. This will defeat algebraic attacks.

8.3.4 Chosen-IV Attacks

A necessary condition for defeating differential-like or statistical chosen-IV attacks is that the initial states for any two chosen IV's (or sets of IV's) are algebraically and statistically unrelated. The number of cycles in key initialization has been chosen so that the Hamming weight of the differences in the full initial 160-bit state for two IV's after initialization is close to random. This should prevent chosen-IV attacks.

It may be tempting to improve the efficiency of the key initialization by just decreasing the number of initial clockings. Considering the 80-bit variant of Grain, after only 80 clocks, all bits in the state will depend on both the key and the IV. However, in a chosen-IV attack it is possible to reinitialize the cipher with the same key but with an IV that differs in only one position from the previous IV. Consider the case when the number of initial clockings is 80 and the last bit of the IV is flipped i.e., s_{63} is flipped. This is the event that occurs if the IV is chosen as a sequence number. Looking at the difference of the states after initialization it is clear that several positions will be predictable. The bit s_{63} is not used in the feedback or in the filter function, hence, the first register update will be the same in both cases. Consequently, the bit s_0 will be the same in both initializations. In the next update, the flipped bit will be in position s_{62}. This position is used in the linear feedback of the LFSR, and consequently the bit s_1 will always be different for the two initializations. Similar arguments can be used to show that the difference in the state will be deterministic in more than half of the

160 state bits. This deterministic difference in the state can be exploited in a distinguishing attack. Let $\mathbf{x}$ be the input variables to the output function, H, after the first initialization and let $\mathbf{x}_\Delta$ be the input variables to the output function after the second initialization. Now, compute the distribution of $\Pr(\mathbf{x}, \mathbf{x}_\Delta)$. If this distribution is biased, it is possible[1] that the distribution of the difference in the first output bit,

$$\Pr(H(\mathbf{x}) \oplus H(\mathbf{x}_\Delta)), \tag{8.26}$$

is biased. Assume that

$$\Pr(H(\mathbf{x}) \oplus H(\mathbf{x}_\Delta) = 0) = 1/2(1 + \varepsilon), \quad 0 < |\varepsilon| \leq 1. \tag{8.27}$$

then the number of initializations we need will be in the order of $1/\varepsilon^2$. This attack can be optimized by calculating which output bit will give the highest bias since it is not necessarily the bits in the registers corresponding to the input bits of $H(\mathbf{x})$ that have deterministic difference after the initializations. This attack shows that it is preferred that the probability that any state bit is the same after initialization with two different IVs should be close to 0.5. As with the case of 80 initialization clocks, it is easy to show that after 96, 112 and 128 there are also state bits that will always be the same or that will always differ.

It is possible to reduce the required number of initial clockings by loading the NFSR and LFSR differently. If each entry of the registers is loaded with the xor of a few key and IV bits and each key and IV bit influences the loading of several entries, differences in the IV will propagate faster. The reason for not doing this is mainly that all the extra xors needed would make the cipher larger in hardware.

8.3.5 Fault Attacks

Amongst the strongest attacks conceivable on any cipher, are fault attacks. Fault attacks against stream ciphers have been initiated in [HS04], and have shown to be efficient against many known constructions of stream ciphers. This suggests that it is hard to completely defeat fault attacks on stream ciphers. In the scenario in [HS04] it is assumed that the attacker can apply some bit flipping faults to one of the two feedback registers at his will. However he has only partial control over their number, location, and exact timing, and similarly on what concerns his knowledge. A stronger assumption one can make, is that he is able to flip a single bit (at a time instance, and thus at a location, he does not know exactly). In addition, he can reset the

[1] It is possible, but maybe not very likely. One unbiased linear variable is enough to make the output unbiased.

device to its original state and then apply another randomly chosen fault to the device. We adapt the methods in [HS04] to the present cipher. Thereby, we make the strongest possible assumption (which may not be realistic) that an attacker can induce a single bit fault in the LFSR, and that he is somehow able to determine the exact position of the fault. The aim is to study input-output properties for $H(B_t, S_t)$, and to derive information on the inputs. As long as the difference induced by the fault in the LFSR does not propagate to position b_{t+63} in Grain or b_{t+95} in Grain-128, the difference observed in the output of the cipher is coming from inputs of $H(B_t, S_t)$ from the LFSR alone. If an attacker is able to reset the device and to induce a single bit fault many times and at different positions that he can correctly guess from the output difference, we cannot preclude that he will get information about a subset of the state bits in the LFSR. Such an attack seems more difficult under the (more realistic) assumption that the fault induced affects several state bits at (partially) unknown positions, since in this case it is more difficult to determine the induced difference from output differences.

Likewise, one can consider faults induced in the NFSR alone. These faults do not influence the contents of the LFSR. However, faults in the NFSR propagate nonlinearly and their evolution will be harder to predict. Thus, a fault attack on the NFSR seems more difficult.

8.4 Hardware Performance

The Grain family of stream ciphers is designed to be very small in hardware. In this section we give an estimate of the gate count resulting from a hardware implementation of the cipher. We also give performance results from independent implementers in order to show how Grain compares to other stream ciphers in eSTREAM.

The gate count is an important property that indicates how large and expensive the construction is. The size of a gate depends on the type of gate that is used and the gate count is a normalized number indicating the number of gates, equivalent to 2 input nand gates, that are required in a construction. A 2 input nand gate can be constructed using 4 transistors while an xor gate needs 10 transistors. Adding extra inputs to the nand gate require 2 transistors per extra input. A D-flipflop can be implemented in several ways depending on the functionality. A gate count of 8 is adequate in our application. With this as background, the gate count for the building blocks used in the Grain ciphers are listed in Table 8.2. However, these should not be seen as natural constants since they will depend on the implementation in an actual chip. Based on these numbers, we estimate the number of gates needed in an implementation. The gate count for Grain can be found in Table 8.3 and for Grain-128 in Table 8.4.

Function	*Gate Count*
D-flipflop	8
NAND2	1
NAND3	1.5
NAND4	2
NAND5	2.5
NAND6	3
XOR2	2.5

Table 8.2: The gate count used for different functions.

Grain	*Speed Increase*				
Building Block	1x	2x	4x	8x	16x
LFSR	640	640	640	640	640
NFSR	640	640	640	640	640
$f(\cdot)$	12.5	25	50	100	200
$g(\cdot)$	75	150	300	600	1200
$H(\cdot)$	50.5	101	202	404	808
Total	**1418**	**1556**	**1832**	**2384**	**3488**

Table 8.3: The estimated gate count in an actual implementation of Grain.

Note that these numbers are just estimates, e.g., the multiplexers needed in order to switch between key/IV loading, initialization and keystream generation are not included in the count. Also, two extra xors are needed in key initialization mode. The exact number of gates needed for each function will depend on the implementation anyway, and thus, the numbers in the tables should be seen as ballpark figures.

The hardware performance of a cipher can be measured with several different metrics in mind e.g., area, throughput and power. The importance of the different metrics will depend on the application. Being candidates in the eSTREAM project, Grain and Grain-128 has been implemented in hardware by several independent researchers. The ciphers have also been compared to some of the other candidates in eSTREAM. In [GB07], ASIC implementations in $0.13\mu m$ Standard Cell CMOS of several eSTREAM candidates were conducted and compared with respect to different metrics. Three situations were considered

(*i*) **Maximum clock frequency.** The implementation running at the fastest

Grain-128	*Speed Increase*					
Building Block	1x	2x	4x	8x	16x	32x
LFSR	1024	1024	1024	1024	1024	1024
NFSR	1024	1024	1024	1024	1024	1024
$f(\cdot)$	12.5	25	50	100	200	400
$g(\cdot)$	37	74	148	296	592	1184
$H(\cdot)$	35.5	71	142	284	568	1136
Total	**2133**	**2218**	**2388**	**2728**	**3408**	**4768**

Table 8.4: The estimated gate count in an actual implementation of Grain-128.

possible speed, maximizing the throughput.

(*ii*) **Output rate of 10Mbps.** This corresponds to an implementation targeting an environment with a fixed throughput of 10Mbps e.g., WLAN.

(*iii*) **Clock frequency of 100kHz.** This corresponds to an implementation targeting environments with a fixed clock frequency of 100kHz. An example is RFID applications.

In [GB07] there are many different metrics considered and in Table 8.5 we give, in addition to the gate count, two different metrics.

- **Energy/bit.** The total power consumption divided by the throughput.
- **Throughput/Area.** The throughput divided by the area of the design. This is often used as a measure of the design efficiency.

The original paper additionally considers metrics such as initialization cycles, maximum clock frequency, maximum throughput, leakage power, total power and area-time product.

At the time of writing, Grain has advanced to the third phase in the hardware profile of eSTREAM together with 7 other candidates. Apart from the 4 ciphers given in Table 8.5, Pomaranch Version 3, Edon80 [GMKG05], DECIM^{v2} [BBC$^+$06] and Moustique [DK07] are also in the hardware category of phase 3. In [KKLRP06] the authors demonstrated an implementation of Edon80 with a gate count of approximately 3000. Without going into details, Edon80 consists of 80 stages, each state being updated by a quasigroup transformation. The implementation in [KKLRP06] goes through the stages one by one and is thus very slow. A faster implementation can update all stages in one clock cycle but will require approximately a gate count of 7500. To the best of our knowledge, there are no implementations of DECIM^{v2} and Pomaranch Version 3 giving hardware performance.

Cipher	Keysize	Gate Count	Max Clock Freq		10 Mbps		100 kHz	
			Energy/bit pJ/bit	Tput/Area kbps/μm^2	Energy/bit pJ/bit	Tput/Area kbps/μm^2	Energy/bit pJ/bit	Tput/Area kbps/μm^2
Grain	80	1294	10.73	108.00	10.95	1.490	33.0	0.0149
Grain 4x	80	1678	3.08	319.33	3.41	1.150	11.2	0.0460
Grain 8x	80	2191	1.83	445.78	2.29	0.880	7.6	0.0704
Grain 16x	80	3239	1.21	588.27	1.95	0.596	5.8	0.0953
Trivium	80	2599	17.74	26.61	18.12	0.742	56.1	0.0074
Trivium 4x	80	2660	4.52	119.88	4.92	0.725	14.6	0.0290
Trivium 8x	80	2801	2.44	198.16	2.88	0.689	8.0	0.0551
Trivium 16x	80	3185	1.41	395.57	1.99	0.606	5.1	0.0969
Trivium 32x	80	3787	0.86	571.88	1.61	0.509	3.2	0.1630
Trivium 64x	80	4921	0.57	874.14	1.64	0.392	2.2	0.2509
F-FCSR-H	80	4760	3.27	127.13	4.06	0.405	13.2	0.0324
Grain-128	128	1857	16.51	96.20	16.77	1.039	43.5	0.0104
Grain-128 4x	128	2129	4.49	211.98	4.87	0.906	14.0	0.0362
Grain-128 8x	128	2489	2.50	360.52	2.99	0.775	8.6	0.0620
Grain-128 16x	128	3189	1.55	523.10	2.24	0.605	5.8	0.0968
Grain-128 32x	128	4617	1.04	604.92	2.19	0.418	4.6	0.1337
Mickey-128	128	5039	30.28	15.82	31.07	0.383	111.7	0.0038

Table 8.5: Performance results of a few eSTREAM candidates in 0.13μm CMOS Standard Cell. Note that the basic implementations of Grain and Trivium as well as Mickey128 outputs 1 bit/clock while F-FCSR-H outputs 8 bits/clock.

Though, in [BBC+06] the authors of DECIM^{v2} indicate that it can be implemented with gate count of about 2500. In [JHK07] the authors of Pomaranch Version 3 claims that the 128-bit variant and the 80-bit variant can be implemented with a gate count of 3400 and 2200 respectively. Moustique is a self-synchronizing stream cipher and is thus fundamentally different from the other designs, which are all synchronous. In [GCB06] a hardware implementation of Mosquito (the predecessor of Moustique) was done and the gate count was about 6800 for a fast pipelined variant and 4200 for a slower variant. These figures should be quite similar for Moustique.

These figures indicate that Grain is undoubtedly the smallest stream cipher in eSTREAM and if area is the most important feature Grain should be the first choice. It also provides flexibility in that the speed can be increased very easily. However, if speed is very important then the fastest version of

Trivium seems to be the most attractive choice.

8.5 Summary

In this chapter we introduced a new family of stream ciphers, Grain. Two different versions, denoted Grain and Grain-128, have been specified. The designs target hardware environments where small area is of high importance. The basic implementation is very small but outputs only one bit/clock. An important feature in the Grain ciphers is the possibility to easily increase the throughput by adding some extra hardware. This is done by simply implementing the relatively small feedback and output functions several times. This flexibility makes the Grain ciphers attractive for a wide range of applications spanning from the most demanding in terms of small hardware area to applications requiring a very high throughput.

Together with Trivium, which is also both small and flexible, the Grain ciphers seem to currently provide the most attractive confidentiality solutions for constrained hardware environments. However, it should be noted that security is still the most important feature and cryptanalysis results may render a stream cipher useless overnight. New primitives need several years of public security evaluations before they can be recommended for use in widespread applications.

9

Concluding Remarks

This thesis has presented cryptanalysis results on stream ciphers, both on a general construction and on specific designs. A new class of weak feedback polynomials for linear feedback shift registers has been defined. Also, different methods to recover the initial state of the shift register have been shown for two algorithms that irregularly decimates the output sequence of the shift register. Further, we have given cryptanalytic results for two stream ciphers in the eSTREAM project. Most attacks given in the thesis are not possible to mount in practice since they require data, memory and/or computational complexity that is beyond the ability of today's computers. However, additional observations may improve the complexities significantly. Finally, a new design is proposed. It is designed to be extremely small and simple to implement in hardware.

The goal of eSTREAM is, according to the web page, to identify new stream ciphers that might become suitable for widespread adoption. This is done by allowing researchers to propose an algorithm and then let the cryptographic community analyze the algorithms. If an algorithm, after several years, is still considered secure, efficient and attractive, then eSTREAM can consider it in its final portfolio of algorithms. The opposite approach is for developers to construct their own algorithm and, hoping it is secure, implement it in a product. Widespread algorithms such as E_0 in Bluetooth and $A5/1$ in GSM are examples of this approach. These two stream ciphers have been shown to be very insecure and there are many attacks proposed on them, each new attack improving on the previous. Hopefully, the efforts in eSTREAM can help avoiding mistakes such as E_0 and $A5/1$ in the future.

Bibliography

[ACG+06] F. Armknecht, C. Carlet, P. Gaborit, S. Künzli, W. Meier, and O. Ruatta. Efficient computation of algebraic immunity for algebraic and fast algebraic attacks. In *Advances in Cryptology—EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 147–164. Springer-Verlag, 2006.

[AK03] F. Armknecht and M. Krause. Algebraic attacks on combiners with memory. In D. Boneh, editor, *Advances in Cryptology—CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 2003.

[Arm04] F. Armknecht. Improving fast algebraic attacks. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 65–82. Springer-Verlag, 2004.

[Bab95] S. Babbage. A space/time tradeoff in exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection*, number 408 in IEE Conference Publication, 1995.

[BBC+05] C. Berbain, O. Billet, A. Canteaut, N. Courtois, B. Debraize, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert. Decim - a new stream cipher for hardware applications. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/004, 2005. *http://www.ecrypt.eu.org/stream*.

[BBC+06] C. Berbain, O. Billet, A. Canteaut, N. Courtois, B. Debraize, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert. DECIM v2. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/004, 2006. *http://www.ecrypt.eu.org/stream*.

[Ber05] D. J. Bernstein. Understanding brute force. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/036, 2005. *http://www.ecrypt.eu.org/stream*.

[BGM06] C. Berbain, H. Gilbert, and A. Maximov. Cryptanalysis of Grain. In M. Robshaw, editor, *Fast Software Encryption 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 15–29. Springer-Verlag, 2006.

[BL05] A. Braeken and J. Lano. On the (im)possibility of practical and secure nonlinear filters and combiners. In B. Preneel and S. Tavares, editors, *Selected Areas in Cryptography—SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 159–174. Springer-Verlag, 2005.

[Blu04] SIG Bluetooth. Core specification v2.0 + edr. Available at *http://www.bluetooth.com*, 2004.

[BMS06] A. Biryukov, S. Mukhopadhyay, and P. Sarkar. Improved time-memory trade-offs with multiple data. In *Selected Areas in Cryptography—SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 110–127. Springer-Verlag, 2006.

[BS00] A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In T. Okamoto, editor, *Advances in Cryptology—ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2000.

[BS05] E. Biham and J. Seberry. Py (roo) : A fast and secure stream cipher using rolling arrays. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/023, 2005. *http://www.ecrypt.eu.org/stream*.

[CGJ06] C. Cid, H. Gilbert, and T. Johansson. Cryptanalysis of Pomaranch. *IEE Proceedings - Information Security*, 153(2):51–53, 2006.

[CKM93] D. Coppersmith, H. Krawczyk, and Y. Mansour. The shrinking generator. In D.R. Stinson, editor, *Advances in Cryptology—CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1993.

[CKPS00] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In B. Preneel, editor, *Advances in Cryptology—EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer-Verlag, 2000.

[CM03] N. Courtois and W. Meier. Algebraic attacks on stream ciphers with linear feedback. In E. Biham, editor, *Advances in Cryptology—EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 345–359. Springer-Verlag, 2003.

[Cou03a] N. Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In D. Boneh, editor, *Advances in Cryptology—CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 176–194. Springer-Verlag, 2003.

[Cou03b] N. Courtois. Higher order correlation attacks, XL algorithm and cryptanalysis of Toyocrypt. In P.J. Lee and C.H. Lim, editors, *Information Security and Cryptology - ICISC 2002*, volume 2587 of *Lecture Notes in Computer Science*, pages 182–199. Springer-Verlag, 2003.

[CP04] J. Y. Cho and J. Pieprzyk. Algebraic attacks on SOBER-t32 and SOBER-t16 without stuttering. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, 2004.

[CT91] T. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley series in Telecommunication. Wiley, 1991.

[CT00] A. Canteaut and M. Trabbia. Improved fast correlation attacks using parity-check equations of weight 4 and 5. In B. Preneel, editor, *Advances in Cryptology—EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 573–588. Springer-Verlag, 2000.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[Dau05] M. Daum. Narrow t-functions. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 50–67. Springer-Verlag, 2005.

[Den82] D. Denning. Cryptography and data security, 1982. p.100, Out of Print.

[DGM05] D. K. Dalai, K. C. Gupta, and S. Maitra. Cryptographically significant Boolean functions: Construction and analysis in terms of algebraic immunity. In *Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 98–111. Springer-Verlag, 2005.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[DK05] J. Daemen and P. Kitsos. Submission to ecrypt call for stream ciphers: the self-synchronizing stream cipher MOSQUITO. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/018, 2005. *http://www.ecrypt.eu.org/stream*.

[DK07] J. Daemen and P. Kitsos. The self-synchronizing stream cipher MOUSTIQUE. eSTREAM, ECRYPT Stream Cipher Project, 2007. *http://www.ecrypt.eu.org/stream*.

[DLP05] J. Daemen, J. Lano, and B. Preneel. Chosen ciphertext attack on SSS. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/044, 2005. *http://www.ecrypt.eu.org/stream*.

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

[DT06] F. Didier and J. Tillich. Computing the algebraic immunity efficiently. In M. Robshaw, editor, *Fast Software Encryption 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 359–374. Springer-Verlag, 2006.

[ECR] ECRYPT. eSTREAM: ECRYPT Stream Cipher Project, IST-2002-507932. Available at *http://www.ecrypt.eu.org/stream/*.

[EFF07] EFF webpage, Accessed Feb 2007. *http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/*.

[EHJ04] H. Englund, M. Hell, and T. Johansson. Correlation attacks using a new class of weak feedback polynomials. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 2004.

[EHJ06] H. Englund, M. Hell, and T. Johansson. Two general attacks on Pomaranch-like keystream generators. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/001, 2006. *http://www.ecrypt.eu.org/stream*.

[EHJ07] H. Englund, M. Hell, and T. Johansson. A note on distinguishing attacks. In T. Helleseth, P. Vijay Kumar, and Ø. Ytrehus, editors, *Proceedings of the 2007 IEEE Information Theory Workshop on Information Theory for Wireless Networks*, pages 87–90, 2007.

[EJ00] P. Ekdahl and T. Johansson. SNOW - a new stream cipher. In *Proceedings of First Open NESSIE Workshop*, 2000.

[EJ02] P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. In K. Nyberg and H. Heys, editors, *Selected Areas in Cryptography—SAC 2002*, volume 2595 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 2002.

[Fau02] J. C. Faugére. A new efficient algorithm for computing Gröbner bases without reduction to zero. In *Workshop of Applications of Commutative Algebra*. ACM Press, 2002.

[FM00] S. R. Fluhrer and D. A. McGrew. Statistical analysis of the alleged RC4 keystream generator. In B. Schneier, editor, *Fast Software Encryption 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 19–30. Springer-Verlag, 2000.

[For90] R. Forré. The strict avalanche criterion: spectral properties of Boolean functions and an extended definition. In *Advances in Cryptology—CRYPTO'88*, volume 403 of *Lecture Notes in Computer Science*, pages 450–468. Springer-Verlag, 1990.

[Gam07] B. M. Gammel. Achterbahn stream cipher site. *http://www.matpack.de/achterbahn/index.html*, 2007. Latest accessed June 2007.

[GB07] T. Good and M. Benaissa. Hardware results for selected stream cipher candidates. The State of the Art of Stream Ciphers, Workshop Record, SASC 2007, Bochum, Germany, January 2007.

[GCB06] T. Good, W. Chelton, and M. Benaissa. Review of stream cipher candidates from a low resource hardware perspective. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/016, 2006. *http://www.ecrypt.eu.org/stream*.

[GG07] R. Göttfert and B. M. Gammel. On the frame length of Achterbahn-128/80. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/041, 2007. *http://www.ecrypt.eu.org/stream*.

[GGK05a] B. M. Gammel, R. Göttfert, and O. Kniffler. The Achterbahn stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/002, 2005. *http://www.ecrypt.eu.org/stream*.

[GGK05b] B. M. Gammel, R. Göttfert, and O. Kniffler. Improved Boolean combining functions for Achterbahn. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/072, 2005. *http://www.ecrypt.eu.org/stream*.

[GGK06a] B. M. Gammel, R. Göttfert, and O. Kniffler. Achterbahn-128/80. eSTREAM, ECRYPT Stream Cipher Project, 2006. *http://www.ecrypt.eu.org/stream*.

[GGK06b] B. M. Gammel, R. Göttfert, and O. Kniffler. Status of Achterbahn and tweaks. The State of the Art of Stream Ciphers, Workshop Record, SASC 2006, Leuven, Belgium, February 2006.

[GMKG05] D. Gligoroski, S. Markovski, L. Kocarev, and M. Gusev. Edon80. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/007, 2005. *http://www.ecrypt.eu.org/stream*.

[Gün88] C.G. Günther. Alternating step generators controlled by de Bruijn sequences. In D. Chaum and W.L. Price, editors, *Advances in Cryptology—EUROCRYPT'87*, volume 304 of *Lecture Notes in Computer Science*, pages 91–103. Springer-Verlag, 1988.

[Gol96] J.D. Golić. Computation of low-weight parity-check polynomials. *Electronic Letters*, 32(21):1981–1982, October 1996.

[Gol97a] J.D. Golić. Cryptanalysis of alleged A5 stream cipher. In W. Fumy, editor, *Advances in Cryptology—EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer-Verlag, 1997.

[Gol97b] J.D. Golić. Linear statistical weakness of alleged RC4 keystream generator. In W. Fumy, editor, *Advances in Cryptology—EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 226–238. Springer-Verlag, 1997.

[GS04] A. Gouget and H. Sibert. The bit-search generator. The State of the Art of Stream Ciphers, Workshop Record, SASC 2004, Brugge, Belgium, October 2004.

[GSB+05] A. Gouget, H. Sibert, C. Berbain, N. Courtois, B. Debraize, and C. Mitchell. Analysis of the bit-search generator and sequence compression techniques. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 196–214. Springer-Verlag, 2005.

[Hel80] M.E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, IT-26(4):401–406, July 1980.

[HJ05] M. Hell and T. Johansson. Some attacks on the bit-search generator. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 215–227. Springer-Verlag, 2005.

[HJ06a] M. Hell and T. Johansson. Cryptanalysis of Achterbahn-version 2. In E. Biham, editor, *Selected Areas in Cryptography—SAC 2006*, volume 4356 of *Lecture Notes in Computer Science*, pages 45–55. Springer-Verlag, 2006.

[HJ06b] M. Hell and T. Johansson. On the problem of finding linear approximations and cryptanalysis of Pomaranch version 2. In E. Biham, editor, *Selected Areas in Cryptography—SAC 2006*, volume 4356 of *Lecture Notes in Computer Science*, pages 220–234. Springer-Verlag, 2006.

[HJ06c] M. Hell and T. Johansson. Two new attacks on the self-shrinking generator. *IEEE Transactions on Information Theory*, 52(8):3837–3843, 2006.

[HJ07] M. Hell and T. Johansson. Cryptanalysis of Achterbahn-128/80. *IET Information Security*, 1(2):47–52, 2007.

[HJM06] M. Hell, T. Johansson, and W. Meier. Grain - a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing, Special Issue on Security of Computer Network and Mobile Systems.*, 2006.

[HJMM06] M. Hell, T. Johansson, A. Maximov, and W. Meier. A stream cipher proposal: Grain-128. In *International Symposium on Information Theory—ISIT 2006*. IEEE, 2006.

[HLY+05] J. Hong, D. H. Lee, Y. Yeom, D. Han, and S. Chee. T-function based stream cipher TSC-3. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/031, 2005. *http://www.ecrypt.eu.org/stream*.

[HLYH05] J. Hong, D. H. Lee, Y. Yeom, and D. Han. A new class of single cycle T-functions. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 68–82. Springer-Verlag, 2005.

[HR02] P. Hawkes and G. Rose. Guess-and-determine attacks on SNOW. In K. Nyberg and H. Heys, editors, *Selected Areas in Cryptography—SAC 2002*, volume 2595 of *Lecture Notes in Computer Science*, pages 37–46. Springer-Verlag, 2002.

[HR03] P. Hawkes and G. Rose. Primitive specification for SOBER-128. Cryptology ePrint Archive, Report 2003/081, 2003. *http://eprint.iacr.org/*.

[HR04] P. Hawkes and G. Rose. Rewriting variables: The complexity of fast algebraic attacks on stream ciphers. In M. Franklin, editor, *Advances in Cryptology—CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 390–406. Springer-Verlag, 2004.

[HS04] J.J. Hoch and A. Shamir. Fault analysis of stream ciphers. In *CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 240–253. Springer-Verlag, 2004.

[HS05] J. Hong and P. Sarkar. New applications of time memory data tradeoffs. In B. Roy, editor, *Advances in Cryptology—ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 353–372. Springer-Verlag, 2005.

[JÖ2] F. Jönsson. *Some Results on Fast Correlation Attacks*. PhD thesis, Lund University, Department of Information Technology, P.O. Box 118, SE–221 00, Lund, Sweden, 2002.

[Jan05] C.J.A. Jansen. Stream cipher design based on jumping finite state machines. Available at *http://eprint.iacr.org/2005/267/.*, 2005.

[JHK06] C.J.A. Jansen, T. Helleseth, and A. Kholosha. Pomaranch - design and analysis of a family of stream ciphers. The State of the Art of Stream Ciphers, Workshop Record, SASC 2006, Leuven, Belgium, February 2006.

[JHK07] C.J.A. Jansen, T. Helleseth, and A. Kholosha. A lightweight implementation of the Pomaranch S-box. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/042, 2007. *http://www.ecrypt.eu.org/stream*.

[JM06] A. Joux and F. Müller. Chosen-ciphertext attacks against MOSQUITO. In M. Robshaw, editor, *Fast Software Encryption 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 390–404. Springer-Verlag, 2006.

[JMM05] T. Johansson, W. Meier, and F. Müller. Cryptanalysis of Achterbahn. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/064, 2005. *http://www.ecrypt.eu.org/stream*.

[JMM06] T. Johansson, W. Meier, and F. Müller. Cryptanalysis of Achterbahn. In M. Robshaw, editor, *Fast Software Encryption 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 2006.

[Kai05] U. Kaiser. Hermes-8. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/012, 2005. *http://www.ecrypt.eu.org/stream*.

[Kha05] S. Khazaei. Cryptanalysis of Pomaranch (CJCSG). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/065, 2005. *http://www.ecrypt.eu.org/stream*.

[KHK05] S. Khazaei, M. Hassanzadeh, and M. Kiaei. Distinguishing attack on Grain. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/071, 2005. *http://www.ecrypt.eu.org/stream*.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M.J. Wiener, editor, *Advances in Cryptology—CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[KKLRP06] M. Kasper, S. Kumar, K. Lemke-Rust, and C. Paar. A compact implementation of Edon80. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/057, 2006. *http://www.ecrypt.eu.org/stream*.

[KMP+98] L.R. Knudsen, W. Meier, B. Preneel, V. Rijmen, and S. Verdoolaege. Analysis methods for (alleged) RC4. In K. Ohta and D. Pei, editors, *Advances in Cryptology—ASIACRYPT'98*, volume 1998 of *Lecture Notes in Computer Science*, pages 327–341. Springer-Verlag, 1998.

[KPP+06] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with COPACOBANA – a cost-optimized parallel code breaker. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 101–118. Springer-Verlag, 2006.

[Kra02] M. Krause. BDD-based cryptanalysis of keystream generators. In L.R. Knudsen, editor, *Advances in Cryptology—EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 222–237. Springer-Verlag, 2002.

[KS03] A. Klimov and A. Shamir. A new class of invertible mappings. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 470–483. Springer-Verlag, 2003.

[KS04a] A. Klimov and A. Shamir. Cryptographic application of T-functions. In *Selected Areas in Cryptography—SAC 2003*, volume 3006 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 2004.

[KS04b] A. Klimov and A. Shamir. New cryptographic primitives based on multiword T-functions. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2004.

[KS05] A. Klimov and A. Shamir. New Applications of T-functions in Block Ciphers and Hash Functions. In *Fast Software Encryption 2005*. Springer-Verlag, 2005.

[LC04] S. Lin and D. J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., 2004.

[LKH+04] D. H. Lee, J. Kim, J. Hong, J. W. Han, and D. Moon. Algebraic attacks on summation generators. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 34–48. Springer-Verlag, 2004.

[Mas69] J.L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15:122–127, 1969.

[Max05a] A. Maximov. A new stream cipher Mir-1. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/017, 2005. *http://www.ecrypt.eu.org/stream*.

[Max05b] A. Maximov. Two linear distinguishing attacks on VMPC and RC4A and weakness of RC4 family of stream ciphers. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 342–358. Springer-Verlag, 2005.

[Max06] A. Maximov. Cryptanalysis of the Grain family of stream ciphers. In F. Lin, D. Lee, B. Lin, S. Shieh, and S. Jajodia, editors, *ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*, pages 283–288. ACM, 2006.

[McE87] R. J. McEliece. *Finite Fields for computer scientists and engineers*. Kluwer Academic Publishers, 1987.

[Mih96] M. Mihaljevic. A faster cryptanalysis of the self-shrinking generator. In J. Pieprzyk and J. Seberry, editors, *First Australasian Conference on Information Security and Privacy ACISP'96*, volume 1172 of *Lecture Notes in Computer Science*, pages 182–189. Springer-Verlag, 1996.

[MJ05] A. Maximov and T. Johansson. Fast computation of large distributions and its cryptographic applications. In B. Roy, editor, *Advances in Cryptology—ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 313–332. Springer Verlag, 2005.

[MKH+06] D. Moon, D. Kwon, D. Han, J. Lee, G. H. Ryu, D. W. Lee, Y. Yeom, and S. Chee. T-function based stream cipher TSC-4. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/024, 2006. *http://www.ecrypt.eu.org/stream*.

[MPC04] W. Meier, E. Pasalic, and C. Carlet. Algebraic attacks and decomposition of Boolean functions. In *Advances in Cryptology—EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 474–491. Springer-Verlag, 2004.

[MS88] W. Meier and O. Staffelbach. Fast correlation attacks on stream ciphers. In C.G. Günter, editor, *Advances in Cryptology—EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, pages 301–316. Springer-Verlag, 1988.

[MS89] W. Meier and O. Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1(3):159–176, 1989.

[MS94] W. Meier and O. Staffelbach. The self-shrinking generator. In A. De Santis, editor, *Advances in Cryptology—EUROCRYPT'94*, volume 905 of *Lecture Notes in Computer Science*, pages 205–214. Springer-Verlag, 1994.

[MS01] I. Mantin and A. Shamir. Practical attack on broadcast RC4. In M. Matsui, editor, *Fast Software Encryption 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer-Verlag, 2001.

[MvOV97] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[NP07a] M. Naya-Plasencia. Cryptanalysis of Achterbahn-128/80. Fast Software Encryption 2007, 2007. Preproceedings.

[NP07b] M. Naya-Plasencia. Cryptanalysis of Achterbahn-128/80 with a new keystream limitation. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/004, 2007. *http://www.ecrypt.eu.org/stream*.

[Nyb91] K. Nyberg. Perfect nonlinear S-boxes. In *Advances in Cryptology—EUROCRYPT'91*, volume 547 of *Lecture Notes in Computer Science*, pages 378–386. Springer-Verlag, 1991.

[Nyb93] K. Nyberg. Differentially uniform mappings for cryptography. In T. Helleseth, editor, *Advances in Cryptology—EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*, pages 55–64. Springer-Verlag, 1993.

[Oec03] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology—CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer-Verlag, 2003.

[PLL+91] B. Preneel, W. Van Leekwijck, L. Van Linden, R. Govaerts, and J. Vandewalle. Propagation characteristics of Boolean functions. In *Advances in Cryptology—EUROCRYPT'90*, volume 473 of *Lecture Notes in Computer Science*, pages 161–173. Springer-Verlag, 1991.

[PP03] S. Paul and B. Prenel. Analysis of non-fortuitous predictive states of the RC4 keystream generator. In T. Johansson and S. Maitra, editors, *Progress in Cryptology—INDOCRYPT 2003*, volume 2904 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 2003.

[PP04a] S. Paul and B. Preneel. A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 2004.

[PP04b] S. Paul and B. Preneel. A new weekness in the RC4 keystream generator. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 2004.

[RH02] G. Rose and P. Hawkes. On the applicability of distinguishing attacks against stream ciphers. Available at *http://eprint.iacr.org/2002/142*, 2002.

[RHPdV05] G. Rose, P. Hawkes, M. Paddon, and M. Wiggers de Vries. Primitive specification for SSS. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/028, 2005. *http://www.ecrypt.eu.org/stream*.

[Rom92] S. Roman. *Coding and Information Theory*. Graduate Texts in Mathematics. Springer-Verlag, 1992.

[Rue86] R.A. Rueppel. Correlation immunity and the summation generator. In H.C. Williams, editor, *Advances in Cryptology—CRYPTO'85*, volume 218 of *Lecture Notes in Computer Science*, pages 260–272. Springer-Verlag, 1086.

[Saa02] M-J.O. Saarinen. A time-memory tradeoff attack against LILI-128. In J. Daemen and V. Rijmen, editors, *Fast Software Encryption 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 231–236. Springer-Verlag, 2002.

[Sar03] P. Sarkar. Hiji-bij-bij: a new stream cipher with a self-synchronizing mode of operation. In *Progress in Cryptology—INDOCRYPT 2003*, volume 2904 of *Lecture Notes in Computer Science*, pages 36–51. Springer-Verlag, 2003.

[Sha49] C.E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 27:656–715, 1949.

[Sie84] T. Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, 30:776–780, 1984.

[Sie85] T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on Computers*, 34:81–85, 1985.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[WP07] H. Wu and B. Preneel. Differential cryptanalysis of the stream ciphers Py, Py6 and Pypy. The State of the Art of Stream Ciphers, Workshop Record, SASC 2007, Bochum, Germany, January 2007.

[WT86] A. F. Webster and S. E. Tavares. On the design of S-boxes. In H. C. Williams, editor, *Advances in Cryptology—CRYPTO'85*, pages 523–534. Springer-Verlag, 1986.

[ZKL01] E. Zenner, M. Krause, and S. Lucks. Improved cryptanalysis of the self-shrinking generator. In V. Varadharajan and Y. Mu, editors, *Australasian Conference on Information Security and Privacy ACISP'01*, volume 2119 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 2001.

[Zol04] B. Zoltak. VMPC one-way function and stream cipher. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 210–225. Springer-Verlag, 2004.