Some Results on Distinguishing Attacks on Stream Ciphers

Englund, Håkan

2007

Some Results on Distinguishing Attacks on Stream Ciphers

Håkan Englund

LUND UNIVERSITY

Ph.D. Thesis, December 14, 2007

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

ISBN: 91-7167-046-7
ISRN: LUTEDX/TEIT-07/1042-SE

Abstract

Stream ciphers are cryptographic primitives that are used to ensure the privacy of a message that is sent over a digital communication channel. In this thesis we will present new cryptanalytic results for several stream ciphers.

The thesis provides a general introduction to cryptology, explains the basic concepts, gives an overview of various cryptographic primitives and discusses a number of different attack models.

The first new attack given is a linear correlation attack in the form of a distinguishing attack. In this attack a specific class of weak feedback polynomials for LFSRs is identified. If the feedback polynomial is of a particular form the attack will be efficient.

Two new distinguishing attacks are given on classical stream cipher constructions, namely the filter generator and the irregularly clocked filter generator. It is also demonstrated how these attacks can be applied to modern constructions. A key recovery attack is described for LILI-128 and a distinguishing attack for LILI-II is given.

The European network of excellence, called eSTREAM, is an effort to find new efficient and secure stream ciphers. We analyze a number of the eSTREAM candidates. Firstly, distinguishing attacks are described for the candidate Dragon and a family of candidates called Pomaranch. Secondly, we describe resynchronization attacks on eSTREAM candidates. A general square root resynchronization attack which can be used to recover parts of a message is given. The attack is demonstrated on the candidates LEX and Pomaranch. A chosen IV distinguishing attack is then presented which can be used to evaluate the initialization procedure of stream ciphers. The technique is demonstrated on four candidates: Grain, Trivium, Decim and LEX.

Contents

Abstract **iii**

Preface **xi**

1 Introduction **1**
1.1 A Modern Digital Communication System 2
1.2 Cryptography 3
1.2.1 Unkeyed Cryptography 5
1.2.2 Symmetric-Key Cryptography 6
1.2.3 Asymmetric-Key Cryptography 10
1.3 Cryptanalysis 11
1.3.1 Attack Scenarios 12
1.3.2 Success of the Attack 12
1.4 Thesis Outline 13

2 Building Stream Ciphers **15**
2.1 Stream Ciphers . . . 16
2.1.1 Synchronous Stream Ciphers . . . 17
2.1.2 Self-Synchronizing Stream Ciphers . . . 19
2.2 The Building Blocks of Stream Ciphers . . . 20
2.2.1 Linear Feedback Shift Registers . . . 21
2.2.2 Nonlinear Feedback Shift Registers . . . 24
2.2.3 Boolean Functions . . . 24
2.2.4 S-boxes . . . 27
2.3 Classical Stream Ciphers . . . 28
2.3.1 Nonlinear Combination Generator . . . 28
2.3.2 Nonlinear Filter Generator . . . 29
2.3.3 Clock-Controlled Generators . . . 30
2.4 Summary . . . 31

3 Stream Cipher Cryptanalysis **33**
3.1 Tools for Cryptanalysis . . . 34
3.1.1 The Birthday Problems . . . 34
3.1.2 How to find Low Weight Multiples of Binary LFSRs . . . 35
3.1.3 Hypothesis Testing . . . 37
3.2 Brute Force Attacks . . . 43
3.3 Correlation Attacks . . . 44
3.4 Distinguishing Attacks . . . 45
3.4.1 The Random and the Cipher Distribution are Known . . . 48
3.4.2 Only The Random Distribution is Known . . . 48
3.5 Time Memory Data Trade-Off Attacks . . . 49
3.6 Resynchronization Attacks . . . 51
3.7 Algebraic Attacks . . . 52
3.8 Summary . . . 53

4 Correlation Attacks Using a New Class of Weak Feedback Polynomials **55**
4.1 A Basic Distinguishing Attack . . . 56
4.2 A More General Distinguishing Attack Using Correlated Vectors . . . 58
4.3 The Parameters in the Attack . . . 62
4.3.1 The Structure of $g_i(x)$ . . . 62
4.3.2 The Vector Length . . . 64
4.3.3 The Number of Groups . . . 64
4.4 Finding Multiples of the Characteristic Polynomial of a Desired Form . . . 64
4.5 Comparing the Proposed Attack with a Basic Distinguishing Attack . . . 66

4.6 Summary 67

5 A New Simple Technique to Attack Filter Generators and Related Ciphers 69
5.1 Building a Distinguisher 70
5.2 Using More than one Parity Check Equation 74
5.3 The Weight Three Attack 76
5.4 A Key Recovery Attack on LILI-128 77
5.4.1 Description of LILI-128 78
5.4.2 The Attack Applied on LILI-128 79
5.4.3 Results with a Weight Three Multiple 80
5.4.4 Results with a Weight Four Multiple 81
5.4.5 Results with a Weight Five Multiple 81
5.4.6 The Weight Three Attack 81
5.4.7 Summary of Attack on LILI-128 82
5.5 Summary 82

6 Cryptanalysis of Irregularly Clocked Filter Generators 85
6.1 An Efficient Distinguisher 86
6.1.1 Finding a Low Weight Multiple 87
6.1.2 Calculating the Correlation of h for a Weight Three Recursion 87
6.1.3 The Positions of the Windows 88
6.1.4 Determining the Size of the Windows 88
6.1.5 Estimating the Number of Bits We Need to Observe . . 89
6.1.6 Complexity of Calculating the Samples 90
6.1.7 Hypothesis Testing 91
6.1.8 Summary of the Attack 91
6.2 Possible Improvements Using Vectorial Samples 91
6.2.1 An Efficient Distinguisher Using Vectors 92
6.2.2 Considering Words In The Undecimated Sequence . . 94
6.3 Cryptanalysis of LILI-II 96
6.3.1 Simulations on a Scaled Down Version of LILI-II 98
6.3.2 Results 99
6.4 Summary 100

7 Distinguishing Attacks on the Pomaranch Family of Stream Ciphers 101
7.1 Description of Pomaranch 102
7.1.1 Pomaranch Version 1 104
7.1.2 Pomaranch Version 2 104
7.1.3 Pomaranch Version 3 104
7.2 Previous Attacks on the Pomaranch Stream Ciphers 105

7.3 Properties Used in the Attack . . . 105
7.3.1 The Period of Registers . . . 106
7.3.2 The Filter Function . . . 106
7.3.3 Linear Approximations of jump registers . . . 107
7.4 Attacking Different Versions of Pomaranch . . . 108
7.4.1 One Type of Registers with Linear Filter Function . . . 109
7.4.2 Different Registers with Linear Filter Function . . . 110
7.4.3 Nonlinear Filter Function . . . 111
7.5 Attack Complexities for the Existing Versions of the Pomaranch Family . . . 112
7.5.1 Pomaranch Version 1 . . . 112
7.5.2 Pomaranch Version 2 . . . 113
7.5.3 Pomaranch Version 3 . . . 113
7.6 Summary . . . 115

8 Find the Dragon **117**
8.1 A Description of Dragon . . . 118
8.2 Linear Approximation of the Function F . . . 120
8.3 Constructing a Distinguisher . . . 121
8.4 Calculation of the Noise Distribution . . . 123
8.4.1 Truncate the word size . . . 124
8.4.2 Approximate $\boxplus$ with $\oplus$ . . . 125
8.5 Attack Scenarios . . . 126
8.6 Summary . . . 126

9 A Square Root Resynchronization Attack **129**
9.1 Classical Distinguishing Attack on OFB mode . . . 130
9.2 New Attack Scenario . . . 130
9.2.1 Distinguished Points . . . 133
9.3 Analysis of Leak Extraction (LEX) . . . 134
9.4 Analysis of Pomaranch . . . 135
9.5 Summary . . . 136

10 A Framework for Chosen IV Statistical Analysis of Stream Ciphers 137
10.1 Boolean Functions . . . 139
10.2 A Framework for Chosen IV Statistical Attacks . . . 140
10.3 A Generalized Approach . . . 140
10.4 The Monomial Distribution Test . . . 141
10.5 The Maximal Degree Monomial Test . . . 142
10.5.1 Other Possible Tests . . . 144
10.6 Experimental Results . . . 144
10.6.1 Grain-128 . . . 145
10.6.2 Trivium . . . 148

10.6.3 Decim 150
10.6.4 Lex 152
10.7 Summary 153

11 Concluding Remarks **155**

A Variance of the number of combinations **157**

B Notations **161**

Bibliography **163**

Preface

Parts of this thesis have been presented at various conferences and been published in journals. The thesis is based on material from eight previously published papers. In all papers all authors have contributed on equal terms. The papers are the following:

- H. ENGLUND, M. HELL AND T. JOHANSSON. "Correlation Attacks Using a New Class of Weak Feedback Polynomial". 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.
- H. ENGLUND AND T. JOHANSSON. "A New Simple Technique to Attack Filter Generators and Related Ciphers". In H. Handschuh and M.A. Hasan, editors, *Selected Areas in Cryptography - 2004, Waterloo, Canada*, volume 3357 of *Lecture Notes in Computer Science*, pages 39–53, Springer-Verlag, 2005.
- H. ENGLUND AND A. MAXIMOV. "Attack the Dragon". In S. Maitra, V. Madhavan and R. Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005, Bangalore, India*, volume 3797 of *Lecture Notes in Computer Science*, pages 130–142, Springer-Verlag, 2005.
- H. ENGLUND AND T. JOHANSSON. "A New Distinguisher for Clock Controlled Stream Ciphers". In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption - 2005, Paris, France*, volume 3557 of *Lecture Notes in Computer Science*, pages 181–195, Springer-Verlag, 2005.

- H. ENGLUND AND T. JOHANSSON. "Three Ways to Mount Distinguishing Attacks on Irregularly Clocked Stream Ciphers". *International Journal of Security and Networks*, volume 1, Nos. 1/2, 2006, pages 95–102, Inderscience Enterprises Ltd, 2006.
- H. ENGLUND, M. HELL AND T. JOHANSSON. "Two General Attacks On Pomaranch-like Keystream Generators". In A. Biryukov, editor, *Fast Software Encryption - 2007, Luxembourg*, volume 4593 of *Lecture Notes in Computer Science*, pages 274–289, Springer-Verlag, 2007.
- 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.
- H. ENGLUND, T. JOHANSSON AND M. SÖNMEZ TURAN. "A Framework for Chosen IV Statistical Analysis of Stream Ciphers". In K. Srinathan, C.P. Rangan and M. Yung, editors, *Progress in Cryptology - INDOCRYPT 2007, Chennai, India*, volume 4859 of *Lecture Notes in Computer Science*, pages ??–??, Springer-Verlag, 2007.

Acknowledgments

Firstly, I would like to thank my supervisor Thomas Johansson for being a superb supervisor. Always sparkling with new ideas, always pushing me on when I need pushing. He has always had time for me and he has been a great guide through this journey.

Secondly, I would like to thank my fellow Ph.D. students at the department. Fredrik, Kora, Maja, Marcus, Suleyman and Thomas, you have all been great friends. Especially, I would like to thank Sasha for always making our trips so much more exciting; and Martin for being there to save me when Sasha's ideas got too crazy and for being a great friend.

I would also like to express my gratitude to all the colleagues at the former IT department for alway being helpful and for creating a nice working environment.

I would like to thank my parents for being the best parents in the world, and also my sister for always understanding and being the best possible friend. Finally, I would like to thank my wonderful Jessica, thank you for always encouraging me when things get tough. You make every day into a joyful one, even the rainiest day seem sunny when I am with you.

1

Introduction

The word "cryptology" comes from the Greek words *kryptos*, which means hidden, and *logos*, which means word. In the early days, cryptology mainly consisted in writing secret messages. In modern days, cryptology has grown to include many other topics, such as integrity protection, authentication and digital watermarking. Cryptology is indeed a very old subject; ever since the beginning of writing, there has been a need for secret communication. Already 4000 years ago, the Egyptians began modifying hieroglyphs on the graves of kings and rulers. Throughout history, the use and development of cryptologic algorithms have been driven primarily by government agencies and by the military. The broad community has always regarded cryptology as a mysterious field. In the Middle Ages it was even considered to be a black art. Today, with the exploding development and spreading of the Internet, secret and safe communication has become important to everybody, and cryptology is hopefully not as scary anymore.

Even though most people are unaware of it, they probably come into contact with cryptology every day, e.g. when they access their Internet bank, use a bank-ID to shop on the Internet, use a mobile phone to make a call, watch a DVD or digital television or file their tax return over the Internet.

However, cryptology on its own cannot solve all problems; awareness of how to use cryptographic methods is very important. Lately, various media have paid much attention to phishing and pharming attacks directed at Internet banks. These attacks circumvent all cryptographic protection by

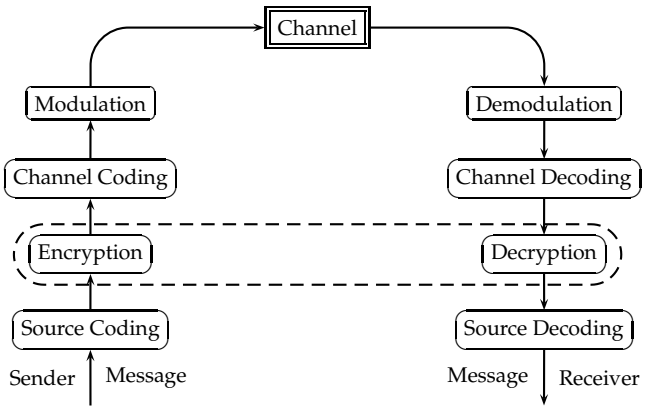


Figure 1.1: The place of cryptography in a digital communication system.

using social engineering to acquire secrets such as passwords or secret keys. In order to prevent such attacks, it is important to teach people the proper use of cryptographic solutions.

We will begin our introduction to the area of cryptology by describing the components of a modern digital communication system.

1.1 A Modern Digital Communication System

Most communication systems today are digital, e.g., the Internet and mobile phone networks. Assume that a digital message is to be sent over a channel. The channel might be a wire in the case of Internet, or the air in the case of mobile phones. A number of problems immediately present themselves. How do we prevent unintended reading and manipulation of the message? What happens if the channel introduces errors into the message? How do we transform a digital message to an analog signal that can be transferred over an analog channel? Figure 1.1 shows a block diagram of a digital communication system that aims to solve these problems. In the figure, the blocks that involve cryptology are marked with a dashed circle.

- *Source coding/Source decoding*. A message, e.g., a picture or a text, typically contains a lot of redundancy. Source coding removes such re-

dundancy, i.e., the message is compressed. Source decoding reverses the compression.

- *Encryption/Decryption*. After compression, encryption is performed to protect the secrecy of the message. Additional functions, such as integrity protection, may also be introduced here. The operation of decryption is the inverse of encryption.

- *Channel coding/Channel decoding*. When a message is transmitted over a channel, transmission errors may occur. In order to be able to detect or correct these errors, one adds new redundancy to the message, which gives the message a certain structure. In the channel decoding, this structure is used to detect or correct imposed errors.

- *Modulation/Demodulation*. The symbols are transformed into an analog signal suitable for transmission over the channel. Examples of information carriers are amplitude, frequency and the phase of the signal. The demodulation at the receiver end of the channel transforms the analog signal back into a digital signal.

It is important that source coding is performed before encryption, as redundancy in a message facilitates attacks. For the same reason it is important to perform encryption before channel coding, as channel coding adds new redundancy to the message.

Cryptology is a large field divided into two parts: *cryptography*, the art of designing cryptographic primitives and *cryptanalysis*, the art of trying to identify and exploit the weaknesses of cryptographic primitives.

1.2 Cryptography

As previously mentioned, during most of its history, cryptography has mainly provided algorithms for the encryption and decryption of messages. Modern cryptography provides services that can be classified into four categories.

- *Confidentiality* aims to assure that information cannot be accessed by anyone except the intended recipient. Confidentiality can be achieved by encrypting a message using a cipher. This category is the main topic discussed in this thesis.

- *Data integrity*, the goal is to assure that data are whole or unaltered, or that only authorized users are allowed to alter data. Popular ways to protect the integrity of data are to use *message authentication codes* (MACs) and *digital signatures*.

- *Authentication* tries to establish whether someone or something is who or what they claim to be. A simple and commonly used way to do authentication is to use a password. Only the correct user is supposed to know his password. A more complicated way is to use digital certificates constructed using public key cryptography.
- *Non-repudiation*, the concept of ensuring that a contract cannot later be denied by either of the parties involved, i.e., if you have signed a contract you should not be able to claim that it was someone else who signed it. This too can be achieved by using digital signatures together with digital certificates.

To be able to describe and analyze cryptographic algorithms we will need to introduce some notations. Let

- $\mathcal{A}$ be an *alphabet*, an alphabet is a set of symbols, in most cases the symbols will be b-bit strings, i.e., $\mathcal{A} = \{0,1\}^b$.
- $\boldsymbol{m}$ be a *message* (also called a *plaintext*) that we wish to transmit. A message usually consists of many symbols, say n_m symbols, and all symbols belong to the alphabet $\mathcal{A}_m$. The message is written as the vector

$$\boldsymbol{m} = (m_0, \ldots, m_{n_m-1}), \quad m_i \in \mathcal{A}, 0 \leq i \leq n_m - 1. \tag{1.1}$$

  Let $\mathcal{M}$ denote the set of all possible messages, i.e., the message space.
- K be the *key* used by a cryptographic algorithm. More specifically we denote the key used for encryption by K_e, and the key used for decryption by K_d. The key is seen as the vector of symbols

$$K = (k_0, \ldots, k_{n_K-1}), \quad k_i \in \mathcal{A}, 0 \leq i \leq n_K - 1, \tag{1.2}$$

  where every symbol is from an alphabet, $\mathcal{A}_K$. Let $\mathcal{K}$ denote the entire key space.
- IV be an *initialization vector*. An IV consist of n_{IV} symbols from some alphabet $\mathcal{A}_{IV}$

$$IV = (iv_0, \ldots, iv_{n_{IV}-1}), \quad iv_i \in \mathcal{A}_{IV}, 0 \leq i \leq n_{IV} - 1. \tag{1.3}$$

  Let $\mathcal{IV}$ denote the space of initialization vectors.
- $\boldsymbol{c}$ be the encrypted message, called the *ciphertext*. The ciphertext is written as a vector of symbols

$$\boldsymbol{c} = (c_0, \ldots, c_{n_c-1}), \quad c_i \in \mathcal{A}_c, 0 \leq i \leq n_c - 1. \tag{1.4}$$

  Let $\mathcal{C}$ denote the set of all possible ciphertexts.

- E_{K_e} be the *encryption function* that encrypts a message $\boldsymbol{m} \in \mathcal{M}$, using the encryption key K_e. Also, let e_{K_e} be the function that encrypts one message symbol $m \in \mathcal{A}$.
- D_{K_d} be the *decryption function* that decrypts the ciphertext $\boldsymbol{c} \in \mathcal{C}$, using decryption key K_d. Let d_{K_d} be the function that decrypts one ciphertext symbol $c \in \mathcal{A}$.

The encryption function is a mapping $E_{K_e} : \mathcal{M} \rightarrow \mathcal{C}$, the decryption function is the inverse mapping, $D_{K_d} : \mathcal{C} \rightarrow \mathcal{M}$. A requirement for the decryption function is that $\boldsymbol{m} = D_{K_d}\big(E_{K_e}(\boldsymbol{m})\big)$, for all possible pairs of keys and messages. Similarly, the function e_{K_e} is a mapping $e_{K_e} : \{0,1\}^b \rightarrow \{0,1\}^b$ transforming a plaintext symbol into a ciphertext symbol, i.e.,

$$c = e_{K_e}(m), \quad m \in \mathcal{A}_m \text{ and } c \in \mathcal{A}_c \tag{1.5}$$

where $\mathcal{A}_m$ and $\mathcal{A}_c$ are not necessarily the same alphabet. However, usually $\mathcal{A}_m = \mathcal{A}_c$, and in what follows we will assume that $\mathcal{A} = \mathcal{A}_m = \mathcal{A}_c$. The decryption function is the inverse mapping converting ciphertext symbols to plaintext symbols,

$$m = d_{K_d}(c), \quad m, c \in \mathcal{A}. \tag{1.6}$$

As for the encryption and decryption functions, a requirement is that $m = d_{K_d}\big(e_{K_e}(m)\big)$, for all possible pairs of keys and messages symbols.

A *cryptographic primitive* is a building block, i.e., an algorithm, that provides a cryptographic service. In the 19th century, Auguste Kerckhoffs formulated a number of important rules about the design of cryptographic primitives. His most famous rule says that the secrecy of the message, given the ciphertext, should depend entirely on the secrecy of the secret key. Many attempts have been made to keep primitives secret. However, in most cases the algorithms have leaked and many of the primitives have been found to be weak, e.g., the A5 algorithm used in GSM. In a well designed cryptographic algorithm, only the key needs to remain secret.

Cryptographic primitives are divided into three groups, *unkeyed primitives*, *symmetric-key primitives*, and *asymmetric-key primitives*.

1.2.1 Unkeyed Cryptography

In the family of unkeyed primitives we mainly find tools used for assuring message integrity and authentication, namely *hash functions*, as well as one-way permutations and random sequences. Hash functions are functions that take a message of arbitrary length and produce a fixed length digest, common output digest lengths are 160 and 256 bits. Desirable properties of a good hash function, denoted as $h(.)$, are

- *Ease of computation*. The hash function should be easy to compute.
- *Uniformity*. Hash values should be distributed uniformly over the entire output space.
- *Preimage resistance*. Given a hash-value y, it should be computationally infeasible to find a message $\boldsymbol{m} \in \mathcal{M}$ such that $h(\boldsymbol{m}) = y$.
- *2nd Preimage resistance*. Given a a message $\boldsymbol{m_1} \in \mathcal{M}$ and its corresponding hash value $h(\boldsymbol{m_1})$, it should be computationally infeasible to find $\boldsymbol{m_2} \in \mathcal{M}$ such that $\boldsymbol{m_1} \neq \boldsymbol{m_2}$ and $h(\boldsymbol{m_2}) = h(\boldsymbol{m_1})$.
- *Collision resistance*. It should be computationally infeasible to find any two distinct messages $\boldsymbol{m_1}, \boldsymbol{m_2} \in \mathcal{M}$ such that $h(\boldsymbol{m_1}) = h(\boldsymbol{m_2})$.

The research area of hash function cryptanalysis has been very active lately, and many primitives have been broken.

1.2.2 Symmetric-Key Cryptography

In symmetric-key cryptography the sender and the receiver share a secret key, K. This key is used for both encryption and decryption, i.e., $K_e = K_d = K$. Hence, the decryption algorithm is the inverse of the encryption algorithm, i.e., $\boldsymbol{m} = D_K\big(E_K(\boldsymbol{m})\big)$ for all $K \in \mathcal{K}$ and $\boldsymbol{m} \in \mathcal{M}$, and similarly $m = d_K\big(e_K(m)\big)$, for all $K \in \mathcal{K}$ and $m \in \mathcal{A}$.

The sender, in cryptology usually called Alice, wants to send a message to a receiver, usually called Bob. Assume that Alice and Bob share a secret key, K. Alice uses K to encrypt the message to produce the ciphertext symbol as $\boldsymbol{c} = E_K(\boldsymbol{m})$. The ciphertext is then sent to Bob over an insecure channel. Bob deciphers the ciphertext using the same secret key, and obtains the message $\boldsymbol{m} = D_K(\boldsymbol{c})$. The eavesdropper, Eve, is listening to the channel. In most scenarios, Eve tries to either recover the message or find the secret key that Alice and Bob use. Figure 1.2 describes the scenario considered when studying symmetric cryptology.

Symmetric cryptographic algorithms are in general much more efficient than asymmetric algorithms in both hardware and software. One major problem when using symmetric cryptography is how to exchange the secret key between the sender and the receiver. This problem can be solved by using asymmetric cryptography for the key exchange.

In the group of symmetric-key primitives we find three important types of primitives, *block ciphers*, *stream ciphers* and *message authentications codes* (MACs). We will start with block ciphers.

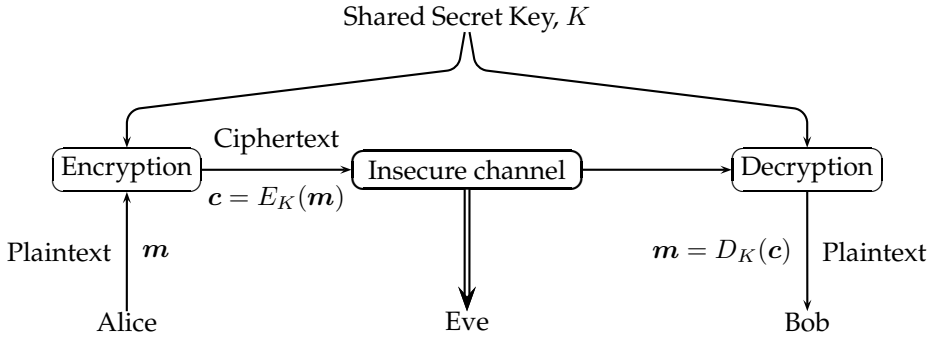


Figure 1.2: Alice sends an encrypted message to Bob using symmetric-key cryptography. Alice and Bob shares a secret key. Alice sends the ciphertext over an insecure channel to Bob. The adversary, Eve, has access to the ciphertext but not to the secret key.

1.2.2.1 Block Ciphers

A block cipher takes an n-bit plaintext block as input and permutes it into an n-bit ciphertext block, see Figure 1.3.

The integer n is called the *block size* of the cipher. A message $\boldsymbol{m}$ of length l bits is divided into $\lceil l/n \rceil$ blocks, each with a length of n bits, i.e., $\mathcal{A} = \{0,1\}^n$. We can then write the message as the vector $\boldsymbol{m} = (m_0, \ldots, m_{\lceil l/n \rceil})$, where $m_i \in \mathcal{A}, 0 \leq i \leq \lceil l/n \rceil$. If l is not a multiple of n, some bits, e.g., zeros, have to be added to the message until the total length is a multiple of n. This procedure is called *padding*. Block sizes of 64 and 128 bits are common. The permutation is key dependent and one-to-one, the decryption of the ciphertext is performed as the inverse permutation.

As a block cipher is a deterministic permutation, plaintext blocks that occur several times in the message will produce the same ciphertext blocks. A block cipher used in this fashion is said to be used in *electronic code book mode* (ECB mode), see Figure 1.3. As ECB leaks information for repeated message blocks, some other *mode of operation* should be used for encryption with a block cipher. Common modes are

- *Electronic code book mode* (ECB mode). Encryption is performed as $c_t = e_K(m_t)$ and the decryption as $m_t = d_K(c_t)$.
- *Cipher block chaining mode* (CBC mode). Encryption is performed as

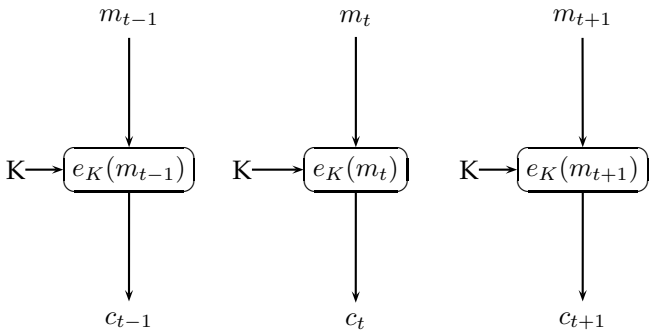


Figure 1.3: A block cipher in ECB mode. Each message block is encrypted independently.

$c_t = e_K(m_t \oplus c_{t-1})$, with $c_{-1} = IV$, and decryption as $m_t = d_K(c_t) \oplus c_{t-1}$.

- *Counter mode* (CTR mode). The block cipher is turned into a stream cipher. Counter mode utilizes a counter, let k denote the initial value of the counter. The encryption is performed as $c_t = m_t \oplus e_K(IV||k + t),\ t \geq 0$, where $IV||k$ is the concatenation of the initialization vector and the counter. Similarly, the decryption is performed as $m_t = c_t \oplus d_K(IV||k + t)$.
- *Output feedback mode* (OFB mode). Turns the block cipher into a stream cipher. A keystream $z_0, z_1, \ldots$ is generated as $z_t = e_K(z_{t-1})$, with $z_{-1} = IV$, and the encryption is performed as $c_t = m_t \oplus z_t$. Decryption is done as $m_t = c_t \oplus z_t$, where z_t is generated as in the encryption.

The most famous block ciphers are the Data Encryption Standard (DES), and the Advanced Encryption Standard (AES), which is based on the primitive called Rijndael [DR02].

Block ciphers are used in many areas of cryptology besides confidentiality protection. They can, for example, be used to build hash functions, stream ciphers and message authentication codes.

1.2.2.2 Stream Ciphers

A stream cipher is a primitive that operates on individual symbols, where the encryption transformation changes for each symbol. A stream cipher

tries to imitate the behavior of the *one-time pad* (OTP).

The one-time pad uses a key with the same length as the message. Assume we would like to encrypt a message $\boldsymbol{m} = (m_0, \ldots, m_{n-1})$ of length n. The OTP key then also has to be of length n, i.e., $K = (k_0, \ldots, k_{n-1})$, where $m_i,\ k_i, 0 \leq i \leq n-1$ are elements from the same alphabet. The message and the key are combined using the XOR operation in order to produce the ciphertext, i.e.,

$$c_t = m_t \oplus k_t,\ 0 \leq t \leq n-1. \tag{1.7}$$

However, for such a system to be a true OTP, the key symbols $k_i,\ 0 \leq i \leq n-1$ must be completely independent, random and used only once. If this is the case, it can be shown that the OTP offers *perfect security*. Perfect security means, among other things, that even if an adversary has infinite computing power, he will never be able to recover the message. As every guess on a key symbol, k_i, is equally probable, we can receive any possible plaintext symbol, m_i, with equal probability from a ciphertext symbol c_i. Hence, all messages are equally probable.

However, it is not practical to generate and distribute truly random keystreams of the same length as the message. Hence, a stream cipher instead produces a pseudo-random keystream sequence from a much shorter key.

An advantage of stream ciphers compared to block ciphers is that error propagation is much smaller. Stream ciphers have no need for padding as they operate on individual symbols. Stream ciphers can be made much faster than block ciphers in both hardware and software. For constrained environments they can also be made smaller and less energy consuming in hardware. However, there is a wide variety of stream cipher designs, whereas block cipher designs usually have a shared structure. Stream cipher security is not as well studied as block cipher security, and the variability in design ideas makes general analysis difficult.

Stream ciphers are the main topic of this thesis and they will be discussed more thoroughly in later chapters. For a general description of stream ciphers, see Chapter 2.

1.2.2.3 Message Authentication Codes

Message authentication codes (MACs) are key dependent symmetric primitives used for both integrity and authenticity protection. Unlike digital signatures based on hash functions (see Section 1.2.3) MACs can only be verified by the intended receiver (or by anyone in possession of the secret key). A MAC of a message, $\boldsymbol{m}$, is denoted by $MAC_K(\boldsymbol{m})$ for a secret key K.

Assume that Alice and Bob share a secret key, K, and Alice wants to send an integrity protected message to Bob. Alice calculates the MAC and

sends the pair $(\boldsymbol{m}, MAC_K(\boldsymbol{m}))$ over an insecure channel. When receiving the pair, Bob also calculates $MAC_K(\boldsymbol{m})$ and compares the result with the MAC included in the message. If they are the same Bob accepts the message and can be certain that the message has been unaltered and is indeed from Alice. If the MAC algorithm is designed properly, there is a very small probability that, the impostor, Eve, will be able to create other valid pairs $(\boldsymbol{m'}, MAC_K(\boldsymbol{m'}))$ without first finding the secret key.

An attack on a MAC algorithm where valid pairs $(\boldsymbol{m'}, MAC_K(\boldsymbol{m'}))$ can be created without knowledge of K is called *existential forgery*. Existential forgery should be computationally infeasible for a good MAC algorithm.

There are several methods for message authentication code designs. Some primitives are based on block ciphers and hash functions.

1.2.3 Asymmetric-Key Cryptography

Asymmetric cryptography, also called public key cryptography, was invented by Diffie and Hellman [DH76] in 1976. In asymmetric cryptography each user has two different keys, a *public key* used for encryption denoted as K_e, and a *private key* used for decryption denoted as K_d, with $K_e \neq K_d$. The private keys are kept secret, but the public keys are widely distributed.

If Alice wants to send a message, $\boldsymbol{m} \in \mathcal{M}$, to Bob confidentially, she uses Bob's public key to encrypt the message, i.e., $\boldsymbol{c} = E_{K_e}(\boldsymbol{m})$. She then sends the ciphertext to Bob. Eve, the eavesdropper cannot decipher the ciphertext, as she does not have access to Bob's private key. Only Bob can recover the message from the ciphertext by calculating $\boldsymbol{m} = D_{K_d}(\boldsymbol{c})$, see Figure 1.4.

Asymmetric cryptography can also be used for *digital signatures*, which are used to prove the integrity and authenticity of a message. If Bob wants to send a signed message to Alice, Bob uses his private key to sign the message, i.e., $s = E_{K_d}\big(h(\boldsymbol{m})\big)$, where $h(.)$ is a nonlinear function and s is a signature. He then sends the pair $(\boldsymbol{m}, s)$ to Alice. Alice verifies the signature of a received pair $(\boldsymbol{m'}, s)$ by first calculating the intermediate value b as $b = D_{K_d}(s)$ using Bob's public key. She then also produces $h(\boldsymbol{m'})$ from $\boldsymbol{m'}$ using the same nonlinear function, $h(.)$, as Bob, and finally checks if $h(\boldsymbol{m'}) = b$. If the nonlinear function $h(.)$ is chosen properly, the eavesdropper Eve is unlikely to be able to produce a valid pair $(\boldsymbol{m''}, s'')$. Usually a hash function is used for $h(.)$.

Asymmetric primitives are based on mathematical trapdoor one-way problems, i.e., problems that are easy to calculate in one direction, and computationally infeasible to reverse without knowing the secret trapdoor. The *discrete logarithm* is an example of a mathematical one-way function that is frequently used in asymmetric cryptography. Let G be a finite multiplicative cyclic group with n elements, and let b be a generator of G. Then every

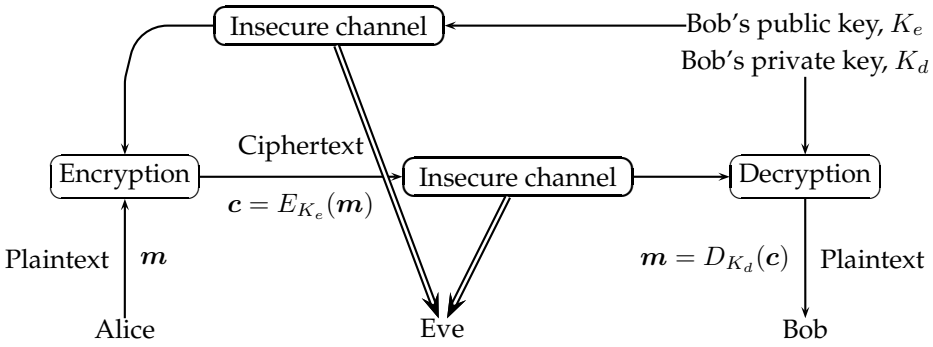


Figure 1.4: Alice uses public key cryptography to send an encrypted message to Bob over an insecure channel. The adversary, Eve, has access to both Bob's public key and the ciphertext.

element $g \in G$ can be written as $g = b^k$ for some integer k. The discrete logarithm problem in G, i.e., to compute

$$k = \log_b(g), \tag{1.8}$$

is believed to be a computationally hard problem for large groups, i.e., for large n. The first proposed asymmetric primitive, the *Diffie Hellman key exchange protocol* is based on this problem.

Another one-way problem believed to be difficult, is the task of *integer factorization*. For a large integer $N = p \cdot q$, where p and q are large prime numbers, it is computationally difficult to find p and q. The very popular primitive *RSA* is based on this problem.

Asymmetric cryptography is computationally inefficient compared to symmetric cryptography. However, asymmetric-key primitives can be used to solve the problem of secret key-exchange for symmetric cryptography, which was discussed in Section 1.2. Systems that uses both asymmetric-key and symmetric-key primitives are sometimes called *hybrid cryptographic systems* or *digital envelopes*.

1.3 Cryptanalysis

When attacking a cryptographic primitive, there are several kinds of attacks and attack scenarios to choose from and some of them will be classified in

what follows. We distinguish between the *attack scenario*, which states how much power the adversary has, and the *success of the attack*, which classifies attacks according to the amount of information that is revealed in the attack.

1.3.1 Attack Scenarios

In this thesis we will consider a number of different attack scenarios, i.e., different assumptions about the power/knowledge of an adversary.

- *Ciphertext-only*. This is the first scenario that comes to mind. In this scenario, the cryptanalyst has access to parts of the ciphertext.
- *Known-plaintext*. In a known-plaintext attack the adversary knows both the plaintext and the corresponding ciphertext.
- *Known-IV-known-plaintext*. In this type of attack, the adversary knows one or several initialization vectors that are used in the encryption. The adversary also knows the ciphertext and plaintext produced for each IV.
- *Chosen-IV-known-plaintext*. In this scenario the adversary can choose one or several IVs that will be used in the encryption. The adversary is assumed to know the plaintext and the corresponding ciphertext for each IV.

For the sake of completeness we will list some other common attack scenarios:

- *Chosen-plaintext.* In this scenario, the adversary can choose any plaintext he wants and obtain the corresponding ciphertext.
- *Adaptive chosen-plaintext.* A chosen plaintext attack in which the choice of plaintexts depend on the previously received ciphertexts.
- *Chosen-ciphertext.* Similar to a chosen-plaintext attack, however in this scenario the adversary can choose any ciphertext and obtain the corresponding plaintext.
- *Adaptive chosen-ciphertext.* A chosen-ciphertext attack in which the choice of ciphertexts depend on the previously received plaintexts.

1.3.2 Success of the Attack

The ultimate goal for a cryptanalyst is of course to recover the secret key used in an encryption. Knowing the secret key gives an adversary the power to recover all old and future messages encrypted with that key. There are

also several other kinds of attacks that might give the adversary important information without revealing the entire secret key. In this context, we will identify five ways in which an attack can succeed:

- *Total break*. The secret key is recovered.
- *Global deduction*. An adversary finds an algorithm, A, which is functionally equivalent to E_{K_e} (or D_{K_d}). A can be used for encryption and decryption without knowledge of K_e and K_d.
- *Instance deduction*. An adversary is able to produce not previously known plaintext symbols of an intercepted ciphertext, or vice versa.
- *Information deduction*. An adversary gains (Shannon) information about the secret key, the plaintext or the ciphertext which he did not have before the attack.
- *Distinguishing algorithm*. By applying this algorithm, an adversary is able to detect statistical anomalies that should not be present in an ideal cipher.

These forms of attacks form a hierarchy, i.e., a total break can be used for global deduction, global deduction can be used for instance deduction and so on.

From here on, we will mainly focus on distinguishing attacks.

1.4 Thesis Outline

The main topic of this thesis is cryptanalysis of stream ciphers, mainly distinguishing attacks. The thesis is organized as follows.

Chapter 2 gives definitions related to stream ciphers, and a general introduction to the problem of designing stream cipher primitives. The chapter describes the common building blocks used in stream ciphers, and gives some examples of classical stream ciphers.

Chapter 3 focuses on the cryptanalysis of stream ciphers. It presents a number of tools that are useful for analyzing stream ciphers and describes some common attacks.

Chapter 4 describes a linear correlation attack. In this attack a specific class of weak feedback polynomials of LFSRs is identified. If the feedback polynomial is of a particular form, there is an efficient attack in the form of a distinguishing attack.

Chapter 5 focuses on cryptanalysis of a classical stream cipher design, called a filter generator. A very simple distinguishing attack on filter generators is described. It is also demonstrated how this attack can be converted into a key recovery attack on a stream cipher called LILI-128.

Chapter 6 describes an efficient method for cryptanalysis of irregularly clocked filter generators. The idea of the attack is to position windows around the estimated position of linear recurrence relationship members. Some possible improvements of the attack are also discussed. A distinguishing attack based on the proposed technique is then demonstrated on a stream cipher called LILI-II (successor of LILI-128).

Chapter 7 focuses on finding distinguishing attacks directed at ciphers that belong to a family of stream ciphers called the Pomaranch family. After analyzing the existing ciphers in this family, we give some design guidelines for future Pomaranch ciphers.

Chapter 8 deals with linear distinguishing attacks on another eSTREAM candidate, called Dragon. By using linear approximations of nonlinear components, a biased expression can be found among the keystream output from the cipher. This expression is used to create a distinguisher.

Chapter 9 introduces a resynchronization attack applicable to all stream ciphers in which there is at least one part of the state that is unaffected by the IV. This attack is different from an ordinary distinguishing attack in the sense that it can be used to recover information about plaintexts.

Chapter 10 discusses certain problems concerning the initialization procedure of stream ciphers. We will describe a chosen IV distinguishing attack. In this attack the stream cipher is modeled as a black box, and by using many initializations we can write the keystream bits as a function of some bits of the initialization vector. The properties of this function are then compared to the expected properties of a random Boolean function.

Chapter 11 summarizes the contributions of the thesis and draws some conclusions.

2

Building Stream Ciphers

A stream cipher is a cryptographic primitive that operates on individual symbols, instead of on entire blocks as block ciphers do. In stream ciphers, the symbols are often chosen to be of size 1, 8 or 32 bits, depending on the application. In software oriented constructions it is often desirable choose a symbol size that coincides with the word size of the CPU of the system. This enables efficient usage of available operations on the CPU. In restricted hardware environments, bit oriented stream ciphers offer a superior throughput compared to block ciphers and can be constructed with a much smaller gate cost. In protocols that use odd sized packages, block ciphers require padding as the encrypted block must be of the cipher's block size. Stream ciphers can be designed in such a way that no padding is required.

In 2004, the European Network of Excellence for Cryptology, ECRYPT, launched a project called eSTREAM, the project is to be completed by May 2008. The aim of the project is to collect a number of promising stream cipher proposals and eventually present a portfolio of stream ciphers that offer some advantages over the block cipher AES. Two profiles have been identified in which stream ciphers were believed to offer superior properties compared to block ciphers (especially AES), namely

- Profile 1: Very good performance in software, i.e., much faster than AES.

- Profile 2: Efficient hardware primitives, in terms of a small number of gates and low power consumption, compared to AES.

A major part of this thesis will investigate the security of some of the eSTREAM proposals.

The outline of this chapter is the following. Section 2.1 gives an introduction to stream ciphers, and describes synchronous and self-synchronizing stream ciphers. Section 2.2 presents four building blocks commonly used in stream cipher primitives; linear feedback shift registers, nonlinear feedback shift registers, Boolean functions and substitution boxes. Section 2.3 describes some classical stream ciphers that are often discussed in the literature. Section 2.4 is a summary of the chapter.

2.1 Stream Ciphers

A classical stream cipher can be seen as a function that takes a message, $\boldsymbol{m} \in \mathcal{M}$, and a key, $K \in \mathcal{K}$, as input. The output of the function is a ciphertext message, $\boldsymbol{c}$. Modern stream ciphers also include an *initialization vector* (IV) also called a *nonce*. An IV should only be used once, and is incorporated to enable the use of the same key for several keystreams. Key exchange procedures are often computationally costly, and we would like to minimize the number of such exchanges. Without the IV, a given key would always produce the same keystream after initialization. If used to produce two ciphertexts, this would leak information about the underlying plaintexts. IVs are public vectors, which can be sent unencrypted and are easily exchanged between sender and receiver. Two identical ciphers initialized with the same key but different IVs will produce different keystreams.

The interface of a modern stream cipher can be described as

$$\boldsymbol{c} = \text{cipher}(K, IV, \boldsymbol{m}). \tag{2.1}$$

A stream cipher operates in two phases:

(*i*) *Setup phase*. Usually, the state of the cipher is initialized with the key, K and possibly an initialization vector, IV. An initialization procedure is then performed to make sure that all IV bits and key bits are properly mixed and spread across the entire state. The internal state after the

(*ii*) *Encryption/decryption phase*. In this phase, the stream cipher generates keystream symbols as a function of an internal state and the secret key. This keystream symbol is combined with the message symbol to create a ciphertext symbol. A new internal state is determined as a

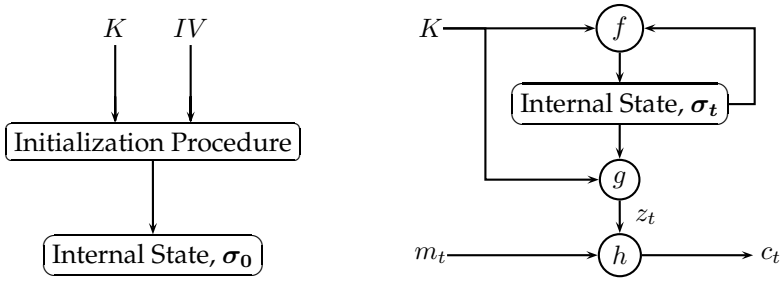
function of the current state, the key and possibly the ciphertext symbol created. Decryption is performed in a similar manner.

Stream ciphers are divided into two families, *synchronous stream ciphers* and *self-synchronizing stream ciphers*. Let us begin with synchronous stream ciphers, which is the most common type.

2.1.1 Synchronous Stream Ciphers

In a synchronous stream cipher, a keystream is produced independently of the plaintext, i.e., a synchronous stream cipher tries to imitate the one-time pad. Ideally, a stream cipher would produce a sequence that is truly random, but in real life true random keystream generators are hard to find. Stream ciphers are often included in the class of algorithms called *pseudo-random number generators*. The output from pseudo-random number generators is not truly random, as knowledge of the state (or the secret key) makes prediction of future sequence symbols trivial. However, without knowledge of the state (or the secret key) the stream of symbols should be indistinguishable from a true random sequence. Desirable properties of pseudo-random sequences are a long period and a uniform distribution.

A synchronous stream cipher can be described as a finite state machine, with a state and an update function. See Figure 2.1 for a general model of the setup and encryption phases of synchronous stream ciphers. In the setup



(a) Setup Phase (b) Encryption Phase

Figure 2.1: General structure of a synchronous stream cipher in setup phase and in encryption phase.

phase the state is initialized by an initialization procedure. This procedure can be designed as a dedicated algorithm entirely separated from the rest of

the cipher. However, in most cases the key and the IV are loaded into the state of the cipher, the cipher is then iterated a number of times, possibly the output of the cipher is feed back into the state. After the initialization procedure is performed, all state symbols should depend on all IV and key bits in a nonlinear way.

A synchronous stream cipher in encryption phase can be described by the following parts

- *Internal state*. Let $\boldsymbol{\sigma_t} = (\sigma_t^0, \ldots, \sigma_t^{n-1})$ denote the internal state at time, t, where $\sigma_t^i,\ 0 \leq i \leq n-1$ are the individual symbols stored in the state.

- *Next-state function*. From the current state, $\boldsymbol{\sigma_t}$, and the key, K, the next state function, denoted as f, produces the next state as

$$\boldsymbol{\sigma_{t+1}} = f(\boldsymbol{\sigma_t}, K). \tag{2.2}$$

- *Keystream function*. Let g denote the keystream function, the keystream function creates a new keystream symbol from the key, K, and the internal state, $\boldsymbol{\sigma_t}$, as

$$z_t = g(\boldsymbol{\sigma_t}, K). \tag{2.3}$$

- *Output function*. Let h denote the output function, the output function combines the message symbol, m_t, and the keystream symbol, z_t. The output is the ciphertext symbol, c_t,

$$c_t = h(m_t, z_t). \tag{2.4}$$

The decryption of synchronous stream ciphers is very easy. The receiver generates the same keystream as the sender, and applies the inverse of the output function to the keystream and the cipher text.

$$m_t = h^{-1}(c_t, z_t). \tag{2.5}$$

The output function, h, is usually chosen to be the simple XOR operation, i.e., $c_t = m_t \oplus z_t$. A synchronous stream cipher that uses the XOR operation as output function is usually called an *additive synchronous stream cipher*. Figure 2.2 illustrates an additive stream cipher viewed as a keystream generator. The inverse of the XOR operations is again the XOR operation, hence the decryption of an additive synchronous stream cipher is exactly the same process as its encryption.

The keystream generation in a synchronous stream cipher is independent of both the message and the ciphertext. A transmission error in one symbol will only lead to one symbol error in the plaintext, hence the error

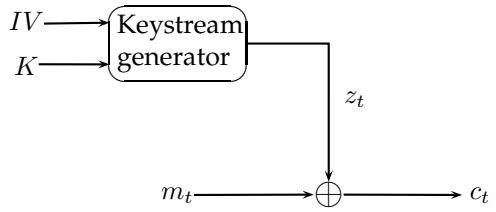


Figure 2.2: Additive stream cipher created from a keystream generator.

propagation of synchronous stream ciphers is very low. On the other hand, synchronous stream ciphers are very sensitive to synchronization errors. If the synchronization between sender and receiver is lost, all future symbols will be decrypted incorrectly. A common way to minimize synchronization errors is to have frequent reinitializations. One divides the data sequence into frames, where each frame has a unique, publicly known, initialization vector (or a frame number). If synchronization is lost, only data from the affected frame will be erroneously decrypted.

A designer of stream ciphers has many possibilities. One can choose a complex next-state function and use a very simple keystream function; an example is a block cipher in OFB mode. In this example, the next-state function is one application of the block cipher, and the keystream function is just the output of the entire state. In contrast, one can use a very simple next-state function and a complex keystream function, such as Salsa [Ber06], or a block cipher in counter mode. It is also possible to use a complex next-state function and a complex keystream function such as Dragon [CHM+05].

2.1.2 Self-Synchronizing Stream Ciphers

A self-synchronizing stream cipher generates keystream symbols as a function of the key and a fixed number of previous ciphertext symbols. Let n be the number of ciphertext symbols stored in the internal state. A self-synchronizing stream cipher can be described by the following equations:

$$\begin{aligned} \boldsymbol{\sigma_t} &= (c_{t-n}, \ldots, c_{t-1}), \\ z_t &= g(\boldsymbol{\sigma_t}, K), \\ c_t &= h(z_t, m_t), \end{aligned} \tag{2.6}$$

where g is the keystream function and h the output function as for synchronous stream ciphers.

Like synchronous stream ciphers, self-synchronizing stream ciphers operate in a setup phase and an encryption/decryption phase. During the

setup phase, the initialization vector, IV, is used to initialize the state $\boldsymbol{\sigma}$, and hence $z_0 = g(iv_0, \ldots, iv_{n-1}, K)$. During the encryption phase, the output function, h, combines a keystream symbol, z_t, with a message symbol, m_t. The output is the next ciphertext symbol, c_t. Once again, the output function, h, is usually the XOR operation. The encryption/decryption process is shown in Figure 2.3.

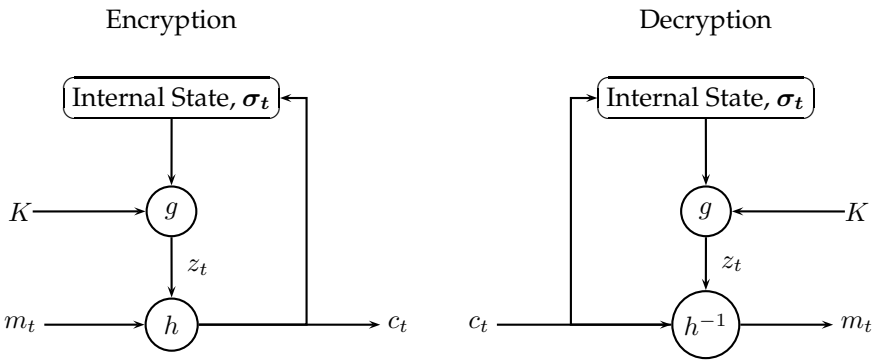


Figure 2.3: General structure of a self-synchronizing stream cipher.

The keystream symbol produced depends on the previously produced n ciphertext symbols. If symbols are inserted or deleted, a self-synchronizing stream cipher will resynchronize after n symbols. Hence, self-synchronizing stream ciphers solve the previously discussed synchronization problems of synchronous stream ciphers. However, there are few proposals of self-synchronizing stream ciphers, and the security has not yet been thoroughly studied. Two examples of self-synchronizing stream ciphers submitted to the eSTREAM project are Mosquito [DK05] and SSS [RHPdV05], both of which have been broken [JM06], [DLP05]. However, Mosquito has since been replaced by Moustique [DK07].

From here on, "stream cipher" will refer to additive synchronous stream ciphers.

2.2 The Building Blocks of Stream Ciphers

In this section we describe some commonly used building blocks for the construction of stream ciphers. We start with a description of a linear feedback shift register, which is a very popular building block in stream cipher primitives.

2.2.1 Linear Feedback Shift Registers

The *linear feedback shift register* (LFSR) is a common building block in stream ciphers . An LFSR with a primitive feedback polynomial generates sequences with very long periods and good statistical properties. Binary LFSRs are also very suitable for implementations that are very efficient in hardware. An LFSR is constructed out of r memory cells and each cell can hold one symbol, let s denote the symbol stored in a memory cell. A symbol is an element from the field $\mathbb{F}_q$, where $q = p^k$ for some prime number p and some integer k. For efficient stream cipher constructions, the binary field or some extension of it is commonly used, i.e., $q = 2^k$ for some integer $k \geq 1$. The content of the LFSR at a specific time is called the *state* of the LFSR. Let the state of the LFSR at time t be denoted as $\boldsymbol{s_t} = (s_t, \ldots, s_{t+r-1}), \quad s_i \in \mathbb{F}_q$, where $\boldsymbol{s_0} = (s_0, \ldots, s_{r-1})$ is the *initial state* of the LFSR. Figure 2.4 gives a general overview of an LFSR, in which $f_i \in \mathbb{F}_q$, and $0 \leq i \leq r$ are constants. Let $\oplus$ denote bitwise XOR and assume that $f_r \neq 0$. The LFSR then produces

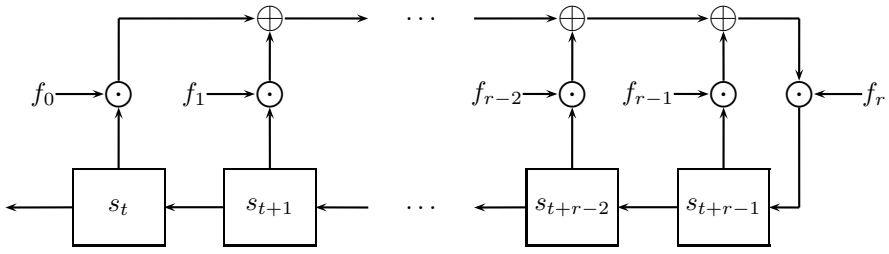


Figure 2.4: General form of an LFSR of length r.

a periodic sequence $s_0, s_1, \ldots$ where each symbol, $s_t, t \geq r$, can be described by a linear function of the previous r output symbols. The *linear recurrence relation* of an LFSR sequence is written as

$$s_{t+r} = f_r \bigoplus_{i=0}^{r-1} f_i s_{t+i}, \quad \forall t \geq r. \tag{2.7}$$

For LFSRs defined over the binary field, i.e., $f_i \in \mathbb{F}_2, \ 0 \leq i \leq r$ and $s_t \in \mathbb{F}_2, \ \forall t \geq 0$, the linear recurrence relation can be written as

$$\bigoplus_{i=0}^{r} f_i s_{t+i} = 0, \quad t \geq 0 \text{ with } f_r = 1, \tag{2.8}$$

where $\oplus$ is equivalent to the XOR operation. Additional to the linear recurrence relation, commonly the feedback of the LFSR is described as a polynomial. Two representations of this polynomial will be used in the thesis, namely:

- *Feedback polynomial.* Describes the feedback connection of the LFSR as a polynomial.

$$f(x) = f_r - f_{r-1}x - \ldots - f_1 x^{r-1} - f_0 x^r. \tag{2.9}$$

- *Characteristic polynomial.* The characteristic polynomial is the reciprocal of the feedback polynomial.

$$\tilde{f}(x) = -f_0 - f_1 x - \ldots - f_{r-1}x^{r-1} + f_r x^r. \tag{2.10}$$

From here on, we will assume that the feedback polynomial of the LFSR is *non-singular*, i.e., $f_0 \neq 0$. Then r is called the *degree* of the feedback polynomial. The number of nonzero taps in the feedback polynomial is called the *weight* of the polynomial and is denoted as w. An LFSR of length r has q^r distinct states. The all-zero state will always lead to a next state which is also the all-zero state, hence the longest achievable sequence before the symbols begin to repeat themselves is $q^r - 1$ symbols. The number of symbols produced before a sequence repeats itself is called the *period* of the sequence and is denoted as T. The period of an LFSR sequence depends on the properties of the feedback polynomial $f(x)$.

Definition 2.1: Let $f(x) \in \mathbb{F}_q[x]$ denote an *irreducible polynomial* of degree r_f, and let $p(x) \in \mathbb{F}_q[x]$ be another polynomial of degree r_p, then $p(x) \nmid f(x),\ \forall p(x) \in \mathbb{F}_q[x]\ s.t.\ 1 < r_p < r_f$.

The smallest T for which $f(x)$ divides $1 + x^T$, i.e., $f(x)|1 + x^T$, is called the *period* of the polynomial $f(x)$.

Definition 2.2: An *irreducible polynomial* $f(x) \in \mathbb{F}_q[x]$ of degree r, with a period of $q^r - 1$, is called a *primitive polynomial*.

For every non-zero initial state, an LFSR with a primitive feedback polynomial will visit every other non-zero state before it returns to the initial state. Such a sequence is called a *maximal-length shift register sequence*, or just an m-sequence.

Not only do m-sequences offer long and easily calculated periods, the sequences have many nice statistical properties, e.g., good run distribution properties and a good autocorrelation function.

run -length	0-runs	1-runs
1	2^{r-3}	2^{r-3}
2	2^{r-4}	2^{r-4}
$\vdots$	$\vdots$	$\vdots$
$r-2$	1	1
$r-1$	1	0
r	0	1

Table 2.1: The run distribution for any m-sequence of length $2^r - 1$.

- *Run-distribution properties.* A *run* of length l is a subsequence (of maximal length) of exactly l consecutive zeros or ones. For an m-sequence of length $2^r - 1$, the distribution of runs is tabulated in Table 2.1.
- *Autocorrelation function.* Let, $\boldsymbol{x} = (x_0, \ldots, x_{d-1})$, be a vector[1] of length d, where $x_i \in \{-1, +1\},\ 0 \leq i \leq d-1$. The autocorrelation function, denoted $C(\tau)$, is the correlation between $\boldsymbol{x}$ and the τ^{th} cyclic shift of $\boldsymbol{x}$, $(x_\tau, \ldots, x_{i+\tau \bmod d})$. We define $C(\tau)$ as

$$C(\tau) = \sum_{i=0}^{d-1} x_i x_{i+\tau \bmod d}. \tag{2.11}$$

  For a random sequence, we would like $C(\tau)$ to have very small values.

  For an m-sequence of length r, the autocorrelation function can be calculated as

$$c(\tau) = \begin{cases} -1, \text{ if } \tau \neq 0 \\ r, \text{ if } \tau = 0 \end{cases}. \tag{2.12}$$

For more on the properties of LFSRs and m-sequences, see [McE87].

LFSRs are linear and can easily be reconstructed from the output symbols. Let us introduce the linear complexity of a sequence. Let $L(\boldsymbol{s})$ denote the *linear complexity* of a sequence $\boldsymbol{s} = (s_0, s_1, \ldots, s_{n-1}) \in \mathbb{F}_q^n$, it is defined as the length of the shortest LFSR that generates $\boldsymbol{s}$ as its first n symbols. The linear complexity of a sequence can be determined by the Berlekamp-Massey algorithm [Mas69]. The Berlekamp-Massey algorithm efficiently determines the feedback polynomial of the shortest LFSR that generates the sequence, $\boldsymbol{s}$, from $2L(\boldsymbol{s})$ symbols of $\boldsymbol{s}$. For an LFSR sequence, the linear complexity is at most the length of the register, i.e., r. If we observe more than $2r$ consecutive output symbols from an LFSR, we can recover the entire state and feedback polynomial of the LFSR. Hence, it is unwise to build a

[1] In this vector we use $+1, -1$ instead of $0, 1$, e.g., $(0, 0, 1) \rightarrow (+1, +1, -1)$.

stream cipher entirely based on an LFSR. Two classical ways to destroy the linearity of an LFSR are to use non-linear Boolean functions and to clock the LSFR irregularly.

2.2.2 Nonlinear Feedback Shift Registers

In recent years, several stream ciphers that use *nonlinear feedback shift registers* (NLFSRs) have been proposed. NLFSRs are similar to ordinary LFSRs in their construction. The difference is that a nonlinear next-state function is used instead of a linear next-state function. Figure 2.5 shows a general model of a NLFSR where f denotes the nonlinear next state function. To

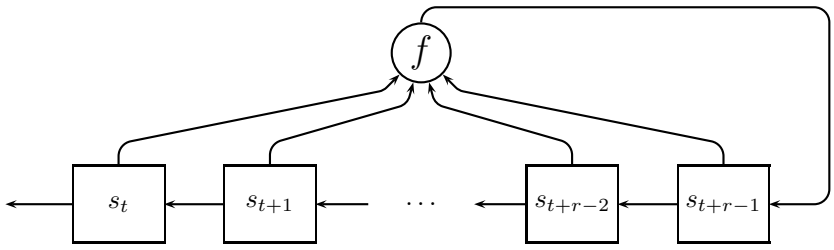


Figure 2.5: General form of a NLFSR of length r.

use NLFSRs is an old idea that has once again become popular in stream cipher constructions. However, even though much research has been done in the area, NLFSRs remains a difficult topic. Many of the attractive properties of LFSRs are lost when nonlinear next-state functions are used, e.g., knowledge of the period and the nice statistical properties of the output stream. However, if the state is large enough, one can argue that a NLFSR is unlikely to end up in a short period cycle. The advantage of NLFSRs over LFSRs is that there is no short linear recurrence relation that is satisfied with probability one, a property that is often exploited in attacks against LFSR based stream ciphers.

2.2.3 Boolean Functions

A Boolean function, denoted as h, is a mapping from a binary vector, $\boldsymbol{x} = (x_0, \ldots, x_{d-1})$ of length d where $x_i \in \mathbb{F}_2$, to a single bit, i.e., $h : \mathbb{F}_2^d \rightarrow \mathbb{F}_2$.

The set of all Boolean functions of d variables is denoted $\mathcal{B}_d$. In total there are 2^{2^d} Boolean functions of d variables.

There are many ways to represent a Boolean function, and in cryptology the *truth table* representation and the *algebraic normal form* (ANF) are the most common.

- *Truth table*. If the number of input variables of a Boolean function, h, is reasonably small, a truth table can be constructed. The truth table is a table in which all input vectors are listed together with the corresponding output value. If h is a function that takes d input variables, the number of input vectors are 2^d.

- *ANF*. For every Boolean function, h, of d variables there is a unique binary vector $\boldsymbol{a} = (a_0, \ldots, a_{2^d-1})$ such that

$$h(\boldsymbol{x}) = a_0 x_0 \oplus \ldots \oplus a_{d-1} x_{d-1} \oplus a_d x_0 x_1 \oplus \ldots \oplus a_{2^d-1} x_0 x_1 \ldots x_{d-1}. \quad (2.13)$$

  This representation is called the ANF of a Boolean function.

In cryptology, we are interested in various properties of Boolean functions. Let us first introduce the Walsh transform that will be of help in the analysis of the properties of Boolean functions.

Definition 2.3 (Walsh transform): The *Walsh transform* of a Boolean function, $h : \mathbb{F}_2^d \to \mathbb{F}_2$, is an integer valued function $\mathcal{W}_h : \mathbb{F}_2^d \to [-2^d, 2^d]$ defined as

$$\mathcal{W}_h(\boldsymbol{\omega}) = \sum_{\boldsymbol{x} \in \{0,1\}^n} (-1)^{h(\boldsymbol{x}) \oplus \boldsymbol{x} \cdot \boldsymbol{\omega}}, \quad (2.14)$$

where $\boldsymbol{x}, \boldsymbol{\omega} \in \mathbb{F}_2^d$ and $\boldsymbol{x} \cdot \boldsymbol{\omega} = x_0\omega_0 \oplus \ldots \oplus x_{d-1}\omega_{d-1}$.

The values of the coefficients $\mathcal{W}_h(\boldsymbol{\omega}),\ \boldsymbol{\omega} \in \mathbb{F}_2^d$ form the *Walsh-spectrum* of h.

We will also introduce the *Hamming weight* of a vector.

Definition 2.4 (Hamming weight): The number of ones of a vector is called the Hamming weight of the vector, let $w_H(\boldsymbol{\omega})$ denote the Hamming weight of vector $\boldsymbol{\omega}$.

The *Hamming distance* between two functions is defined as follows

Definition 2.5 (Hamming distance): Let $h_1(\boldsymbol{x}), h_2(\boldsymbol{x}) \in \mathcal{B}_d$ be two Boolean functions, the Hamming distance, $d_H(h_1, h_2)$, between $h_1(\boldsymbol{x})$ and $h_2(\boldsymbol{x})$ is

$$d_H(h_1, h_2) = |\{\boldsymbol{x} \in \mathbb{F}_2^n | h_1(\boldsymbol{x}) \neq h_2(\boldsymbol{x})\}|. \quad (2.15)$$

The first properties to be discussed are the balancedness and the algebraic degree of a Boolean function $h(\boldsymbol{x})$.

Definition 2.6 (Balancedness): A function is said to be *balanced* if $\Pr\big(h(\boldsymbol{x}) = 0\big) = \Pr\big(h(\boldsymbol{x}) = 1\big) = \frac{1}{2}$, when $\boldsymbol{x}$ is uniformly chosen from $\mathbb{F}_2^d$.

For a balanced function we have $\mathcal{W}_h(\mathbf{0}) = 0$, where $\mathbf{0} = (0, \ldots, 0)$.

Definition 2.7 (Algebraic degree): Let the *algebraic degree* of a Boolean function, h, be denoted $deg(h)$. The algebraic degree is the number of variables in the highest order term with a non-zero coefficient in the ANF form.

A function with an algebraic degree of at most one is called an *affine function*, the set of all possible affine functions of d input variables is denoted as $\mathcal{A}_d$. An affine function with the constant term equal to zero is called a *linear function*.

Definition 2.8 (Nonlinearity): Let the *nonlinearity* of a Boolean function, $h(\boldsymbol{x})$, be denoted $nl(h)$. The nonlinearity is the Hamming distance to the nearest affine function, i.e.,

$$nl(h) = \min_{g \in \mathcal{A}_d} d_H(h, g). \tag{2.16}$$

Nonlinearity is a very important property of Boolean functions used in stream ciphers. A common approach in cryptanalysis is to approximate the Boolean function with a simpler linear function. The nonlinearity of the function determines how good a linear approximation can be. It can be shown that the nonlinearity can be calculated via the Walsh transform as

$$nl(h) = 2^{d-1} - \frac{1}{2} \max_{\boldsymbol{\omega} \in \mathbb{F}_2^d} |\mathcal{W}_h(\boldsymbol{\omega})|. \tag{2.17}$$

Next, we introduce the correlation immunity of a Boolean function.

Definition 2.9 (Correlation immunity): Let $X_i,\ 0 \leq i \leq d-1$ be identically distributed random variables with $\Pr(X_i = 0) = \Pr(X_i = 1) = \frac{1}{2},\ 0 \leq i \leq d-1$. A Boolean function, $h(\boldsymbol{x})$, of d variables, is said to be k^{th} order *correlation immune* if $h(X_0, \ldots, X_{d-1})$ is statistically independent of any subset $X_{i_0}, \ldots, X_{i_{k-1}}$, of k or fewer random variables, where $0 \leq i_0 \leq \ldots \leq i_{k-1} \leq d-1$.

A Boolean function that is balanced and k^{th} order correlation immune is said to be *k-resilient*. Alternatively, the Walsh transform can be used to determine the correlation immunity of a Boolean function. A Boolean function is k^{th} order correlation immune if and only if its Walsh transform satisfies

$$\mathcal{W}_h(\boldsymbol{\omega}) = 0,\ \forall \boldsymbol{\omega} \in \mathbb{F}_2^d\ s.t.\ 1 \leq w_H(\boldsymbol{\omega}) \leq k. \tag{2.18}$$

Siegenthaler [Sie84] showed that there is a trade off between correlation immunity, k, and the algebraic degree, $deg(h)$, of a Boolean function, h, of d variables, namely

$$k + deg(h) \leq d. \tag{2.19}$$

EXAMPLE *2.1:* Let us study the described properties for the Boolean function

$$h(x_0, x_1, x_2) = x_0 \oplus x_0x_1 \oplus x_1x_2,$$

which is written in ANF form. The corresponding truth table is shown in Table 2.2. The Walsh transform is calculated in the same table. From the

x_0	x_1	x_2	$h(x_0, x_1, x_2)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

ω_0	ω_1	ω_2	$\mathcal{W}_h(\omega_0, \omega_1, \omega_2)$
0	0	0	0
0	0	1	4
0	1	0	0
0	1	1	-4
1	0	0	4
1	0	1	0
1	1	0	4
1	1	1	0

Table 2.2: The left table shows the truth table for the Boolean function $h(x_0, x_1, x_2) = x_0 \oplus x_0x_1 \oplus x_1x_2$. The right table shows the Walsh spectrum for $h(.)$.

table we can tell that the function is balanced, the nonlinearity is $2^{3-1} - \frac{1}{2}\max_{\boldsymbol{\omega}\in\mathbb{F}_2^3} |\mathcal{W}_h(\boldsymbol{\omega})| = 2$. We also see that the function is not first order correlation immune as $\mathcal{W}_h(0, 0, 1) = \mathcal{W}_h(1, 0, 0) \neq 0$.

2.2.4 S-boxes

A *substitution box* (S-box) is a mapping $h : \mathbb{F}_2^d \rightarrow \mathbb{F}_2^m$. It is basically a representation of m Boolean functions that take the same d input variables and produce m output symbols. Let $\boldsymbol{x} = (x_0, \ldots, x_{d-1})$ be a vector of the input variables to the S-box, and let $\boldsymbol{y} = (y_0, \ldots, y_{m-1})$ be the output vector. An S-box can be described by the following system of equations

$$\begin{aligned} y_1 &= h_0(x_0, \ldots, x_{d-1}), \\ &\vdots \\ y_{m-1} &= h_{m-1}(x_0, \ldots, x_{d-1}). \end{aligned} \tag{2.20}$$

The properties of Boolean functions that are of interest to cryptographic S-box applications are the same as the ones discussed in Section 2.2.3. The dif-

ference is that the properties are extended into the vector domain. Properties of S-boxes are taken as the minimum value over all linear combinations of the output functions $h_0, \ldots, h_{m-1}$. For example, the algebraic degree of an S-box is calculated as

$$\min_{\boldsymbol{\omega} \in \mathbb{F}_2^{m*}} deg\big(\sum_{i=0}^{m-1} \omega_i h_i\big), \tag{2.21}$$

where $\mathbb{F}_2^{m*}$ means all nonzero vectors of length m.

S-boxes are very popular in both block cipher and stream ciphers designs. An S-box with moderate values of m and n can be implemented as a table and is well suited for fast software implementations. See [Pas03] for a more thorough treatment of S-box properties.

2.3 Classical Stream Ciphers

In this section we present some general design principles and three classical LFSR-based stream ciphers. Although these constructions are not used in practice, they are often used as a starting point for the construction of stream ciphers.

2.3.1 Nonlinear Combination Generator

The *nonlinear combination generator* is constructed from n LFSRs. The output of the LFSRs are combined via a Boolean function, h, called the *combining function*. See Figure 2.6 for a description of a nonlinear combination generator. To destroy the linearity of the LFSRs, the combining function needs to have a high nonlinearity, see Section 2.2.3. Siegenthaler [Sie84] exploited correlations between single LFSRs and the keystream. To prevent this kind of attack the combining function must also have high correlation immunity. See Section 3.3 for a more thorough description of Siegenthaler's attack.

To withstand the Berlekamp-Massey algorithm, the combining function should also have a high algebraic degree. Let $r_i,\ 1 \leq i \leq n$ be the degrees of LFSR$_i$, assume that all LFSRs in the combination generator have distinct degrees greater than two, i.e., $r_i \neq r_j,\ \forall i, j\ s.t.\ i \neq j$. Then the linear complexity of the produced keystream is $L(\boldsymbol{z}) = f(r_1, \ldots, r_n)$ evaluated over the integers [MvOV97].

Braeken and Lano [BL05] gives an overview of what properties the combination function must have to withstand e.g. distinguishing, algebraic and fast correlation attacks. They conclude that it is highly unlikely that it is possible to construct a practical secure combination generator of modest state size (say ≤ 256 bits).

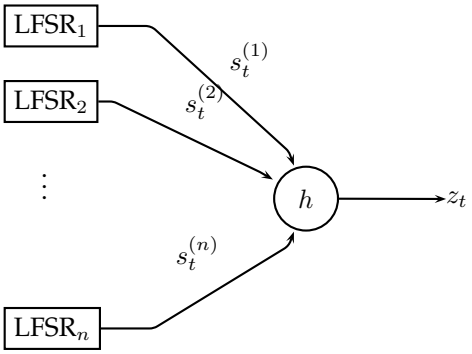


Figure 2.6: A combination generator of n LFSRs with combining function h.

2.3.2 Nonlinear Filter Generator

A *nonlinear filter generator* consists of two parts, one is linear, i.e., an LFSR, and one is a nonlinear function, h, as shown in Figure 2.7. As it is very easy to recreate the initial state of an LFSR from the output stream, we need to destroy the linearity in the keystream. This is the purpose of the nonlinear function also called the *filter function*. The filter function takes d symbols from fixed positions in the LFSR and produces one keystream symbol, z. Figure 2.7 describes the structure of a nonlinear filter generator, in which the filter function is denoted by h. Note that every filter generator can be

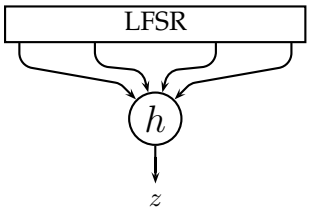


Figure 2.7: Filter Generator.

rewritten as an equivalent nonlinear combination generator that uses several shifted versions of the same LFSR. The filter function then takes one bit from each version as input, i.e., the combination generator would in this case consist of d LFSRs.

In cryptography, much work has been done on nonlinear functions, see

for example [Pas03, MvOV97] for more information. As for the combination generator, Braeken and Lano also discuss what properties a filter function needs to have in order to withstand common attacks [BL05]. They conclude that a secure filter generator of modest state size is a more realistic construction than a combination generator.

2.3.3 Clock-Controlled Generators

In nonlinear combining generators and nonlinear filter functions, Boolean functions are used to provide high nonlinearity. In a *clock-controlled generator* the main idea is to introduce nonlinearity by decimating the output sequence in some unpredictable way.

An example of a clock-controlled generator is the shrinking generator, [CKM93], in which one LFSR is used to decimate the output of another. The structure of the shrinking generator is given in Figure 2.8. In the figure,

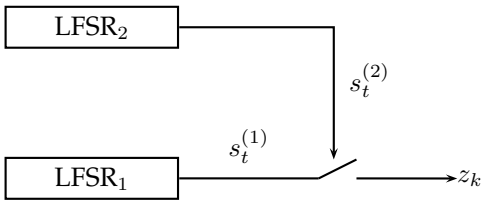


Figure 2.8: The shrinking generator.

the output from LFSR$_2$ is denoted as $s_t^{(2)}$ and is used to decimate the output from LFSR$_1$, denoted $s_t^{(1)}$. If $s_t^{(2)} = 1$, then $z_k = s_t^{(1)}$ is taken as the next keystream symbol for some index k, if $s_t^{(2)} = 0$, then $s_t^{(1)}$ is discarded. See Figure 2.9 for a summary of the decimation process. The self-shrinking

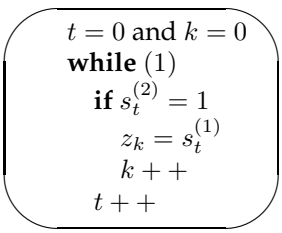


Figure 2.9: Decimation in the shrinking generator.

generator [MS94] uses only a single LFSR which decimates its own output. The stream cipher A5 used in GSM phones is another famous example of a clock-controlled generator.

2.4 Summary

In this chapter we give a general overview of stream ciphers. We defined synchronous and self-synchronizing stream ciphers, in particular we introduced additive stream ciphers, the class of ciphers studied in the present thesis. We have also discussed some common building blocks used in the construction of stream ciphers, such as LFSRs, NLFSRs, Boolean functions and S-boxes. Many cryptographically important properties of these building blocks have been discussed. There are many more possible building blocks. However, the building blocks presented here are the ones that are most relevant to the purposes of this thesis. We also give some examples of classical stream ciphers, i.e., the nonlinear combination generator, the nonlinear filter generator and clock-controlled generators. These designs are well known and well studied, and many modern stream ciphers are based on these designs.

3

Stream Cipher Cryptanalysis

For as long as people have tried to keep things secret, there have also been people on the other side, who try to find out the secrets. The ultimate goal of cryptanalysis is to recover either the secret key or the message. Although much research has been done in the area of stream cipher design, it is still not an easy task to design a secure stream cipher. Before a cryptographic primitive is released to wide deployment, it has to be extensively studied and analyzed. Theoretical attacks of high complexity are of great interest, as these attacks illuminate weaknesses in the design. These weaknesses might be explored further, which may lead to devastating attacks on the primitive.

In this chapter we first present three cryptanalytic techniques used in cryptanalysis of stream ciphers, and then present a number of possible attacks.

The first section presents the birthday problem, an algorithm for finding low weight multiples of LFSRs and theory for hypothesis testing. All of these are extensively used in stream cipher cryptanalysis. Section 3.2 describes the brute force attack, a general attack applicable to all ciphers. Section 3.3 explains the correlation attack as described by Siegenthaler, and also discuss improvements such as fast correlation attacks. The main topic of this thesis, distinguishing attacks, are introduced in Section 3.4. Section 3.5 describes time memory data trade-off attacks and Section 3.6 introduces resynchronization attacks. Algebraic attacks are shortly described in Section 3.7 and finally, Section 3.8 summarizes the chapter.

3.1 Tools for Cryptanalysis

Recall the attack scenarios and how we classified the success of an attack in Section 1.3. In attacks against additive synchronous stream ciphers we commonly assume that the plaintext is known, and hence also that the keystream is known. This is a reasonable assumption, as there are many situations in which the structure of messages makes parts of them easy to guess. An important known-plaintext attack was used by the British army to decipher encrypted German messages during World War II. At exactly the same time every day, the Germans sent military weather reports from U-Boats to shore stations. These reports always contained the word "Wetter" in the same position. With knowledge of the weather in the area from where the message was sent, even more of the plaintext could be guessed at. This known plaintext helped the British cryptanalysts recover the daily key, which in turn enabled them to recover other, more interesting messages.

In addition to knowledge of the plaintext, the adversary might in some situations acquire knowledge of the keystream from multiple known initialization vectors. Sometimes the assailant may even be able to interfere and choose initialization vectors himself and observe the resulting keystream. Such attacks are called resynchronization attacks and will be discussed in Section 3.6.

Let us begin by discussing three general problems that are often utilized in cryptanalysis: the birthday problem, how to find low weight multiples of LSFRs and how to test hypotheses. The solutions to these problems lie behind the design of many kinds of attack.

3.1.1 The Birthday Problems

The classical *birthday problem* proceeds from the fact that if there are more than 23 people in the same room, the probability that two people in that group have the same birthday is more than 50%. Intuitively, it seems impossible that the probability can be that high. Simple statitistical calculations, however, shows that it is true.

Assume we have N independent and identically distributed random variables, $X_0, \ldots, X_{N-1}$, each uniformly drawn from the integers $0, 1, \ldots, n-1$, i.e., from the alphabet $\mathcal{X} = \{0, 1, \ldots, n-1\}$. The probability that at least two of the random variables have the same value can be written as

$$1 - \prod_{i=0}^{N-1} (1 - \frac{i}{n}) \approx 1 - e^{-N^2/2n}. \tag{3.1}$$

According to this approximation, we see that we need $N \approx 1.2\sqrt{n}$ for a collision probability higher than 50%. The statistical property that is exemplified

by the birthday problem is utilized in many areas of cryptology, e.g., in the area of hash functions, the collision resistance is compared to the results of the birthday problem and in attacks against stream ciphers, the birthday problem is employed in time memory trade-off attacks. It is also used to find low weight multiples of LFSRs, a topic that will be further discussed in the next section.

There is another kind of birthday problem that often occurs in cryptography. Assume we have random variables, $X_0, X_1, \ldots$, uniformly drawn from the same alphabet as before, $\mathcal{X} = \{0, 1, \ldots, n-1\}$. Let us store k uniformly chosen values from $\mathcal{X}$ in a table, i.e., the table covers a fraction of $\frac{k}{n}$ of all values. Assume we have a realization, $x_0, \ldots, x_{N-1}$, where $x_i \in \mathcal{X},\ 0 \leq i \leq N-1$, of the sequence of random variables, $X_0, \ldots, X_{N-1}$. What is the length of this sequence, before we expect one of the variables in the sequence to take on a value stored in the table? The probability that at least one of these random variables is stored in the table can be calculated as

$$1 - \big(1 - \frac{k}{n}\big)^N. \tag{3.2}$$

Assume we store $k = \mathcal{O}(\sqrt{n})$ integers in the table, and observe a sequence of random variables of length $N = \mathcal{O}(\sqrt{n})$. It can be shown, see [MvOV97], that the probability in Equation (3.2) will be

$$\lim_{n\to\infty} 1 - \big(1 - \frac{k}{n}\big)^N \approx 1 - e^{-\frac{kN}{n}}. \tag{3.3}$$

We see that the probability of finding a collision is approximately the same as the probability derived in Equation (3.1).

3.1.2 How to find Low Weight Multiples of Binary LFSRs

Many attacks e.g. as fast correlation attacks and distinguishing attacks, require low weight recursions for LFSR sequences. As in Section 2.2.1, we denote the feedback polynomial of a binary linear feedback polynomial as $f(x) \in \mathbb{F}_2[x]$. The degree of the polynomial is denoted by r_f and the weight is denoted by w_f. We wish to find a multiple, denoted by $g(x)$, i.e., $g(x) = f(x) \cdot k(x)$ for some polynomial $k(x)$, where the weight of the multiple is w_g. In practice, we wish to find polynomials with weights as low as possible, e.g., $w_g \in \{3, 4, 5\}$.

Several methods to find multiples of low weight have been proposed and they optimize different aspects, e.g., finding multiples with the lowest possible degree or accepting a higher degree and reduce the complexity in order to find the multiple.

According to Golić [Gol96], if $f(x) \in \mathbb{F}_2[x]$ is a randomly chosen feedback polynomial, the critical degree when multiples, $g(x) \in \mathbb{F}_2[x]$, of weight

w_g start to appear is

$$r_g \approx (w_g - 1)!^{1/(w_g-1)} 2^{r_f/(w_g-1)}. \tag{3.4}$$

The expected number of multiples for a given degree r_g and weight w_g is estimated by

$$\frac{r_g^{w_g-1}}{(w_g-1)!2^{r_f}}. \tag{3.5}$$

The first method to find multiples of polynomials was described by Meier and Staffelbach in [MS89]. The algorithm is based on calculation of polynomial residues and on the birthday problem. Meier and Staffelbach's algorithm was used to find multiples of weight three or four.

Golić [Gol96] describes how to generalize the ideas in Meier and Staffelbach to find low weight multiples of arbitrary weights. The algorithm focuses on finding multiples with the lowest possible degree. The maximal weight is $w_g = 2k + 1$ for odd weighted multiples, and $w_g = 2k$ for even weighted multiples, for some positive integer k.

The first step in the algorithm is to compute the residues

$$x^i \mod f(x),\ 0 \leq i \leq r_g. \tag{3.6}$$

The next step is to compute and store the sum of all combinations of k residues from step one, i.e., compute

$$x^{i_1} + \ldots + x^{i_k} \mod f(x),\ 0 \leq i_1 \leq \ldots \leq i_k \leq r_g \tag{3.7}$$

for all $\binom{r_g}{k}$ possible combinations. The table now contains the following pairs

$$\big((i_1, \ldots, i_k),\ x^{i_1} + \ldots + x^{i_k} \mod f(x)\big), \tag{3.8}$$

where $0 \leq i_1 \leq \ldots \leq i_k \leq r_g$. The table is then sorted with respect to the residues, i.e., the second column. If we search for all residues that are equal, i.e., have same value in the second column, we will find all multiples of even weight $w_g = 2k$. That is, we search for all pairs such that

$$x^{i_1} + \ldots + x^{i_k} \equiv x^{j_1} + \ldots + x^{j_k} \pmod{f(x)}, \tag{3.9}$$

where $0 \leq i_1 \leq \ldots \leq i_k \leq r_g$, and $0 \leq j_1 \leq \ldots \leq j_k \leq r_g$ with $i_r \neq j_d, \forall r, d$.

Then

$$g(x) = x^{i_1} + \ldots + x^{i_k} + x^{j_1} + \ldots + x^{j_k} \tag{3.10}$$

is a multiple of $f(x)$. If we instead search for all residues in the table that sums to one, we will find all multiples of odd weight $w_g = 2k + 1$. Then

$$g(x) = 1 + x^{i_1} + \ldots + x^{i_k} + x^{j_1} + \ldots + x^{j_k} \tag{3.11}$$

1. Compute and store all residues $x^i \mod f(x),\ 1 \leq i \leq r_g$.
2. Compute all $\binom{r_g}{k}$ residues $x^{i_1} + \ldots + x^{i_k} \mod f(x)$, $0 \leq i_1 \leq \ldots \leq i_k \leq r_g$, and store the following pairs in a table $\big((i_1, \ldots, i_k),,\ x^{i_1} + \ldots + x^{i_k} \mod f(x)\big)$.
3. Sort the table from step (2) according to the residues (second column).
4. Find all zero and one matches in the table, i.e., find all residues such that $x^{i_1} + \ldots + x^{i_k} \equiv x^{j_1} + \ldots + x^{j_k} \pmod{f(x)}$ for multiples of weight $w_g = 2k$, and $1 + x^{i_1} + \ldots + x^{i_k} \equiv x^{j_1} + \ldots + x^{j_k} \pmod{f(x)}$ for multiples of weight $w_g = 2k + 1$.

Figure 3.1: Polynomial residue algorithm for finding multiples of polynomials.

is a multiple of $f(x)$. The algorithm is described in Figure 3.1. The complexity of this algorithm is approximately $O(S \log S)$ and it requires $O(S)$ memory, where

$$S = \begin{cases} \frac{(2k)!^{1/2}}{k!} 2^{r_f/2}, & \text{odd weights, } w = 2k+1 \\ \frac{(2k-1)!^{k/(2k-1)}}{k!} 2^{r_f k/(2k-1)} & \text{even weights, } w = 2k \end{cases}. \quad (3.12)$$

Wagner [Wag02] presented a generalization of the birthday problem, i.e., given k lists of n-bit values, find a way to choose one element from each list, so that these k values XOR to zero. This algorithm finds multiples of higher degree than the previously discussed algorithm but requires lower computational complexity and memory. However, the improvements are marginal for moderate values of the weight, w_g. As we in practice only will work only with multiples of weight $w_g \in \{3, 4, 5\}$, we will from now on use the algorithm described in Figure 3.1 to find low weight multiples of polynomials.

3.1.3 Hypothesis Testing

In a hypothesis test we have two hypotheses that we would like to test against each other, the null hypothesis, denoted H_0 and the alternative hypothesis, denoted H_1. Let us first introduce some notations. Let X be a random variable over a finite alphabet, $\mathcal{X}$, from some distribution, and let

$x \in \mathcal{X}$ be a realization of X. The *distribution function*, $\mathcal{P}_X$, for X is defined as

$$\mathcal{P}_X(x) = \Pr(X \leq x), \quad -\infty \leq x \leq \infty. \tag{3.13}$$

The *probability mass function*, $\mathcal{D}_X$, for a discrete random variable, X, is defined as

$$\mathcal{D}_X(x) = \Pr(X = x), \quad x \in \mathcal{X}. \tag{3.14}$$

In this thesis we mainly use discrete random variables, however on rare occasions we will also use the *probability density function* (PDF), denoted by f_X, for a continuous random variable, X. PDF is a function, which can be integrated to obtain the probability that the random variable takes a value in a given interval, hence

$$\mathcal{P}_X(x) = \int_{-\infty}^{x} f_X(x), \quad \infty \leq x \leq \infty. \tag{3.15}$$

When talking about distributions for a random variable X, we will loosely use $\mathcal{D}_X$ for notational purposes. Let $x_0, \ldots, x_{N-1}$ be a realization of the sequence $X_0, \ldots, X_{N-1}$ of random variables. A *decision rule*, denoted by δ, is a rule that decides which of the two hypothesis is most likely to be correct,

$$\delta(x_0, \ldots, x_{N-1}) = \begin{cases} H_0, \text{ if } x_0, \ldots, x_{N-1} \in \mathcal{AR} \\ H_1, \text{ otherwise} \end{cases},$$

where the set $\mathcal{AR}$ denotes the *acceptance region*. The *rejection region* is denoted by $\mathcal{AR}^c$. With a decision rule, two types of errors, denoted type I and type II, are of interest, see Table 3.1. Type I errors occur when the null hypothesis is incorrectly rejected, and type II errors when the null hypothesis is retained even though it is wrong. Let α denote the probability of type I

		Decision	
		Accept H_0	Reject H_0
Truth	H_0	Correct Retention	Type I Error
	H_1	Type II Error	Correct Rejection

Table 3.1: Decision errors for hypothesis testing.

errors, α is often called the *significance level* of the hypothesis test. The opposite is usually called the *confidence level* of the test and is denoted $1 - \alpha$. Let $1 - \beta$ denote the probability of not making a type II error, $1 - \beta$ is called the *power level*. The overall error probability, denoted as P_e, is expressed as

$$P_e = \pi_1 \alpha + \pi_2 \beta, \tag{3.16}$$

where $\pi_1,\ \pi_2$ are the *a priori* probabilities of the two hypotheses. We would like to minimize both α and β, but this is often difficult. In general α is considered more important and is fixed to a small value while β is unknown.

We will now describe two different hypothesis tests. The first is the χ^2-test, which is used when only one distribution is known and we would like to decide whether our samples follow that distribution. The second test, called the Neyman-Pearson test, is used when two distributions are known and we want to decide from which distribution our samples are drawn.

3.1.3.1 The χ^2-test: The Case of One Known Distribution

In some situations the distribution of a sample sequence is unknown and we would like to find out if it follows some known distribution, usually the normal distribution or the uniform distribution. The χ^2 goodness-of-fit test is often used in such cases.

In a χ^2 goodness-of-fit test one tries to decide whether or not the observed relative frequency distribution follows a specified distribution, $\mathcal{D}_0$. Assume we have N *independently and identically distributed* (i.i.d.) random variables $X_i,\ 0 \leq i \leq N-1$ from the alphabet $\mathcal{X}$ with unknown distribution $\mathcal{D}_X$. Let $x_0, \ldots, x_{N-1}$ be a realization of $X_i,\ 0 \leq i \leq N-1$. Then our two hypotheses are:

- $H_0\ :\ x_0, \ldots, x_{N-1}$ are samples from $\mathcal{D}_0$, i.e., $\mathcal{D}_X = \mathcal{D}_0$.
- $H_1\ :\ x_0, \ldots, x_{N-1}$ are not samples from $\mathcal{D}_0$, i.e., $\mathcal{D}_X \neq \mathcal{D}_0$.

It is important to note that even if we do not reject the null hypothesis, it might still be false as the sample size may be insufficient to establish the falsity of the null hypothesis.

Let $O_X(x)$ denote the number of observations of a specific outcome $x \in \mathcal{X}$. $E(x)$ is the expected number of observations of x under the assumption that $\mathcal{D}_X = \mathcal{D}_0$. Then the chi-square statistic is defined as

$$\chi^2 = \sum_{x \in \mathcal{X}} \frac{\big(O_X(x) - E(x)\big)^2}{E(x)} \xrightarrow{d} \chi^2(\nu), \tag{3.17}$$

where $\xrightarrow{d}$ means convergence in distribution, and ν is called the degrees of freedom (i.e., number of independent pieces of information). The *density function* for the Chi-square distribution is defined as

$$f_{\chi^2}(x) = \frac{1}{\Gamma(\nu/2)2^{\nu/2}} e^{-x/2} x^{\nu/2-1}, \tag{3.18}$$

where the *gamma function* $\Gamma(x)$ is determined as

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt. \tag{3.19}$$

Let $\chi^2_\alpha(\nu)$ be the value such that $\mathcal{P}_{\chi^2}(\chi^2_\alpha(\nu)) = 1 - \alpha$. For a one-sided χ^2 goodness-of-fit test, the null hypothesis, H_0, is rejected in favor of H_1 if the test statistic χ^2 is greater than the $\chi^2_\alpha(\nu)$ value, for some significance level α with ν degrees of freedom. If χ^2 is smaller than the tabulated value we accept hypothesis H_0, and hence we say that the samples are from $\mathcal{D}_0$. Hence, the decision rule is

$$\delta(\mathcal{D}_X) = \begin{cases} H_0, \text{ if } \chi^2 \leq \chi^2_\alpha(\nu) \\ H_1, \text{ otherwise} \end{cases}.$$

3.1.3.2 The Neyman-Pearson test: The Case of Two Known Distributions

In this section we assume that we know that a sequence of random variables $X_i \in \mathcal{X},\ 0 \leq i \leq N-1$ with distribution $\mathcal{D}_X$, either is from distribution $\mathcal{D}_0$ or is from distribution $\mathcal{D}_1$, where $\mathcal{D}_0$ and $\mathcal{D}_1$ are both known. Let $x_0, x_1, \ldots, x_{N-1}$ be an observed sequence from distribution $\mathcal{D}_X$. We would like to determine if this sequence is from $\mathcal{D}_0$ or $\mathcal{D}_1$. Our hypotheses are

- H_0 : $x_0, \ldots, x_{N-1}$ is a sequence from $\mathcal{D}_0$, i.e., $\mathcal{D}_X = \mathcal{D}_0$.
- H_1 : $x_0, \ldots, x_{N-1}$ is a sequence from $\mathcal{D}_1$, i.e., $\mathcal{D}_X = \mathcal{D}_1$.

To perform the actual hypothesis test we use the Neyman-Pearson lemma.

Lemma 3.1 (Neyman-Pearson lemma): Let $X_0, X_1, \ldots, X_{N-1}$ be drawn i.i.d. according to mass function $\mathcal{D}_X$. Consider the decision problem corresponding to the hypotheses $\mathcal{D}_X = \mathcal{D}_1$ vs. $\mathcal{D}_X = \mathcal{D}_0$. For $\lambda \geq 0$ define a region

$$\mathcal{AR}(\lambda) = \left\{ (x_0, x_1, \ldots, x_{N-1}) : \frac{\mathcal{D}_0(x_0, x_1, \ldots, x_{N-1})}{\mathcal{D}_1(x_0, x_1, \ldots, x_{N-1})} > \lambda \right\}. \tag{3.20}$$

Let $\alpha = \mathcal{D}_0(\mathcal{AR}^c(\lambda))$ and $\beta = \mathcal{D}_1(\mathcal{AR}(\lambda))$ be the error probabilities corresponding to the decision region, $\mathcal{AR}$, ($\mathcal{AR}$ denotes the acceptance region and $\mathcal{AR}^c$ the rejection region). Let $\mathcal{B}$ be any other decision region with associated error probabilities α^* and β^*. If $\alpha^* \leq \alpha$, then $\beta^* \geq \beta$.

The region $\mathcal{AR}(\lambda)$ minimizes α and β. In our case we set α and β to be equal and hence $\lambda = 1$. As all random variables $X_i,\ 0 \leq i \leq N-1$ are assumed to be independent we can rewrite the Neyman-Pearson lemma as a *log-likelihood ratio*, denoted by LLR,

$$LLR = \sum_{i=0}^{N-1} \left(\log_2 \frac{\mathcal{D}_0(x_i)}{\mathcal{D}_1(x_i)} \right). \tag{3.21}$$

The optimal decision function is then defined as

$$\delta(\mathcal{D}_X) = \begin{cases} H_0, \text{ if } LLR \geq 0 \\ H_1, \text{ if } LLR < 0 \end{cases}. \tag{3.22}$$

We also need to know how many keystream bits we need to observe in order to make a correct decision.

Sample Size Estimation

An important question in hypothesis tests is the sample size, i.e., how many samples are required for the hypothesis test to reliably distinguish between two distributions. The number of samples required depends greatly on how similar the distributions are. If two distributions, $\mathcal{D}_0$ and $\mathcal{D}_1$, are close to each other we can write them as

$$\begin{aligned} \mathcal{D}_0(x) &= p_x + \varepsilon_x, \\ \mathcal{D}_1(x) &= p_x, \end{aligned} \qquad \text{where } |\varepsilon_x| \ll p_x, \forall x \in \mathcal{X}. \tag{3.23}$$

We will now present two measures of the distance between two distributions, called the *statistical distance* and the *Squared Euclidean Imbalance*.

- *Statistical distance.* The statistical distance, denoted $|\mathcal{D}_0 - \mathcal{D}_1|$, between two known distributions, $\mathcal{D}_0$ and $\mathcal{D}_1$, over the finite alphabet $\mathcal{X}$, is defined as
$$|\mathcal{D}_0 - \mathcal{D}_1| = \frac{1}{2} \sum_{x \in \mathcal{X}} |\mathcal{D}_0(x) - \mathcal{D}_1(x)|, \tag{3.24}$$
where x is an element of $\mathcal{X}$. Assume that $x_0, x_1, \ldots, x_{N-1}$ is a sequence of samples from the distribution $\mathcal{D}_1$. If the statistical distance between distribution $\mathcal{D}_0$ and $\mathcal{D}_1$ is $|\mathcal{D}_0 - \mathcal{D}_1| = \varepsilon$, the number of samples, N, we need to observe to be able to distinguish between the two distributions is $N \approx 1/\varepsilon^2$, see [CHJ02] for a more details on the statistical distance.

- *Squared Euclidean Imbalance.* Let $\mu_i,\ i \in \{0,1\}$ denote the expected value, and $\sigma_i^2,\ i \in \{0,1\}$ the corresponding variance of the log-likelihood

ratio when the underlying distribution is $\mathcal{D}_i,\ i \in \{0,1\}$. Baignères, Junod and Vaudenay [BJV04], showed that

$$\mu_0 = -\mu_1 = \frac{1}{2}\sum_{x\in\mathcal{X}}\frac{\varepsilon_x^2}{p_x} \quad \text{and} \quad \sigma_0^2 = \sigma_1^2 = \sum_{x\in\mathcal{X}}\frac{\varepsilon_x^2}{p_x}\ . \tag{3.25}$$

Let us denote the distribution function for the normal distribution by $\Phi(x)$, where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{1}{2}u^2}du. \tag{3.26}$$

Using the central limit theorem Baignères *et al.* also showed that

$$\Pr\Big(\frac{LLR - N\mu_i}{\sigma_i\sqrt{N}} < t \mid \mathcal{D}_X = \mathcal{D}_i\Big) \overset{N\to\infty}{\longrightarrow} \Phi(t). \tag{3.27}$$

If we assume that the *a priori* probabilities in Equation (3.16) are equal, i.e., $\pi_0 = \pi_1 = 0.5$, then we obtain the following expression for the the total error probability

$$\begin{aligned} P_e &= \tfrac{1}{2}\big(\Pr(LLR < 0 \mid \mathcal{D}_X = \mathcal{D}_0) + \Pr(LLR \geq 0 \mid \mathcal{D}_X = \mathcal{D}_1)\big) = \\ &= \tfrac{1}{2}\big(\Pr(LLR < 0 \mid \mathcal{D}_X = \mathcal{D}_0) + 1 - \Pr(LLR < 0 \mid \mathcal{D}_X = \mathcal{D}_1)\big). \end{aligned} \tag{3.28}$$

If N is large, (3.27) gives us

$$\begin{aligned} \Pr(LLR < 0 | \mathcal{D}_X = \mathcal{D}_i) &= \Pr\big(\tfrac{LLR - N\mu_i}{\sigma_i\sqrt{N}} < -\tfrac{\sqrt{N}\mu_i}{\sigma_i} \mid \mathcal{D}_X = \mathcal{D}_i\big) = \\ &= \Phi(-\tfrac{\sqrt{N}\mu_i}{\sigma_i}). \end{aligned} \tag{3.29}$$

We know that $\Phi(-x) = 1 - \Phi(x)$ and finally we can derive an expression for the error probability,

$$\begin{aligned} P_e &= \tfrac{1}{2}\Big(\Phi\big(-\tfrac{\sqrt{N}\mu_0}{\sigma_0}\big) + 1 - \Phi\big(-\tfrac{\sqrt{N}\mu_1}{\sigma_1}\big)\Big) = \\ &= \frac{1}{2}\Big(\Phi\big(-\frac{1}{2}\sqrt{N\sum_{x\in\mathcal{X}}\frac{\varepsilon_x^2}{p_x}}\big) + 1 - \Phi\big(\frac{1}{2}\sqrt{N\sum_{x\in\mathcal{X}}\frac{\varepsilon_x^2}{p_x}}\big)\Big) = \\ &= \Phi\big(-\frac{1}{2}\sqrt{N\sum_{x\in\mathcal{X}}\frac{\varepsilon_x^2}{p_x}}\big). \end{aligned} \tag{3.30}$$

This expression can be used to estimate the number N of samples we need to achieve a certain error probability. We see that if we use

$$N = \frac{k}{\sum_{x \in \mathcal{X}} \frac{\varepsilon_x^2}{p_x}} \tag{3.31}$$

samples the error probability becomes $P_e = \Phi(-\sqrt{k}/2)$, for some constant k.

Definition 3.1 (Squared Euclidean Imbalance): Assume that one of the distributions is the uniform distribution, i.e., $\mathcal{D}_1(x) = \frac{1}{|\mathcal{X}|}$, $\forall x \in \mathcal{X}$. Let $\varepsilon_x = \mathcal{D}_0(x) - \mathcal{D}_1(x)$, where $x \in \mathcal{X}$. The Squared Euclidean Imbalance (SEI) is defined as

$$\Delta(\mathcal{D}_0) = |\mathcal{X}| \sum_{x \in \mathcal{X}} \varepsilon_x^2. \tag{3.32}$$

According to Equation (3.31) and Equation (3.30) we need approximately

$$N \approx \frac{k}{\Delta(\mathcal{D}_0)} \tag{3.33}$$

samples to reliably distinguish distribution $\mathcal{D}_0$ from the uniform distribution. Varying the value of the constant, k, we can achieve any error probability. In this thesis we will in general use $k \approx 11$ which results in an error probability of $P_e \approx 0.05$.

For a more thorough description of hypothesis testing, we refer to Cover and Thomas [CT91].

3.2 Brute Force Attacks

A brute force attack is a generic attack applicable to all symmetric ciphers. In this attack, keys are guessed and the output of the cipher is verified in some manner. Usually it is assumed that some amount of ciphertext and plaintext is known and can be compared to the output of the cipher for each guess. If the key used is of length $|K|$ bits, there are in total $2^{|K|}$ keys to try and on average the correct guess is expected after $2^{|K|-1}$ trials.

Brute force attacks are usually used as a threshold for other attacks. A cipher is said to be broken or insecure if an attack with lower computational complexity than the brute force attack is found. Lately, there has been some discussions on how to compare other kinds of attacks with brute force attacks, as the brute force attack is highly parallelizable with low memory

cost. Some suggest that instead of only comparing computational complexity, one should consider the computational complexity multiplied with the amount of memory required in the attack.

The key length of a cipher is usually chosen so as to make brute force attacks infeasible, stream ciphers often use key lengths of $128 - 256$ bits.

3.3 Correlation Attacks

The idea behind correlation attacks was first described by Siegenthaler in 1984 [Sie84]. He studied correlations between LFSRs and the keystream of nonlinear combination generators. Traditionally, correlation attacks are described for combination generators, see for example Handbook of Applied Cryptography [MvOV97].

As mentioned in Section 2.3.1, a nonlinear combination generator consists of n LFSRs, and we will let $r^{(j)}$ denote the length of LFSR$_j$, $1 \leq j \leq n$. The output symbol from LFSR$_j$ at time t is denoted as $s_t^{(j)}$. Assume there is a correlation between the output symbol of LFSR$_j$, $s_t^{(j)}$, and the keystream symbol z_t, i.e., $\Pr(s_t^{(j)} = z_t) \neq 0.5$. If we guess the initial state of LFSR$_j$ and count the number of coinciding bits in the keystream and the LFSR output, then the correlation can be found for the correct guess of the initial state. The principle of the attack is illustrated in Figure 3.2. It will take at most

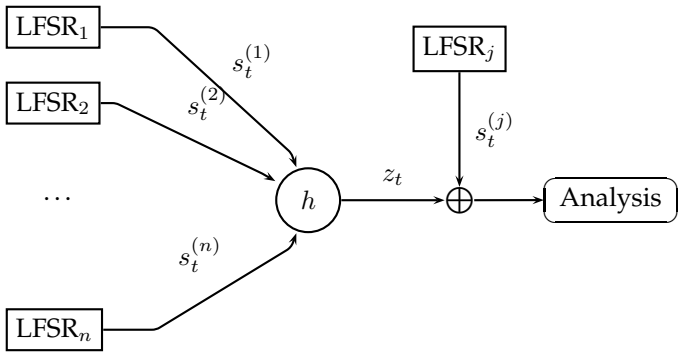


Figure 3.2: Principle of a correlation attack against a combination generator.

$2^{r^{(j)}} - 1$ trials to find the correct initial state for LFSR$_j$. When there are correlations between several of the LFSRs and the keystream, all the initial states of these LFSRs can be found independently of each other. When all of

the LFSRs are correlated with the keystream, the entire state of the generator can be determined if we make

$$\sum_{i=1}^{n}(2^{r^{(i)}}-1) \tag{3.34}$$

guesses.

To prevent the attack described by Siegenthaler, the nonlinear function, $h(.)$, must have a high correlation immunity. A k^{th} order correlation immune combiner function means that $k+1$ LFSRs have to be guessed simultaneously in order for an assailant to be able to observe a correlation with the output of the function.

Meier and Staffelbach [MS88] introduced a new way to model a stream cipher based on LFSRs. The output is modeled as an LFSR sequence that is passed through a binary symmetric channel (BSC), with crossover probability p, i.e., the channel introduces an error such that the bit changes value with probability p. The problem is then transformed into a decoding problem, i.e., we look at the noisy output from the BSC and try to decode the transmitted linear sequence. The correlation attack proposed by Meier and

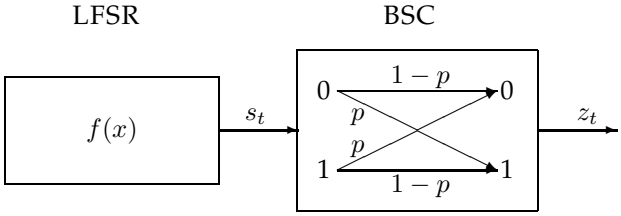


Figure 3.3: Model for a correlation attack.

Staffelbach, as well as many of the following attacks, rely on the existence of many low weight parity check equations for the LFSR sequence. Such low weight parity checks are equivalent to low weight multiples of the feedback polynomial. See Section 3.1.2 for a description of how to find such equations.

There are many improvements of the correlation attack that have been published, see [CT00, CS91, CJS00, JJ02, JJ00, JJ99].

3.4 Distinguishing Attacks

A *distinguishing attack* is an attack in which the adversary tries to determine whether a given sequence is produced by a known cipher or if it appears to

be a random sequence. In some cases a distinguishing attack can be used to create a key recovery attack.

A distinguisher can be seen as a black box that takes a sequence of symbols as input, and gives either *cipher* or *random* as output. A distinguishing attack is considered to be successful if the distinguisher gives a correct answer for most sequences. Figure 3.4 gives a general overview of a distinguisher.

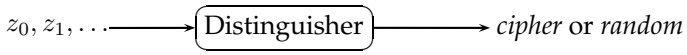


Figure 3.4: A distinguisher seen as a black box. The distinguisher takes keystream symbols $z_t,\ t = 0, 1, \ldots$ as input, and gives either *cipher* or *random* as output.

Distinguishing attacks are common and can be directed at both block ciphers and stream ciphers. Distinguishing attacks on block ciphers are used to detect statistical anomalies in the primitives, and these distinguishers are commonly used to analyze the resistance to linear and differential cryptanalysis, see e.g. Junod [Jun03]. Similarly, distinguishers for stream ciphers can also be seen as tools that detect anomalies in stream cipher primitives. A distinguishing attack of high complexity on a primitive might not pose a threat to the primitive *per se*. However, it does show that there is a weakness in the primitive. Such weaknesses may later be exploited in much more devastating attacks.

We will now give an example that highlights one situation in which distinguishing attacks are useful.

EXAMPLE *3.1:* Assume that the people in a small country are going to vote in a referendum. Alice is going to send her vote as a ciphertext, $\boldsymbol{c} = \boldsymbol{m} + \boldsymbol{z}$, where $\boldsymbol{m}$ is the vote and $\boldsymbol{z}$ is the keystream. Now assume that there are two possible ways to vote, either $\boldsymbol{m} = \boldsymbol{m}^{(1)}$ or $\boldsymbol{m} = \boldsymbol{m}^{(2)}$, and that both of them include some large amount of data (e.g. they are both pictures).

The adversary Eve, who eavesdrops on the channel, receives the ciphertext, $\boldsymbol{c}$. She guesses that Alice voted $\boldsymbol{m}^{(1)}$. Eve then adds $\boldsymbol{m}^{(1)}$ to the received ciphertext and depending on whether she made the right or wrong guess she obtains

$$\hat{\boldsymbol{z}} = \boldsymbol{c} \oplus \boldsymbol{m}^{(1)} = \begin{cases} \boldsymbol{z} & \text{correct guess} \\ \boldsymbol{m}^{(1)} \oplus \boldsymbol{m}^{(2)} \oplus \boldsymbol{z} & \text{wrong guess} \end{cases}. \tag{3.35}$$

Also assume that $\boldsymbol{m}^{(1)} + \boldsymbol{m}^{(2)}$ is random, an incorrect guess results in a random sequence. Assume that the cipher that produces $\boldsymbol{z}$ can be attacked

by a distinguishing attack. If Eve applies the distinguisher to the vector $\hat{z}$, she will be able to determine how Alice voted.

Linear statistical distinguishers on stream ciphers were introduced by Golić in 1994 [Gol94]. This work is based on an algorithm for finding non-balanced linear functions of the keystream, which is called the *linear sequential circuit approximation* (LSCA) and was first introduced in [Gol93]. Since then, many distinguishing attacks have been presented, see for example [EJ02, CHJ02].

In most scenarios we assume plaintext to be known and our problem is the following. Given the observed keystream sequence $\boldsymbol{z} = z_0, z_1 \ldots$ we want to distinguish the keystream from a truly random process.

Distinguishing attacks can be divided into two groups.

- *General distinguishing attacks*. Tests that do not consider the structure of the stream cipher at all are called general distinguishing attacks. The keystream generator is modeled as black box, and the statistical properties of the keystream are studied. Such attacks are useful for stream cipher designers who would like to evaluate different randomness properties of a primitive.

- *Cipher-specific distinguishing attacks*. Attacks that utilize the inner structure of a specific stream cipher are called cipher-specific distinguishing attacks. In this group, the distinguishing block in Figure 3.4 is divided into two parts, a processing part and a distinguishing part. In the processing part, keystream symbols are combined to create biased samples. These sample are then sent to the distinguisher, which makes a decision.

  In cipher specific distinguishers, one usually tries to use the inner structure of the cipher to find a relation that results in biased samples. A common approach to find such relations is to approximate all nonlinear parts of the cipher with linear relations. The approximations introduce noise such that the linear relation does not always hold. The aim is to find a relation that holds with as high probability as possible.

Let $\mathcal{D}_{\mathcal{C}}$ be the distribution of a sample sequence constructed from the cipher, and $\mathcal{D}_{\mathcal{U}}$ the uniform distribution. Let $X_0, \ldots, X_{N-1}$ be a sequence of random variables from distribution $\mathcal{D}_X$, and let $x_0, \ldots, x_{N-1}$ be a realization of it. We try to determine if the underlying distribution, $\mathcal{D}_X$, of the observed samples is more likely to be $\mathcal{D}_{\mathcal{C}}$ or $\mathcal{D}_{\mathcal{U}}$. In order to make that decision, distinguishing attacks almost always involve some sort of hypothesis testing, see Section 3.1.3.

A distinguisher is an implementation of a decision rule. Hence, our distinguisher performs a hypothesis test where our hypotheses are

- H_0 : $x_0, \ldots, x_{N-1}$ are samples from $\mathcal{D}_\mathcal{C}$,
- H_1 : $x_0, \ldots, x_{N-1}$ are samples from $\mathcal{D}_\mathcal{U}$.

The ability of a distinguisher to distinguish between two distributions is measured by the *advantage*. Let $\text{Adv}_{\mathcal{AR}}$ denote the advantage for an acceptance region, $\mathcal{AR}$, it is defined as

$$\text{Adv}_{\mathcal{AR}} = |\mathcal{D}_\mathcal{C}(\mathcal{AR}) - \mathcal{D}_\mathcal{U}(\mathcal{AR})|. \tag{3.36}$$

Assume that the *a priori* probabilities of $\mathcal{D}_\mathcal{C}$ and $\mathcal{D}_\mathcal{U}$ are equal. We see that $\mathcal{D}_\mathcal{C}(\mathcal{AR}) = 1-\alpha$ and $\mathcal{D}_\mathcal{U}(\mathcal{AR}) = \beta$, hence we can easily relate the advantage to the error probability of the hypothesis test as

$$\text{Adv}_{\mathcal{AR}} = 1 - (\alpha + \beta) = 1 - 2P_e, \tag{3.37}$$

assuming that $P_e < 1/2$.

We will now discuss two distinguishers in which varying amounts of knowledge about the cipher distribution is available to the assailant.

3.4.1 The Random and the Cipher Distribution are Known

In this section we assume that both the uniform distribution, $\mathcal{D}_\mathcal{U}$, and the distribution of the cipher, $\mathcal{D}_\mathcal{C}$, are known. In this case we use the log-likelihood test based on the optimal Neyman-Pearson test (this test was described in Section 3.1.3.2). Assume that the sequence of samples is $\boldsymbol{x} = x_0, x_1, \ldots, x_{N-1}$ where $x_i \in \mathcal{X},\ 0 \leq i \leq N-1$ and $|\mathcal{X}|$ denotes the cardinality of the sample domain. The log-likelihood ratio is calculated as

$$LLR = \sum_{i=0}^{N-1} \left(\log_2 \frac{\mathcal{D}_\mathcal{C}(x_i)}{1/|\mathcal{X}|} \right). \tag{3.38}$$

The decision rule based on the LLR is

$$\delta(\mathcal{D}_X) = \begin{cases} \textbf{cipher}, \text{ if } LLR \geq 0 \\ \textbf{random}, \text{ if } LLR < 0 \end{cases}. \tag{3.39}$$

3.4.2 Only The Random Distribution is Known

In some situations, the noise distribution, $\mathcal{D}_\mathcal{C}$, from the cipher is not known. Such situations appear in tests where the internal structure of the stream cipher is discarded, i.e., in general distinguishing attacks. In these tests one usually models the keystream generator as a black box, and studies the statistical behavior of the keystream. Examples of such tests are the Die Hard [Mar] test, the Crypt-X [aQUoTiA] test and the NIST test suite [NIS].

In general distinguishing attacks test we would usually use the χ^2 tests described in Section 3.1.3.1.

Assume that we observe the sequence $\boldsymbol{x} = x_0, x_1, \ldots, x_{N-1}$, where $x_i \in \mathcal{X},\ 0 \leq i \leq N-1$. We calculate the test statistic χ^2 according to Equation (3.17), and our decision rule is

$$\delta(\mathcal{D}_X) = \begin{cases} \textbf{cipher}, \text{ if } \chi^2 > \chi^2_\alpha(\nu) \\ \textbf{random}, \text{ if } \chi^2 \leq \chi^2_\alpha(\nu) \end{cases} . \tag{3.40}$$

3.5 Time Memory Data Trade-Off Attacks

In 1980, *time memory trade-off attacks* (TMTO) on block ciphers where introduced by Hellman [Hel80]. The idea is to divide the computational complexity of the attack into a *precomputation phase*, and an *online attack phase*. The precomputation is only performed once and the result is stored in memory. If we spend more time and memory on the precomputation phase, we can reduce the time spent in the online phase.

In the following, we describe *time memory data trade-off attacks* directed at stream ciphers. In this section we will use the following notations:

- N represents the size of the full search space to find a key or a state;
- P represents the computational complexity of the precomputation phase;
- M denotes the amount of required memory;
- T represents the computational complexity of the online attack phase;
- D represents the amount of available keystream.

The complexity of the attack is usually taken as $max(T, M)$, hence, for a successful attack we require $T,\ M < N$. Usually the precomputation time, P, is not taken into account, however, recently there have been some discussion whether attacks with $P > N$ are meaningful.

Babbage [Bab95] and Golić [Gol97] independently introduced trade-off attacks for stream ciphers. Let the search space, N, be the number of possible states and let the size of the state be denoted $|\boldsymbol{\sigma}|$, i.e., $N = 2^{|\boldsymbol{\sigma}|}$. Also, let f be a function, $f : \mathbb{F}_2^{\log N} \rightarrow \mathbb{F}_2^{\log N}$, which takes $\log N$ bits from the state and transforms it to a $\log N$ bit keystream, which is used as a prefix. In other words, run the keystream generator until $\log N$ keystream bits have been produced from the initial state $\boldsymbol{\sigma}$. Create the table

$$\begin{matrix} \big(\boldsymbol{\sigma_0}, f(\boldsymbol{\sigma_0})\big) \\ \vdots \\ \big(\boldsymbol{\sigma_{M-1}}, f(\boldsymbol{\sigma_{M-1}})\big) \end{matrix} , \tag{3.41}$$

from M randomly chosen initial states $\boldsymbol{\sigma_i}$, $0 \leq i \leq M-1$, in the stream cipher. The table is sorted, which can be done with computational complexity $\mathcal{O}(M \log M)$, according to the output keystream, i.e., according to $f(\boldsymbol{\sigma})$. The probability that a randomly chosen initial state will be stored in the table is M/N. Given a keystream sequence of $D = N/M + \log N - 1$ bits, we can extract N/M overlapping $\log N$-bit subsequences. With high probability one of these subsequences will be included in our stored table. If we find a collision in the table, the corresponding starting point is our internal state. This attack is an application of the second birthday problem discussed in Section 3.1.1.

Biryukov and Shamir [BS00] combined the ideas used by Babbage and Golić with the ideas introduced by Hellman. Let f be a function that maps an internal state on to a keystream word $f : \mathbb{F}_2^{\log N} \rightarrow \mathbb{F}_2^{\log N}$. Let $h_i : \mathbb{F}_2^{\log N} \rightarrow \mathbb{F}_2^{\log N}$ be a simple output modification function. Variants of the original f can then be used, i.e., $f_i(x) = h_i\big(f(x)\big)$. In the attack by Golić and Babbage only chains of length one were used. If we take the output from f to be the new state of the keystream generator, we can also produce longer chains for stream ciphers. The chains can be recreated from the starting state of the chain, hence we only need to store the start and the end of each chain. Let $\boldsymbol{\sigma_k^{(i)}}$ denote the k^{th} state in chain i, similarly let $\boldsymbol{z_k^{(i)}}$ denote the k^{th} output from f for chain i. Then, a keystream word at time t can be calculated as $\boldsymbol{z_t^{(i)}} = f(\boldsymbol{\sigma_t^{(i)}})$. The next state used in chain i can be calculated as $\boldsymbol{\sigma_{t+1}^{(i)}} = h_i\big(f(\boldsymbol{\sigma_t^{(i)}})\big)$. Figure 3.5 shows how to construct a matrix that covers $t \times L$ states.

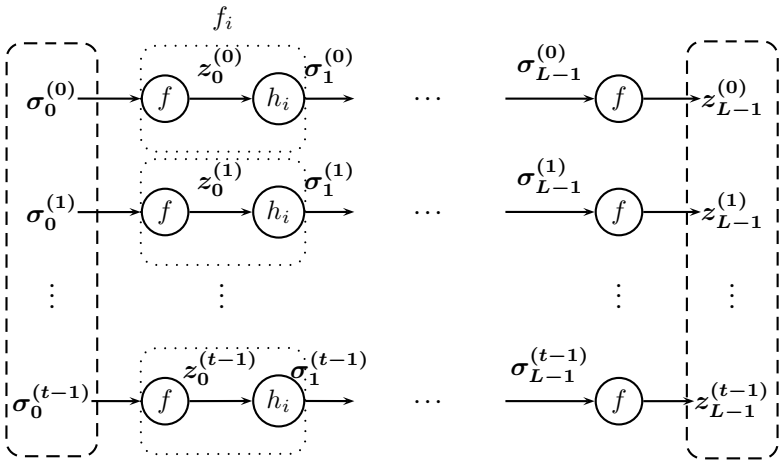


Figure 3.5: Hellman's matrix for a function $f_i(\boldsymbol{\sigma}) = h_i\big(f(\boldsymbol{\sigma})\big)$. The samples that are stored in the table are marked by dashed boxes.

Using birthday problem arguments, Hellman introduced the so called *matrix stopping rule*

$$t^2 L = N \tag{3.42}$$

as a guideline for how to choose t and L. Larger values of t and L lead to repeated entries in the matrix and hence to an inefficient coverage of points. Instead, we will use r different functions, h_i, and for each function we construct a matrix with a size that fulfills the matrix stopping rule. In Hellman's attack on block ciphers, the available keystream was restricted to one output word. When we attack stream ciphers we can use many keystream words, which yields a more powerful trade-off. Biryukov and Shamir showed that it is possible to attack stream ciphers with parameters that fulfill the relationships $P = N/D$ and $TM^2D^2 = N^2$.

To avoid the attacks described above, stream ciphers ought to have an internal state size of at least twice the key length.

3.6 Resynchronization Attacks

Most modern stream ciphers incorporate the usage of an initialization vector. The use of IVs gives an assailant even more freedom in attacks against a cipher. Not only can the assailant observe a sequence of keystream bits from a single keystream, he can observe keystream symbols from multiple keystreams, where distinct IVs and the same key are used. Figure 3.6 illustrates this scenario.

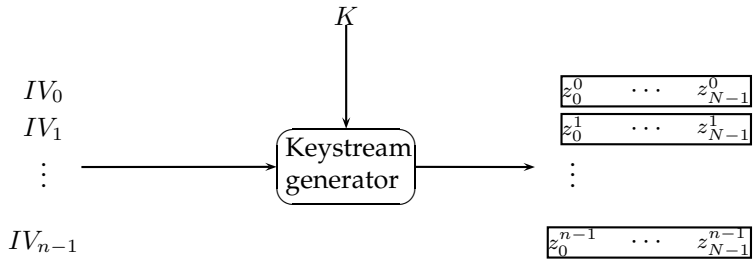


Figure 3.6: A keystream generator, which produces multiple keystreams from n different IVs. The same key is used for all keystreams.

As stated in Section 1.3.1, we distinguish between known IV attacks and chosen IV attacks. We will now discuss two common resynchronization scenarios.

In the first scenario a *fixed resync* is used. The message is divided into frames and each frame is encrypted with a unique IV. Usually, this IV is a frame counter which is just increased by one for every new frame. In this scenario an assailant can launch a known IV attack.

In the second resynchronization scenario, one of the parties of a transmission requests a resynchronization, called a *requested resync*. Under this scenario we assume that the IV may be chosen by one of the participants. This scenario enables a so called chosen IV attack, in which the assailant may also choose the IVs. This scenario might also enable a so called *repeated IV attack*, in which the same IV is used for encryption with several keys (it has been debated whether a repeated IV attack is relevant, as by definition an IV should only be used once).

The ultimate goal for a resynchronization attack is to recover either the secret key or the internal state. However, other attacks such as distinguishing attacks can also be considered as we will see in Chapter 9. Distinguishing attacks on multiple streams from different IVs can be useful to investigate the security of an initialization procedure. This will be further discussed in Section 10.

An example that demonstrates the importance of resynchronization attacks is the attack on the WEP protocol, e.g. [TWP07] based on the paper by Fluhrer, Mantin and Shamir [FMS01b].

3.7 Algebraic Attacks

Algebraic attacks have received much interest lately. These attacks try to reduce the key recovery problem to the problem of how to solve a large system of algebraic equations. Algebraic cryptanalysis goes back to Shannon who stated that to break a good cipher should require "as much work as solving a system of simultaneous equations in a large number of unknowns of a complex type" [Sha49].

In an algebraic attack, the first step is to try to describe the cipher with equations that only involves key bits, keystream bits and intermediate values.

The most famous algorithms can be categorized into two groups, namely *linearization techniques* and algorithms to compute *Gröbner bases*.

The basic *linearization attack* is very simple in its form: every nonlinear monomial is just replaced by a new linear variable and the resulting linear system of equations is solved with Gauss elimination. Usually this technique generates a very large number of unknowns. If a solution is to be found, the original system of equations has to be greatly over-defined. However, in attacks against stream ciphers, it is easy to extract many equations from a large amount of keystream. Every new keystream bit gives one

new equation. Hence, linearization techniques are well suited for attacks against stream ciphers.

Courtois and Meier showed [CM03] that algebraic attacks can be very efficient on some stream ciphers. They also showed that a Boolean function, $h(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{F}_2^d$, of high algebraic degree might not necessarily provide a good protection against algebraic attacks. If we multiply the Boolean function by an appropriately chosen function $k(\boldsymbol{x})$, we can reduce the degree of the function, i.e., $g(\boldsymbol{x}) = k(\boldsymbol{x})h(\boldsymbol{x})$ is of low degree. It has also been shown that the existence of such functions, $k(\boldsymbol{x})$, is equivalent to the existence of *annihilating functions*, i.e., functions, $k(\boldsymbol{x})$, such that $k(\boldsymbol{x})h(\boldsymbol{x}) = 0$ or $k(\boldsymbol{x})\big(1+h(\boldsymbol{x})\big) = 0$ [MPC04]. In the latter work a new property of a Boolean function, $h(.)$, called the *algebraic immunity*, denoted $AI(h)$, was also introduced. The algebraic immunity is the minimum degree of an annihilating function for the considered Boolean function.

3.8 Summary

In this chapter we have described the birthday attack, how to find low weight multiples of LFSRs and hypothesis testing, these are all very useful in the cryptanalysis of stream ciphers and will be frequently used in later chapters.

We discussed some common attacks on stream ciphers, both generic and cipher specific attacks. We described the generic brute force attack applicable to all ciphers. The correlation attack and its later improvements were briefly described, which was followed by an overview of distinguishing attacks. Time memory data trade-off attacks was described in the context of stream cipher cryptanalysis. We also introduced resynchronization attacks, in which multiple keystreams from different initialization vectors can be analyzed. Finally, we briefly mentioned algebraic attacks, which has been a hot topic in stream cipher cryptanalysis lately. The list is far from complete, but it covers the topics of most interest to us.

4

Correlation Attacks Using a New Class of Weak Feedback Polynomials

Due to the fast correlation attack introduced by Meier and Staffelbach [MS88] it is a well known fact that one avoids low weight feedback polynomials in the design of LFSR based stream ciphers.

In this chapter we will identify a new class of such weak feedback polynomials, namely, polynomials that can be written in the form

$$g(x) = g_0(x) + g_1(x)x^{r_1} + \ldots + g_{n-1}(x)x^{r_{n-1}},$$

where $g_0, g_1, \ldots, g_{n-1}$ are all polynomials of low degree. For such feedback polynomials, we identify an efficient correlation attack in the form of a distinguishing attack.

The results of the chapter are as follows. For the new class of such weak feedback polynomials, given above, we present an algorithm for launching an efficient fast correlation attack. The applicability of the algorithm can be divided in two groups.

Firstly, if the feedback polynomial is of the above form with a moderate number of g_i polynomials, the new algorithm will be much more powerful than applying any previously known (like Meier-Staffelbach) algorithm.

This could be interpreted as feedback polynomials of the above kind should be avoided when designing new LFSR based stream ciphers.

Secondly, for an arbitrary feedback polynomial, a standard approach in fast correlation attacks is to first search for low weight multiples of the feedback polynomial. Then, using the low weight multiples one applies the correlation attack. We can do the same and search for multiples of the feedback polynomial that have the above form. It turns out that this approach is in general less efficient than searching for low weight polynomials, but we can always find specific instances of feedback polynomials where we do get an improvement.

We model the keystream generator as a binary symmetric channel which was discussed briefly in Section 3.3. The BSC is once again illustrated in Figure 4.1. The target stream cipher uses two different components, one linear

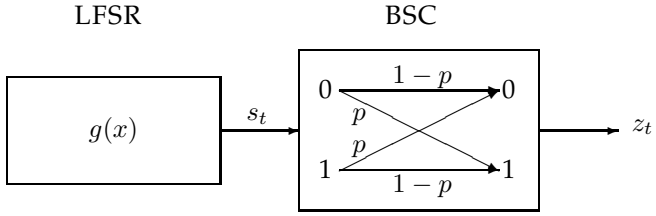


Figure 4.1: The model for a correlation attack.

and one nonlinear. The linear part is an LFSR and the nonlinear function can, through a linear approximation, be seen as a binary symmetric channel (BSC) with crossover probability p (correlation probability $1-p$) with $p \neq 0.5$.

The chapter is based on [EHJ04], and the outline is the following. We will describe how to perform a basic distinguisher using low weight recurrence relations in Section 4.1. We generalize the attack using vectors in Section 4.2, and in Section 4.3 we discuss how different parameters effect the attack. Methods of how to find multiples of the specified kind are described in Section 4.4, and the results of the attack are compared to the basic distinguisher in Section 4.5. Finally, the chapter is summarized in Section 4.6.

4.1 A Basic Distinguishing Attack

We will start by describing the basic binary distinguishing attack. Note that the theory in this section is well known and stems from the correlation attack by Meier and Staffelbach.

We will use the model in Figure 4.1 where the output sequence from the

LFSR is denoted $s_0, s_1 \ldots$. The observed keystream output can be written as a noisy version of the sequence from the LFSR,

$$z_t = s_t \oplus e_t, \tag{4.1}$$

where $e_t \in \mathbb{F}_2$, $t \geq 0$, are binary variables representing the noise introduced by the approximation. The noise is assumed to have a biased distribution

$$\Pr(e_t = 0) = 1 - p = \frac{1}{2} + \varepsilon, \tag{4.2}$$

where ε is usually rather small. Assume the characteristic polynomial of the LFSR can be written as

$$f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 + \ldots f_r x^r, \tag{4.3}$$

where $f_i \in \mathbb{F}_2,\ 0 \leq i \leq r$ and where $f_r = 1$. Recalling Section 2.2.1, we can write the linear recurrence relation as

$$\bigoplus_{j=0}^{r} f_j s_{t+j} = 0,\ t \geq 0, \tag{4.4}$$

where r is the length of the LFSR length and $f_j \in \mathbb{F}_2,\ 0 \leq j \leq r$, are known constants.

By adding the corresponding positions in the keystream $\boldsymbol{z}$ instead, we cancel out all the symbols from the LFSR. What remains is a sum of independent noise variables. Let us denote a sample x_t at time t, samples are calcualted as

$$x_t = \bigoplus_{j=0}^{r} f_j z_{t+j}. \tag{4.5}$$

Then by using the linear recurrence relation we can reduce the sum to

$$x_t = \bigoplus_{j=0}^{r} f_j s_{t+j} \oplus \bigoplus_{j=0}^{r} f_j e_{t+j} = \bigoplus_{j=0}^{r} f_j e_{t+j}. \tag{4.6}$$

As the distribution of e_t is nonuniform it is possible to distinguish the sample sequence x_t, $t = 1, 2, \ldots$, from a truly random sequence.

If we assume the binary case, the sum of the noise will have a bias which can be calculated as

$$\Pr\Big(\bigoplus_{j=0}^{r} f_j e_{t+j} = 0\Big) = \frac{1}{2} + 2^{w-1}\varepsilon^w, \tag{4.7}$$

where w is the weight of the feedback polynomial of the LFSR. This equation is very common in cryptology and is called the *piling-up lemma*, introduced by Matsui in [Mat94].

In this chapter we choose to use the Squared Euclidean Imbalance as a measure of the distance between two distributions, as described in Section 3.1.3.2. Let us consider the binary case above. The expression for the SEI becomes

$$\begin{aligned} \Delta(\mathcal{D}_0) &= |\mathcal{X}| \sum_{x \in \mathcal{X}} \varepsilon_x^2 \\ &= 2\big((2^{w-1}\varepsilon^w)^2 + (2^{w-1}\varepsilon^w)^2\big) = 2^{2w}\varepsilon^{2w}. \end{aligned} \tag{4.8}$$

Using Equation (3.33) we approximately need

$$N \approx \frac{11}{\Delta(\mathcal{D}_\mathcal{C})} = \frac{11}{2^{2w}\varepsilon^{2w}} \tag{4.9}$$

samples to distinguish the cipher distribution $\mathcal{D}_\mathcal{C}$ from the uniform distribution. The total error probability of the distinguisher is approximately $P_e = 0.05$, i.e., the distinguisher has the advantage

$$\mathrm{Adv}_{\mathcal{AR}} = |\mathcal{D}_\mathcal{C}(\mathcal{AR}) - \mathcal{D}_\mathcal{U}(\mathcal{AR})| = 1 - 2P_e \approx 0.9. \tag{4.10}$$

We see that the weight of the characteristic polynomial is directly connected to the success of the attack. In the rest of the section we will compare our results to this basic binary distinguisher.

4.2 A More General Distinguishing Attack Using Correlated Vectors

As seen in the previous section, low weight polynomials are easily attacked, and hence they are usually avoided in stream cipher constructions. When the weight of the LFSR grows, the required keystream length and the computational complexity grows exponentially. At some point we argue that the attack is no longer realistic, or it might require more than an exhaustive key search. However, as seen in Section 3.1.2 it is also possible to find low weight multiples of a polynomial. Hence it is also important that the degree of the original polynomial is sufficiently high. In this section we now describe a similar but more general approach that can be applied to another set of characteristic polynomials.

Consider a length r LFSR with characteristic polynomial as described in Equation (4.3). We try to find a multiple, $g(x)$, of the characteristic polynomial so that this polynomial can be written as

$$g(x) = f(x)k(x) = g_0(x) + x^r g_1(x), \tag{4.11}$$

where

$$\begin{aligned} g_0(x) &= g_{(0,0)} + g_{(0,1)}x + \ldots + g_{(0,k)}x^k \\ g_1(x) &= g_{(1,0)} + g_{(1,1)}x + \ldots + g_{(1,k)}x^k \end{aligned} \tag{4.12}$$

are polynomials of some small degree $\leq k$. It is possible that $f(x)$ is already on the form (4.11) and then $k(x) = 1$. In the remaining part of this section we assume that such a polynomial $g(x)$ of the above form is given. We will later also discuss how such polynomial multiples can be found.

Polynomials that can be written on the form (4.11) corresponds to shift registers for which the taps are concentrated to two regions far away from each other. The linear recurrence relation can then be written as the two sums

$$\bigoplus_{i=0}^{k} g_{(0,i)} s_{t+i} \oplus \bigoplus_{i=0}^{k} g_{(1,i)} s_{t+r+i} = 0, \tag{4.13}$$

where s_t is the t^{th} output bit from the LFSR. We now use 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 $e_t,\ t \geq 0$ are introduced by the approximation of the nonlinear part of the cipher. Furthermore, the biased noise has distribution $P(e_t = 0) = \frac{1}{2} + \varepsilon$, and the variables are identically and independently distributed.

Let us introduce the notation Q_t to be the sum

$$\begin{aligned} Q_t &= \bigoplus_{i=0}^{k} z_{t+i} g_{(0,i)} \oplus \bigoplus_{i=0}^{k} z_{t+r+i} g_{(1,i)} \\ &= \bigoplus_{i=0}^{k} e_{t+i} g_{(0,i)} \oplus \bigoplus_{i=0}^{k} e_{t+r+i} g_{(1,i)}. \end{aligned} \tag{4.14}$$

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

Introduce the vectorial noise vector E_t of length L as

$$E_t = (Q_{L \cdot t}, \ldots, Q_{L(t+1)-1}). \tag{4.15}$$

Rewriting the expression for Q_t as a vector multiplication, E_t can also be expressed as

$$E_t = (e_{L\cdot t}, \ldots, e_{L(t+1)+k-1}) \cdot G_0 + (e_{L\cdot t+r}, \ldots, e_{L(t+1)+r+k-1}) \cdot G_1, \quad (4.16)$$

where G_0 and G_1 are the $(L+k) \times L$ matrices

$$G_0 = \begin{pmatrix} g_{(0,0)} & & & \\ g_{(0,1)} & g_{(0,0)} & & \\ \vdots & \vdots & & g_{(0,0)} \\ g_{(0,k)} & g_{(0,k-1)} & \cdots & g_{(0,1)} \\ & g_{(0,k)} & & \vdots \\ & & & g_{(0,k)} \end{pmatrix} \text{ and } G_1 = \begin{pmatrix} g_{(1,0)} & & & \\ g_{(1,1)} & g_{(1,0)} & & \\ \vdots & \vdots & & g_{(1,0)} \\ g_{(1,k)} & g_{(1,k-1)} & \cdots & g_{(1,1)} \\ & g_{(1,k)} & & \vdots \\ & & & g_{(1,k)} \end{pmatrix}.$$

In the preprocessing stage of of the attack, we derive the distribution of the E_t noise vector given in (4.16). We denoted this distribution by $\mathcal{D}_\mathcal{C}$. This can be done easily for small polynomial degrees, as we know the distribution of the noise. Our aim is to distinguish this distribution from the truly random case, so by $\mathcal{D}_\mathcal{U}$ we denote the uniform distribution.

When performing the attack, we collect a sample sequence of length N',

$$Q_0, Q_1, Q_2, \ldots, Q_{N'-1}, \quad (4.17)$$

by $Q_t = \bigoplus_{i=0}^{k} z_{t+i} g_{(0,i)} \oplus \bigoplus_{i=0}^{k} z_{t+r+i} g_{(1,i)}$. This sample sequence is then transformed into a vectorial sample sequence $E_0, E_1, \ldots, E_{N-1}$ by $E_t = (Q_{L\cdot t}, \ldots, Q_{L(t+1)-1})$, $0 \le t \le N-1$ where $N' = L(N+1)-1$. The sample size N needed to distinguish $\mathcal{D}_\mathcal{C}$ and $\mathcal{D}_\mathcal{U}$, where $\mathcal{D}_\mathcal{U}(\boldsymbol{x}) = 1/2^L$, $\forall \boldsymbol{x} \in \mathbb{F}_2^L$, is determined using the Squared Euclidean Imbalance and Equation (3.33) as

$$N \approx \frac{11}{2^L \sum_{\boldsymbol{x} \in \mathbb{F}_2^L} \left(\mathcal{D}_\mathcal{C}(\boldsymbol{x}) - \frac{1}{2^L} \right)^2} \quad (4.18)$$

In the actual attack we will use the hypothesis test as described in Section 3.4.1, i.e., we will use the log-likelihood ratio in Equation (3.21). The proposed algorithm is summarized in Figure 4.2.

The performance of the algorithm depends on the polynomials, $g_0(x)$ and $g_1(x)$ that are used. Figure 4.2 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. $L = 1$ corresponds to the basic approach described in Section 4.1. We see that increasing the vector length will decrease the number of vectors needed. Note that $g_0(x)$ and $g_1(x)$ are

1. Find multiple $g(x)$ s.t. $g(x) = f(x)k(x) = g_0(x) + x^r g_1(x)$.
2. Calculate cipher distribution $\mathcal{D}_\mathcal{C}$.
3. **for** $t = 0 \dots N'$
   $$Q_t = \bigoplus_{i=0}^{k} z_{t+i} g_{(0,i)} \oplus \bigoplus_{i=0}^{k} z_{t+r+i} g_{(1,i)}$$
   end for.
4. **for** $t = 0 \dots N$
   $E_t = (Q_{L \cdot t}, \dots, Q_{L(t+1)-1})$
   end for.
5. Calculate $LLR = \sum_{t=0}^{N} \left(\log_2 \frac{\mathcal{D}_\mathcal{C}(E_t)}{1/2^L} \right)$.
6. **if** $(LLR \geq 0)$
   output **cipher**.
   else
   output **random**.

Figure 4.2: Summary of proposed algorithm.

just two examples of what the polynomials might look like, they do not represent a multiple of any specific primitive polynomial.

Finally, we note that it is possible to generalize the results to polynomials of arbitrarily many groups. Expression (4.11) for the polynomial then becomes

$$g(x) = k(x)f(x) = g_0(x) + x^{r_1} g_1(x) + \ldots + x^{r_{n-1}} g_{n-1}(x), \tag{4.19}$$

where $g_i(x)$ is a polynomial of some small degree $\leq k$, and $r_1 < r_2 < \ldots < r_{n-1}$. It is clear that when n grows it is easier to find multiples of the characteristic polynomial with the desired properties. But it is also clear that when n grows, the attack becomes weaker.

This distinguishing attack that we propose may be mounted on ciphers using a shift register where "good" multiples easily can be found. Ciphers using a polynomial where many taps are close together might be attacked directly without finding any multiple. This attack may be viewed as a new design criteria, *one should avoid LFSRs where multiples of the form (4.11) are easily found*.

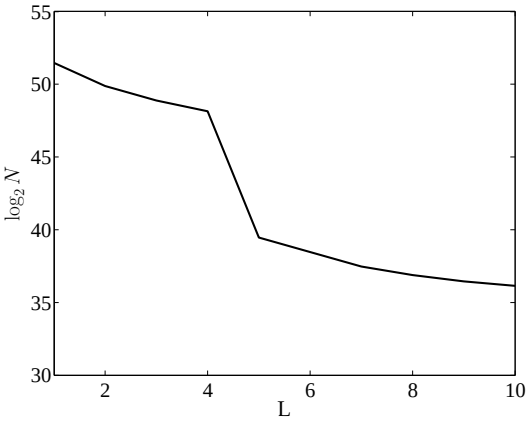


Figure 4.3: The number of vectors needed as a function of the vector length L. In this example $g_0(x) = 1 + x + x^5 + x^6$ and $g_1(x) = 1 + x + x^7 + x^8$.

4.3 The Parameters in the Attack

We have described the new attack in the previous section. We now discuss how various algorithmic parameters effect the results.

4.3.1 The Structure of $g_i(x)$

As the vectors E_t are correlated it is intuitive that the forms of $g_i(x)$ will effect the strength of the attack. Some combinations of polynomials will turn out to be much better than others.

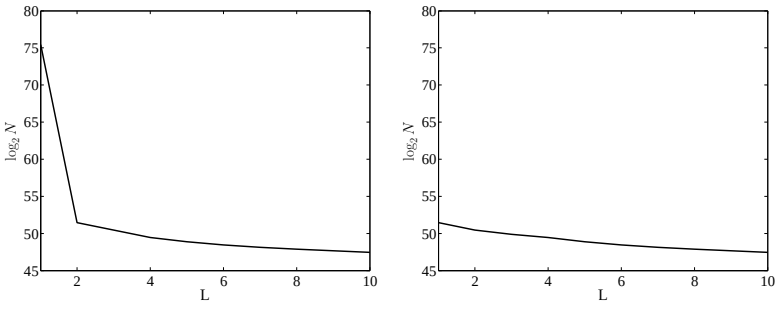
We have tested a number of different polynomials $g_i(x)$ in order to find a set of rules describing how the form of the polynomials effects the result. A definite rule to decide which polynomials are best suited for the attack is hard to find. Some basic properties that characterizes a good polynomial can be found. The parameters that determine how a polynomial will effect the distribution of the noise vectors are the following.

- *The weight of the polynomial.* A feedback polynomial of small weight leads to samples where few noise variables are involved and hence, a large bias. And a large weight means many noise variables and therefore a more uniform noise distribution.

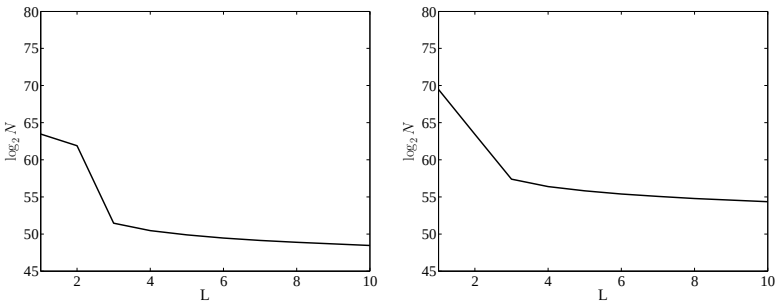
- *Arrangement of the terms in a polynomial.* This is the same as the arrangement of the taps in the LFSR. If there are many taps close together, the

corresponding noise variables will occur more frequently in the noise vector. This will significantly effect the distribution of the noise vectors.

As $r_i,\ 1 \leq i \leq n-1$ are typically large, all polynomials $g_i(x),\ 0 \leq i \leq n-1$ can be considered independent. Their properties will have the same influence on the total result. However, the polynomials are combined to form the distribution. This combination is just a variant of the piling-up lemma so it is obvious that the total distribution is more uniform than the distribution of the individual polynomials. Depending on the form of the individual polynomials, the resulting distribution becomes more or less uniform. Some combinations are "better" than others. Some examples of this can be found in Figure 4.4.



(a) $\boldsymbol{g_0} = (1111101)$ and $\boldsymbol{g_1} = (1110111)$ (b) $\boldsymbol{g_0} = (1001011)$ and $\boldsymbol{g_1} = (1100011)$



(c) $\boldsymbol{g_0} = (1011011)$ and $\boldsymbol{g_1} = (1100111)$ (d) $\boldsymbol{g_0} = (1111101)$ and $\boldsymbol{g_1} = (1011011)$

Figure 4.4: The sequence length needed as a function of the vector length L (logarithmic). In the figures the following vectors $\boldsymbol{g_i} = (g_{(i,0)}, g_{(i,1)}, \ldots, g_{(i,6)})$ corresponds to the polynomial $g_i(x) = g_{(i,0)} + g_{(i,1)}x + \ldots + g_{(i,6)}x^6$ for $i \in \{0, 1\}$.

4.3.2 The Vector Length

The length of the vectors, denoted L, impacts the effectiveness of the algorithm. The idea of using L larger than one is that we can get correlation between the vectors. Every time we increase L by 1 we will also increase the Squared Euclidean Imbalance. Recalling (3.33), we see that increasing the SEI means that we will decrease the number of vectors needed for our hypothesis test. The Squared Euclidean Imbalance is however not a linear function of L. Depending on the form of the polynomials the increase can be much higher for some L than for others. The computational complexity is also higher when L grows as each vector has 2^L different values. The downside is that the complexity of the calculations increase with increasing L. So there is trade off between the number of samples needed and the complexity of the calculations. The largest gain in Squared Euclidean Imbalance is usually achieved when going from $L = i$ to $L = i + 1$ for small i. This phenomenon can be seen in Figure 4.4.

4.3.3 The Number of Groups

As we will show in Section 4.4, it is much easier to find a multiple if we allow the multiple to have more groups than just two. The drawback is that the distribution will become more uniform if more groups are used. This is similar to the binary case in which more taps in the multiple will cause a less biased distribution. The difference is that there is no equivalence to Equation (4.7) for how much more uniform the distribution will be when having many groups, as this depends a lot on the polynomials used. However, if a multiple with many groups can be found where the degree of each group is quite small, using longer vectors still gives a significant improvement in the bias. This can be seen in Figure 4.5 where a polynomial with three groups is used.

4.4 Finding Multiples of the Characteristic Polynomial of a Desired Form

According to the piling-up lemma (4.7), the distribution becomes more uniform if the polynomial is of a high weight. Therefore, the first step in a correlation attack is to find a multiple that has a low weight. The multiple will produce the same sequence so the linear relation that describes the multiple will also satisfy the original LFSR sequence. There are easy and efficient ways of finding a multiple of a given weight. The number of bits needed to actually start the attack depends on the degree of the multiple, which in turn depends the weight of the same. As described in Section 3.1.2

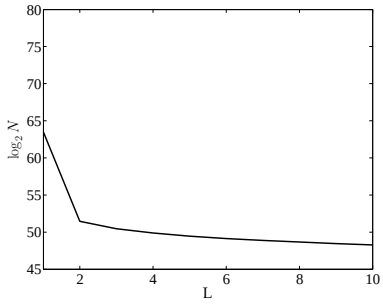


Figure 4.5: The sequence length needed as a function of the vector length L (logarithmic), three groups. In the figure three groups of polynomials are used, namely $\boldsymbol{g_0} = (1011)$, $\boldsymbol{g_1} = (1111)$ and $\boldsymbol{g_2} = (1110)$, using the notation introduced in Figure 4.4.

a multiple of weight w_g of a polynomial that has degree r_f has an expected degree of approximately

$$r_g \geq 2^{\frac{r_f}{w_g - 1}}. \tag{4.20}$$

In Table 4.1 the result of this equation is listed for an LFSR of length 100. We

r_g	w_g								
	2	3	4	5	6	7	8	9	10
$r_f = 100$	2^{100}	2^{50}	2^{33}	2^{25}	2^{20}	2^{17}	2^{14}	2^{123}	2^{11}

Table 4.1: The degree r_g of the multiple as a function of r_f and w_g.

have many times assumed that a multiple of the feedback polynomial has a certain form. We will now look at the problem of finding multiples of a such a form.

Say that we want to find a multiple of the form

$$g(x) = k(x)f(x) = g_0(x) + x^{r_g} g_1(x), \tag{4.21}$$

where $f(x)$ is the characteristic polynomial of the cipher. The degree of $f(x)$ is r_f. This can be found by polynomial division. Assume that we have a $g_1(x)$ of degree smaller than k. We then multiply this polynomial with x^i, $0 \leq i \leq r_g$. The result is divided by the original LFSR-polynomial. This gives us a quotient $q(x)$ and a remainder $g_0(x)$.

$$x^i g_1(x) = f(x) \cdot q(x) + g_0(x). \tag{4.22}$$

We have 2^k different $g_1(x)$-polynomials and i can be chosen in r_g different ways. The remainder $g_0(x)$ from the division is a polynomial with $0 \leq \deg(g_0(x)) < r_f$. If $g_0(x)$ has degree $\leq k$ we have found an acceptable $g_0(x)$. The probability of finding a polynomial of maximum degree k is

$$\Pr\Big(\deg\big(g_0(x)\big) \leq k\Big) = \frac{2^k}{2^{r_f}}. \tag{4.23}$$

We can estimate the number of polynomials $g_0(x)$ with degree smaller than k as

$$2^k \cdot r_g \cdot \Pr\Big(\deg\big(g_0(x)\big) \leq k\Big) = 2^k \cdot r_g \cdot \frac{2^k}{2^{r_f}}. \tag{4.24}$$

We would like this number to be greater or equal to one. Hence we need

$$r_g \geq 2^{r_f - 2k}. \tag{4.25}$$

Examining the result in (4.24) we see that for modest values of k the length of the multiple will become quite large. Therefore we extend our reasoning to the case with arbitrarily many groups, then we have a multiple of the form

$$g(x) = k(x)f(x) = g_0(x) + x^{r_1}g_1(x) + \ldots + x^{r_{n-1}}g_{n-1}(x). \tag{4.26}$$

If we use the same reasoning as above we receive a new expression

$$2^k \cdot r_1 \cdot 2^k \cdot r_2 \cdot \ldots \cdot 2^k \cdot r_{n-1} \cdot \frac{2^k}{2^{r_f}} \geq 1 \;\; \Rightarrow \;\; r_g \geq 2^{\frac{r_f - nk}{n-1}}, \tag{4.27}$$

where it is assumed that $r_1, r_2, \ldots, r_{t-1} \leq r_g$. This gives us an upper bound on all r_i.

In (4.27) we see that by using larger values of n, i.e., more groups, we can lower the length of the multiple. One has to bear in mind though that a larger n, as stated in Section 4.3.3, will usually effect the Squared Euclidean Imbalance in a negative way (from a cryptanalyst point of view). In Table 4.2 we list some values on r_g needed to find a multiple, for some values of k and n.

4.5 Comparing the Proposed Attack with a Basic Distinguishing Attack

Our algorithm can be used in two scenarios. Firstly, if the characteristic polynomial is of the form $g(x) = g_0(x) + g_1(x)x^{r_1} + \ldots + g_{n-1}(x)x^{r_{n-1}}$. The basic distinguisher applied on LFSRs of this family without finding

r_g		n				
		2	3	4	5	6
	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}
k	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 4.2: The degree r_g of the multiple as a function of the number of groups n and the degree of each group k for $r_f = 100$.

any multiple first, is equivalent to applying our algorithm with $L = 1$. As our algorithm has the ability to have correlated vectors with noise variables $(L > 1)$, it will be a significant improvement over the basic algorithm. Using the basic algorithm without first finding a multiple is naive, but if the length r_f of the LFSR is large, the degree of the low weight multiple will also be large, see (4.20). So if $f(x)$ is of high degree then our algorithm can be more effective.

Our algorithm can also be applied to arbitrary characteristic polynomials. The first step is then to find a multiple of the polynomial that is of the form $g(x) = g_0(x) + g_1(x)x^{r_1} + \ldots + g_{n-1}(x)x^{r_{n-1}}$. We then apply our algorithm. By comparing the two equations (4.20) and (4.27) 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, it is still not enough to compensate for the larger amount of noise variables. In this case the proposed attack will give improvements only for certain specific instances of characteristic polynomials, e.g., those having a surprisingly weak multiple of the form $g(x) = g_0(x) + x^{r_g}g_1(x)$ but no low weight multiples where the weight is surprisingly low.

4.6 Summary

Through a new correlation attack, we have identified a new class of weak feedback polynomials, namely, polynomials of the form $f(x) = g_0(x) +$

$g_1(x)x^{r_1} + \ldots + g_{n-1}(x)x^{r_{n-1}}$, where $g_0, g_1, \ldots, g_{n-1}$ 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, as the theoretical performance is easily calculated and we can compare with the basic attack based on low weight polynomials.

5

A New Simple Technique to Attack Filter Generators and Related Ciphers

In this chapter we consider binary stream ciphers where the output from an LFSR is filtered by a nonlinear function as described in Section 2.3.2. The structure of a nonlinear filter generator is pictured in Figure 5.1. Let

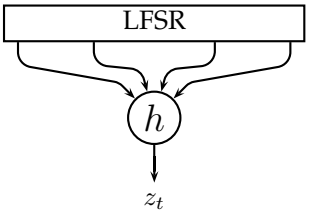


Figure 5.1: Filter Generator.

$s_t, s_{t+1}, \ldots$ denote the output sequence from a length r LFSR with feedback polynomial $f(x)$. Let h denote the nonlinear filter function. At each time t this function takes d input values from the LFSR and produces one output bit z_t. The variables used as inputs to h at time t are the entries in the vector

$\boldsymbol{s_t} = (s_{t+t_1}, s_{t+t_2}, \ldots, s_{t+t_d})$, $\boldsymbol{s_t} \in \mathbb{F}_2^d$. We thus have $z_t = h(\boldsymbol{s_t})$. This is modeled in Figure 5.1.

In this chapter we present a very simple distinguishing attack that can be applied on stream ciphers using a filter generator or a similar structure as a part of the cipher.

Leveiller *et al.* [LZGB03] proposed methods involving iterative decoding and the use of vectors instead of the binary symmetric channel. We use a similar idea, but much simpler in its form and more powerful in its performance, to mount a distinguishing attack. In the basic algorithm we first find a low weight multiple of the LFSR. We then consider the entries of the parity check equation as a vector. Such vectors, regarded as random variables, are non-uniformly distributed due to the parity check, and this is the key observation that we use to perform a distinguishing attack. This allows us to detect statistical deviations in the output sequence, creating the distinguishing attack. We can also present ideas on how to improve the performance by using slightly more complex algorithms.

In order to demonstrate the effectiveness of the proposed ideas, we apply them on a stream cipher called LILI-128. The attack is a *key recovery attack*. LILI-128 has one LFSR controlling the clock of another LFSR. Our approach is to guess the first 39 bits of the key, those bits that are used in the LFSR that controls the clocking. In doing so, the irregular clocking is removed and our analysis on filter generators can be applied to the rest of the cipher. If our guess is correct we will be able to detect some bias in the output sequence through the proposed distinguishing attacks. The complexity for one of the proposed attacks is roughly 2^{53} binary operations and it needs about 2^{47} keystream bits.

The chapter is based on [EJ04], and the outline is the following. We will describe the idea of the distinguisher in Section 5.1, and the attack is expanded in Section 5.2 to use more than one parity check equation. In Section 5.3 we describe a very simple state recovery attack where many weight three relations are used. To demonstrate the practicality of the proposed attack, we show how the stream cipher LILI-128 can be broken using this attack in Section 5.4. Finally, the chapter is summarized in Section 5.5.

5.1 Building a Distinguisher

A usual description of a stream cipher is to model it as a binary symmetric channel (BSC) (see Section 3.3), using linear approximations. But we proceed differently. Instead we write the terms in the weight w parity check equation as a length w vector. This way we use our knowledge of the nonlinear function better than in the BSC model.

If the feedback polynomial of the LFSR is of low weight from the beginning, we can apply our attack directly. Usually this is not the case, and our first step is then to try to find a low weight multiple of the feedback polynomial, see Section 3.1.2. The degree of the multiple gives a lower bound of the number of samples we need to observe, so in this attack we wish to minimize this degree. As the number of samples is of high concern to us we have chosen to work with the method described in Section 3.1.2. Continuing, we now assume that the LFSR sequence is described by a weight w recursion which can be written as

$$s_t \oplus s_{t+\tau_1} \oplus \ldots \oplus s_{t+\tau_{w-1}} = 0, \ \ \forall t \geq 0, \tag{5.1}$$

where $\tau_i, \ 1 \leq i \leq w-1$ denotes the tap positions of the LFSR relative to time t. We write the terms in this relation as a vector, and by noticing that $s_{t+\tau_{w-1}}$ is fully determined by the sum of the other components we get,

$$(s_t, s_{t+\tau_1}, \ldots, s_{t+\tau_{w-1}}) = (s_t, s_{t+\tau_1}, \ldots, \bigoplus_{i=0}^{w-2} s_{t+\tau_i}), \tag{5.2}$$

where $\tau_0 = 0$.

From the LFSR, d different positions are taken as input to the nonlinear function h. For each of these positions, where $t_1, t_2, \ldots, t_d$ denotes its position relative to time t, we can write a vector similar to (5.2). If we consider the following matrix,

$$A_t = \begin{pmatrix} s_{t+t_1} & s_{t+t_1+\tau_1} & \cdots & \bigoplus_{i=0}^{w-2} s_{t+t_1+\tau_i} \\ s_{t+t_2} & s_{t+t_2+\tau_1} & \cdots & \bigoplus_{i=0}^{w-2} s_{t+t_2+\tau_i} \\ \vdots & \vdots & & \vdots \\ s_{t+t_d} & s_{t+t_d+\tau_1} & \cdots & \bigoplus_{i=0}^{w-2} s_{t+t_d+\tau_i} \end{pmatrix}, \tag{5.3}$$

then by writing $\boldsymbol{s_{t+\tau_i}} = (s_{t+t_1+\tau_i}, s_{t+t_2+\tau_i}, \ldots, s_{t+t_d+\tau_i})^T$ we get

$$A_t = (\boldsymbol{s_t}, \boldsymbol{s_{t+\tau_1}}, \ldots, \bigoplus_{i=0}^{w-2} \boldsymbol{s_{t+\tau_i}}). \tag{5.4}$$

In the attack we will not have access to the LFSR output, instead we have access to the output bits from the nonlinear function h. The output values $(z_t, z_{t+\tau_1}, \ldots, z_{t+\tau_{w-1}})$, denoted as $\boldsymbol{z_t}$, can be described as

$$\boldsymbol{z_t} = (z_t, z_{t+\tau_1}, \ldots, z_{t+\tau_{w-1}}) = \big(h(\boldsymbol{s_t}), h(\boldsymbol{s_{t+\tau_1}}), \ldots, h(\bigoplus_{i=0}^{w-2} \boldsymbol{s_{t+\tau_i}})\big). \tag{5.5}$$

As we run through $s_t, s_{t+\tau_1}, \ldots, s_{t+\tau_{w-1}}$ in a nonuniform manner (not all values of $s_t, s_{t+\tau_1}, \ldots, s_{t+\tau_{w-1}}$ are possible), we will (in general) generate a nonuniform distribution of $(z_t, z_{t+\tau_1}, \ldots, z_{t+\tau_{w-1}})$. Leveiller [LZGB03] shows that the distribution of these vectors only depends on the parity. Hence, then it suffices to calculate the following probability

$$p = \Pr\big(h(s_t) + h(s_{t+\tau_1}) + \ldots + h(s_{t+\tau_{w-1}}) = 0 \mid \bigoplus_{i=0}^{w-1} s_{t+\tau_i} = 0\big). \quad (5.6)$$

Molland and Helleseth [MH04] showed that this probability can be calculated as

$$p = \frac{1}{2} + \frac{\sum_{\omega \in \mathbb{F}_2^d} \mathcal{W}_h(\omega)^w}{2^{wd+1}}, \quad (5.7)$$

where $\mathcal{W}_h(\omega)$ is the Walsh coefficient introduced in Section 2.2.3. However, as we will soon use vectors containing several recursions, we will consider the entire distribution of the vectors z. This distribution will be denoted $\mathcal{D}_{\mathcal{C}}(z),\ \forall z \in \mathbb{F}_2^w$. Let $\mathcal{S}$ be the set of vectors z of even Hamming weight (see Section 2.2.3), i.e., $S = \{z \in \mathbb{F}_2^w \mid w_H(z) = \text{even}\}$. We see that

$$p = \sum_{z \in \mathcal{S}} \mathcal{D}_{\mathcal{C}}(z). \quad (5.8)$$

If the number of output bits is large enough, we can perform a hypothesis test according to Section 3.1.3. In this hypothesis test we need the probability distribution $\mathcal{D}_{\mathcal{C}}$. This distribution can be calculated by running through all different values of $s_t, s_{t+\tau_1}, \ldots, s_{t+\tau_{w-1}}$. If d and w are large, the complexity for such a direct approach is too high. Then we can use slightly more advanced techniques based on building a trellis, that have much lower complexity.

When we have determined $\mathcal{D}_{\mathcal{C}}$, we can also estimate the sample size, N, needed to distinguish $\mathcal{D}_{\mathcal{C}}$ from the uniform distribution, $\mathcal{D}_{\mathcal{U}}$. The Squared Euclidean Imbalance and Equation (3.33), where $k = 11$ gives

$$N \approx \frac{11}{2^w \sum_{x \in \mathbb{F}_2^w} \left(\mathcal{D}_{\mathcal{C}}(x) - \frac{1}{2^w}\right)^2} \quad (5.9)$$

since $\mathcal{D}_{\mathcal{U}}(x) = 1/2^w,\ \forall x \in \mathbb{F}_2^w$.

The new basic distinguishing attack is summarized in Figure 5.2, where the final hypothesis test is performed using the log-likelihood ratio, see Equation (3.21). To really show the simplicity of the attack we will demonstrate with an example.

1. Find a weight w_g multiple $g(x)$ of $f(x)$.
2. Calculate the distribution $\mathcal{D}_\mathcal{C}$.
3. Calculate the length N we need to observe.
4. **for** $t = 0 \dots N$
   $\boldsymbol{z_t} = (z_t, z_{t+\tau_1}, \dots, z_{t+\tau_{w-1}})$
   end for
5. Calculate $LLR = \sum_{t=0}^{N} \log_2 \left(\frac{\mathcal{D}_\mathcal{C}(\boldsymbol{z_t})}{1/2^w} \right)$.
6. **if** $(LLR \geq 0)$
   output **cipher**.
   else
   output **random**.

Figure 5.2: Summary of the new basic distinguishing attack.

EXAMPLE *5.1:* We use an example from [LZGB03] in which we consider a weight three multiple, from which the output is filtered by an 8-input, 2-resilient plateaued function.

$$\begin{aligned} h(x_1, \dots, x_8) = \quad & x_1 + x_4 + x_5 + x_6 + x_7 + x_1(x_2 + x_7) + x_2x_6 + \\ & + x_3(x_6 + x_8) + x_1x_2(x_4 + x_6 + x_7 + x_8) + \\ & x_1x_3(x_2 + x_6) + x_1x_2x_3(x_4 + x_5 + x_8). \end{aligned} \tag{5.10}$$

For a parity of weight three and using the notation from Section 5.1 we can write the vectors as

$$(z_t, z_{t+\tau_1}, z_{t+\tau_2}) = \big(h(\boldsymbol{s_t}), h(\boldsymbol{s_{t+\tau_1}}), h(\boldsymbol{s_t} + \boldsymbol{s_{t+\tau_1}})\big). \tag{5.11}$$

If we try all possible inputs to this function and determine the distribution $\mathcal{D}_\mathcal{C}(z_t, z_{t+\tau_1}, z_{t+\tau_2})$ we get Table 5.1. We see that the probability only depends on the parity of these vectors. We can now calculate the Squared Euclidean Imbalance $\Delta(\mathcal{D}_\mathcal{C})$, defined in Section 3.1.3.2, as

$$\begin{aligned} \Delta(\mathcal{D}_\mathcal{C}) &= |\mathcal{X}| \sum_{x \in \mathcal{X}} \varepsilon_x^2 \\ &= 8\big(4 \cdot (\tfrac{8320-8192}{2^{16}})^2 + 4 \cdot (\tfrac{8064-8192}{2^{16}})^2\big) \\ &= 2^{-12}. \end{aligned} \tag{5.12}$$

$z_t, z_{t+\tau_1}, z_{t+\tau_2}$	$\mathcal{D}_{\mathcal{C}}(z_t, z_{t+\tau_1}, z_{t+\tau_2})$
000	$8320/2^{16}$
001	$8064/2^{16}$
010	$8064/2^{16}$
011	$8320/2^{16}$
100	$8064/2^{16}$
101	$8320/2^{16}$
110	$8320/2^{16}$
111	$8064/2^{16}$

Table 5.1: The probability distribution $\mathcal{D}_{\mathcal{C}}(z_t, z_{t+\tau_1}, z_{t+\tau_2})$.

Using Equation (3.33) we can calculate the number of keystream bits we need to observe in order to make a correct decision in the hypothesis test. The number of keystream bits we need is

$$N \approx \frac{11}{2^{-12}} \approx 45056. \tag{5.13}$$

This means that we need approximately 45056 bits to distinguish the cipher from a truly random source. Of course, we can use several weight three recursions (using squaring technique) and decrease the number of required bits. However, there are more powerful possibilities, as we will show in the next section.

5.2 Using More than one Parity Check Equation

The distinguishing attack described in the previous section is in a very simple form. We can improve the performance by using a slightly more advanced technique. If we can find more than one low weight parity check equation, we can use them simultaneously to improve performance. Assume that we have the two parity check equations,

$$\begin{aligned} s_t \oplus s_{t+\tau_1^{(1)}} \oplus \ldots \oplus s_{t+\tau_{w-1}^{(1)}} &= 0 \\ s_t \oplus s_{t+\tau_1^{(2)}} \oplus \ldots \oplus s_{t+\tau_{w-1}^{(2)}} &= 0 \end{aligned}, \tag{5.14}$$

where $\tau_j^{(i)}$ means tap position j in parity check equation i. Similarly as in Equation (5.4) we can use these parity equations in the vector

$$A_t = (\boldsymbol{s_t}, \boldsymbol{s_{t+\tau_1^{(1)}}}, \ldots, \bigoplus_{i=0}^{w-2} \boldsymbol{s_{t+\tau_i^{(1)}}}, \boldsymbol{s_{t+\tau_1^{(2)}}}, \ldots, \bigoplus_{i=0}^{w-2} \boldsymbol{s_{t+\tau_i^{(2)}}}). \tag{5.15}$$

We introduce in this case

$$\boldsymbol{z_t} = (z_t, z_{t+\tau_1^{(1)}}, \ldots, z_{t+\tau_{w-1}^{(1)}}, z_{t+\tau_1^{(2)}}, \ldots, z_{t+\tau_{w-1}^{(2)}}). \tag{5.16}$$

In this case two of the variables are totally determined by the other variables,

$$\boldsymbol{z_t} = \big(h(\boldsymbol{s_t}), \ldots, h(\bigoplus_{i=0}^{w-2} \boldsymbol{s_{t+\tau_i^{(1)}}}), h(\boldsymbol{s_{t+\tau_1^{(2)}}}), \ldots, h(\bigoplus_{i=0}^{w-2} \boldsymbol{s_{t+\tau_i^{(2)}}})\big). \tag{5.17}$$

In a similar manner we can use more than two parity checks in the vectors. Assume that we have P parity check equations. Then we have P positions in the vector that are fully determined by other positions. This means a more skew distribution of the output vector in (5.17). For the particular case of two parity check equations, the algorithm is described in Figure 5.3.

1. Find two weight w multiples of $g(x)$.
2. Calculate the distribution $\mathcal{D}_{\mathcal{C}}$.
3. Calculate the length, N, we need to observe.
4. **for** $t = 0 \ldots N$
   $\boldsymbol{z_t} = (z_t, z_{t+\tau_1^{(1)}}, \ldots, z_{t+\tau_{w-1}^{(2)}})$
   end for
5. Calculate $LLR = \sum_{t=0}^{N} \log_2 \left(\frac{\mathcal{D}_{\mathcal{C}}(\boldsymbol{z_t})}{1/2^w} \right)$.
6. **if** $(LLR \geq 0)$
   output **cipher**.
   else
   output **random**.

Figure 5.3: Summary of the new distinguishing attack using two parity check equations.

EXAMPLE *5.2:* In this section we consider the same example as in Section 5.1, but we use two recursions as described in Section 5.2. Assume the following two recurrence equations are available for the LFSR used

$$\begin{aligned} s_t \oplus s_{t+\tau_1^{(1)}} \oplus s_{t+\tau_2^{(1)}} &= 0 \\ s_t \oplus s_{t+\tau_1^{(2)}} \oplus s_{t+\tau_2^{(2)}} &= 0 \end{aligned} .$$

Hence we observe the keystream vectors

$$(z_t, z_{t+\tau_1^{(1)}}, \ldots, z_{t+\tau_2^{(2)}}) =$$
$$= \big(h(\boldsymbol{s_t}), h(\boldsymbol{s_{t+\tau_1^{(1)}}}), h(\boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_1^{(1)}}}), h(\boldsymbol{s_{t+\tau_1^{(2)}}}), h(\boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_1^{(2)}}})\big),$$

when $(\boldsymbol{s_t}, \boldsymbol{s_{t+\tau_1^{(1)}}}, \boldsymbol{s_{t+\tau_1^{(2)}}})$ runs through all values. Similarly as before we use the Squared Euclidean Imbalance in order to determine the number of vectors N we need to make a correct decision. The SEI is approximately $\Delta(\mathcal{D}_\mathcal{C}) \approx 2^{-10.42}$ and hence we need $N \approx 11/2^{-10.42} = 15019$ vectors to distinguish $\mathcal{D}_\mathcal{C}$ from $\mathcal{D}_\mathcal{U}$. We see that the result is a significant improvement. If we extend the reasoning and use three weight three recursions we get $\varepsilon = 2^{-9.35}$ and hence we need $N \approx 11/2^{-9.35} = 7172$ vectors. The gain of using three recursions instead of two is smaller.

5.3 The Weight Three Attack

If we can find many multiples of weight three of a feedback polynomial we can simplify the description of our attack. With a d-input nonlinear function we write one parity check as

$$\boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_1}} \oplus \boldsymbol{s_{t+\tau_2}} = \boldsymbol{0}. \tag{5.18}$$

If $\boldsymbol{s_t} = \boldsymbol{0}$ we notice that $\boldsymbol{s_{t+\tau_1}} = \boldsymbol{s_{t+\tau_2}}$. If this is the case, then obviously $z_{t+\tau_1} = z_{t+\tau_2}$ (we assume that $h(\boldsymbol{0}) = 0$). Now, assume that we have P weight three parity checks, where $P > d$, then

$$\begin{aligned} \boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_1^{(1)}}} \oplus \boldsymbol{s_{t+\tau_2^{(1)}}} &= \boldsymbol{0}, \\ \boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_1^{(2)}}} \oplus \boldsymbol{s_{t+\tau_2^{(2)}}} &= \boldsymbol{0}, \\ &\vdots \\ \boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_2^{(P)}}} \oplus \boldsymbol{s_{t+\tau_2^{(P)}}} &= \boldsymbol{0}. \end{aligned} \tag{5.19}$$

Again assuming $\boldsymbol{s_t} = \boldsymbol{0}$ we see that we must have

$$\begin{aligned} z_{t+\tau_1^{(1)}} &= z_{t+\tau_2^{(1)}}, \\ z_{t+\tau_1^{(2)}} &= z_{t+\tau_2^{(2)}}, \\ &\vdots \\ z_{t+\tau_1^{(P)}} &= z_{t+\tau_2^{(P)}}. \end{aligned} \tag{5.20}$$

So, as $\Pr(\boldsymbol{s_t} = \boldsymbol{0}) = 2^{-d}$ we will have

$$\Pr(z_{t+\tau_1^{(1)}} = z_{t+\tau_2^{(1)}}, \ \ldots, z_{t+\tau_1^{(P)}} = z_{t+\tau_2^{(P)}}) > 2^{-d}. \tag{5.21}$$

For a purely random sequence, however, this probability is 2^{-P}. It is important to note that when (5.21) holds for some t, it is very probable that $\boldsymbol{s_t} = \mathbf{0}$, i.e., *we have recovered a part of the state*. As all output bits from the length r LFSR can be written as a linear combination of the initial state, $s_{t+t_j} = \bigoplus_{i=0}^{r-1} a_i s_i,\ j = 1 \ldots d$ where $a_i \in \{0, 1\}$ are constants, we get d equations of the kind $s_{t+t_j} = \bigoplus_{i=0}^{r-1} a_i s_i = 0$ for each $\boldsymbol{s_t} = \mathbf{0}$. Finding another value of t for which (5.21) holds gives more expressions describing the state. As a full rank of the system of equations would only lead to the all zero solution, we need to guess at least one bit of the state. Then simple Gauss elimination can be applied to the system to deduce the other state bits. So we have described a *state recovery attack*. This attack has major consequences for any filter generator, as well as for nonlinear combining generators, and possibly also others. Basically, any filter generator of length r where the number of inputs d is smaller than $r/2$ can be broken very easily if we have access to a bit more than $2^{r/2}$ output symbols.

This leads to an attack as described in Figure 5.4.

1. Find P weight three multiples of $f(x)$.
2. Calculate the number of samples N we need to observe.
3. **for** $t = 0 \ldots N - 1$
   if $z_t = 0$ and
   $z_{t+\tau_1^{(1)}} = z_{t+\tau_2^{(1)}}$
   $z_{t+\tau_1^{(2)}} = z_{t+\tau_2^{(2)}}$
   $\vdots$
   $z_{t+\tau_1^{(P)}} = z_{t+\tau_2^{(P)}}$
   then assign $\boldsymbol{s_t} = \mathbf{0}$.
4. Guess at least one $\boldsymbol{s_t}$ and then recover $\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots$ by linear algebra.

Figure 5.4: Summary of the weight three state recovery attack.

5.4 A Key Recovery Attack on LILI-128

In 2000 a project called NESSIE was initialized. The aim of this project was to collect a strong portfolio of cryptographic primitives. After a open call for proposals the submissions were thoroughly evaluated. One proposed candidate in the stream cipher category was called LILI-128 [CDF+00]. The cipher is constructed of two filter generators, where the output from one filter generator is used to irregularly clock the other filter generator, see Figure

5.5. In the analysis of the cipher, the internal state of the LFSR of the filter generator producing the clock controll sequence, will be guessed. Knowing the clocking of the second filter generator methods describe previously in this section can be applied to the cipher. For the correct guess a bias in the output can be found, for all incorrect guesses the output will appear as random data.

5.4.1 Description of LILI-128

LILI-128 has the structure of a filter generator. The only difference is that LILI-128 use an irregular clocking. LILI-128 consists of a first LFSR, called LFSR$_c$, that via a nonlinear function clocks a second LFSR, called LFSR$_d$, irregularly. The structure can be viewed in Figure 5.5. LILI-128 use a key

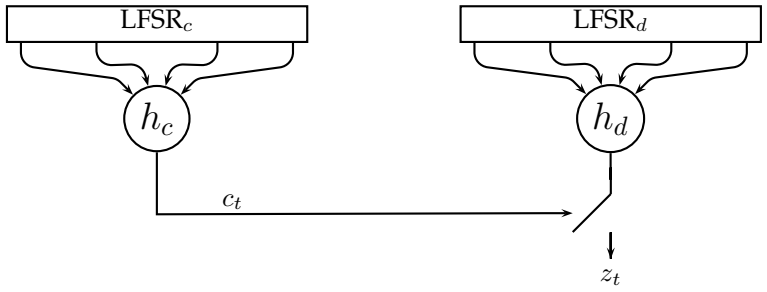


Figure 5.5: LILI-128.

length of 128 bits, the key is used directly to initialize the two binary LFSRs from left to right. As the feedback polynomial of the first shift register, LFSR$_c$ is a primitive polynomial of length 39, the leftmost 39 bits of the key is used to initialize LFSR$_c$. The remaining 89 bits are used in the same manner to initialize LFSR$_d$. The feedback polynomial for LFSR$_c$ is given by

$$f_c(x) = x^{39} + x^{35} + x^{33} + x^{31} + x^{17} + x^{15} + x^{14} + x^2 + 1 \, . \tag{5.22}$$

The Boolean function h_c takes two input bits from LFSR$_c$, namely the bit in stage 12 and the bit in stage 20 of the LFSR. The Boolean function h_c is chosen to be

$$h_c(x_{12}, x_{20}) = 2 \cdot x_{12} + x_{20} + 1 \, . \tag{5.23}$$

The output of this function is used to clock LFSR$_d$ irregularly. The reason for using irregular clocking [CDF+00], was that regularly clocked LFSRs are vulnerable to correlation and fast correlation attacks. The output sequence from h_c is denoted $c(t)$ and $c(t) \in \{1, 2, 3, 4\}$, i.e., LFSR$_d$ is clocked at

least once and at most four times between consecutive outputs. On average, LFSR$_d$ is clocked $\bar{c} = 2.5$ times.

LFSR$_d$ is chosen to have a primitive feedback polynomial of length 89 which produces a maximal-length sequence with a period of $T = 2^{89} - 1$. The feedback polynomial for LFSR$_d$ is

$$f_d(x) = x^{89} + x^{83} + x^{80} + x^{55} + x^{53} + x^{42} + x^{39} + x + 1\,. \tag{5.24}$$

Ten bits are taken from LFSR$_d$ as input to the function h_d, these bits are taken from the positions $(0, 1, 2, 7, 12, 20, 30, 44, 65, 80)$ of the LFSR. The function h_d is given in truth table form, see [CDF+00].

5.4.2 The Attack Applied on LILI-128

In this chapter we will give a description of how we turn our ideas described in this chapter into a key recovery attack on LILI-128. The different steps of our attack can be summarized as follows:

- First we find a multiple of low weight of the LFSR$_d$, i.e., find a multiple $g(x)$ of weight w_g such that $g(x) = f_d(x)k(x)$ for some polynomial $k(x)$. See Section 3.1.2 for a description on how to find such multiples.
- Secondly, we guess the content of LFSR$_c$. For each guess we perform a distinguishing attack on the output keystream. If the guessed key is the correct key, we will detect a certain bias in the output.
- When we have found the correct starting state of LFSR$_c$, we recover the initial state of LFSR$_d$ by just applying some well known attack, e.g. a time-memory trade off attack, or the weight three attack described in Section 5.3.

After calculation of one (or several) multiple(s) of the feedback polynomial of LFSR$_d$, our first step is to guess the initial state of LFSR$_c$. If we guess the correct key the clocking of LFSR$_d$ is correct and we should be able to detect some bias in the keystream. To detect this bias we apply our distinguishing attack on the keystream. If we instead made an incorrect guess, the output sequence will have properties like a random source. For each guess of LFSR$_c$ we need to make a decision whether this is the correct key or not. LILI-128 uses 10 bits as input to the Boolean function h_d. For a weight w multiple we get

$$(z_t, z_{t+\tau_1}, \ldots, z_{t+\tau_{w-1}}) = \big(h_d(\boldsymbol{s_t}), h_d(\boldsymbol{s_{t+\tau_1}}), \ldots, h_d(\bigoplus_{i=0}^{w-2} \boldsymbol{s_{t+\tau_i}})\big), \tag{5.25}$$

where $\boldsymbol{s}_{t+\tau_l}$ is a column vector including the ten inputs to h_d. If we consider the fact that we have an irregularly clocked LFSR, not all of the terms in (5.2) will be used to produce an output bit. If so, we cannot use this relation. Thus we will need more keystream to be able to get the required number of valid vectors we want. As LFSR$_d$ is clocked on average 2.5 times between consecutive outputs we can expect that we need to increase the keystream by roughly a factor $(2/5)^{w-1}$. (This is valid for the case of one weight w parity check. If we would consider all weight w parity checks up to a certain length, we do not need to increase the keystream length at all in the case of irregular clocking.)

We will now describe the details of the attack applied to LILI-128 using multiples, $g(x)$, of the feedback polynomial, $f_d(x)$, of weight, $w_g \in \{3, 4, 5\}$. In the attacks we will compare the results to the traditional attack where the non-linear filter function is approximated with a linear function. In the case of LILI-128, the best such linear approximation is

$$h'_d(x_1, \ldots, x_{10}) = x_1 + x_2 + x_4 + x_5. \tag{5.26}$$

This equation holds with the probability

$$\Pr\big(x_1 + x_2 + x_4 + x_5 = h_d(x_1, \ldots, x_{10})\big) = \frac{1}{2} + 2^{-5}. \tag{5.27}$$

The bias of this approximation is denoted as

$$\varepsilon = \Pr\big(x_1 + x_2 + x_4 + x_5 = h_d(x_1, \ldots, x_{10})\big) - \frac{1}{2} = 2^{-5}. \tag{5.28}$$

Using the piling-up lemma introduced in Section 4.1, for a weight w_g multiple, gives the following expression for the Squared Euclidean Imbalance

$$\Delta(\mathcal{D}_\mathcal{C}) = 2^{2w_g}\varepsilon^{2w_g} = 2^{-8w_g}. \tag{5.29}$$

5.4.3 Results with a Weight Three Multiple

We first use the method described in Section 3.1.2 on LILI-128 to find a multiple of weight three. In this case we have the degree of the original feedback $r_f = 89$ and $w_g = 3$. Hence the degree of the multiple is approximately $r_g = 2^{44.5}$. According to Section 3.1.2, the complexity to find one multiple of weight three is approximately 2^{50}. If we use the distinguishing attack described above on a regularly clocked LFSR$_d$, the bias is $\Delta(\mathcal{D}_\mathcal{C}) = 2^{-16}$ which is greater than we would usually expect. Using the piling-up lemma we would expect $\Delta(\mathcal{D}_\mathcal{C}) = 2^{-24}$.

This means that in our attack, we will need approximately $11/\Delta(\mathcal{D}_\mathcal{C}) = 2^{19.46}$ keystream bits to distinguish it from a stream of random data. We thus

need slightly more than $2^{44.5}/2.5$ bits of received sequence. The complexity for the attack is $2^{38} \cdot 2^{19} \cdot 2.5^2$ as we search through the initial states of LFSR$_c$, each of these states takes 2^{19} bits to distinguish. To get 2^{19} sample values, we need to use $2^{19} \cdot 2.5^2 \approx 2^{22}$ parity checks in the case of irregular clocking. The total complexity is about $2^{60.1}$.

5.4.4 Results with a Weight Four Multiple

To find a multiple of weight four we use the same method as before. In this case we have the degree of the original feedback $r_f = 89$ and $w_g = 4$, hence the degree of the multiple is approximately $r_g = 2^{29.67}$. The complexity to find this multiple is 2^{65}. The bias is calculated to $\Delta(\mathcal{D}_\mathcal{C}) = 2^{-16.14}$. The expected bias using the piling-up lemma is in this case $\Delta(\mathcal{D}_\mathcal{C}) = 2^{-32}$.

We will need approximately $11/\Delta(\mathcal{D}_\mathcal{C}) = 2^{19.60}$ keystream bits in the case when all parities are available. For the uncertainty of positions actually appearing in the output stream, we use the same argument as above. In total we need 2^{22} bits of keystream. As the degree of the polynomial is 2^{30} we will need about $2^{30}/2.5$ bits to detect the bias. The complexity for the attack is about $2^{60.2}$.

5.4.5 Results with a Weight Five Multiple

To find a multiple of weight five we need a degree of the multiple of approximately $2^{17.8}$, the computational complexity to find it is approximately 2^{50}. With a multiple of weight five the bias decreases significantly to $\Delta(\mathcal{D}_\mathcal{C}) = 2^{-32}$ and hence we will need approximately $11/\Delta(\mathcal{D}_\mathcal{C}) = 2^{35.46}$ available checks. We need in total $2^{38.10}$ bits to perform the distinguishing attack. The complexity for the attack is around $2^{76.10}$.

5.4.6 The Weight Three Attack

We can improve the results above by using several low weight parity checks as described in Section 4.4. We do not present the results here, but consider only the modified attack described in Section 5.3. In earlier sections we have stated that the multiples of weight three start to appear at degree $2^{44.5}$. If we use about 15 valid parity equations in the weight three attack, the probability that we make an incorrect decision is low. The total complexity for this attack is roughly 2^{53} as we still guess the contents of LFSR$_c$ 2^{38} times on average, and for each guess we need to test whether the set of equalities $z_{t+\tau_1} = z_{t+\tau_2}, \ldots$ is true. As the probability of such an event occurring for the correct key is $> 2^{-10}$ we run through a bit more, say 2^{12} such t values. As all but one test correspond to the random case, the equalities will hold with probability $1/2$. Hence we need very few comparisons on average (say

2). We also need to include the fact that not all positions are present, a factor 2.5^2. The required keystream length to find 15 valid weight three parity checks is roughly 2^{47}.

5.4.7 Summary of Attack on LILI-128

In Table 5.2 we summarize our result. Note that the work to synchronize po-

		Sequence length	Complexity
Weight three attack		2^{47}	2^{53}
Basic attack with weight	3	2^{43}	2^{60}
	4	2^{29}	2^{60}
	5	2^{35}	2^{76}

Table 5.2: The sequence length and the complexity for different weights, the basic attack is described in Section 5.4.2, and the weight three attack is described in Section 5.4.6.

sitions for each guessed LFSR_c state was not considered in previous work, but can be done without increasing the overall complexity by choosing states in the order they appear in the LFSR_c cycle, see [Mol04].

We compare with the best attacks so far, summarized in Table 5.3. Here we have recalculated the complexity of Saarinen's attack [Saa02] to bit operations, we have also recalculated the complexity of the fast algebraic attack by Courtois [Cou03] according to the suggestions by Hawkes and Rose [HR04].

5.5 Summary

We have presented a new simple attack philosophy on filter generators and related ciphers. The attack uses low weight multiples of the original feedback polynomial. The components of the low weight parity checks are considered as vectors that will be non-uniformly distributed due to the parity checks. We demonstrated the efficiency by attacking LILI-128. We can recover the key using 2^{47} keystream bits with complexity around 2^{53}, an improvement compared to previous attacks. The weight three attack is a very

Attack by	Our	[Saa02]	[CM03]	[Cou03]	[TSS+05]	[JJ02]
Keystream	2^{47}	2^{46}	2^{18}	2^{60}	2^{7}	2^{30}
Complexity	2^{53}	2^{60}	2^{96}	2^{40}	2^{102}	2^{71}
Memory	2^{44}	2^{51}	2^{43}	2^{24}	–	2^{46}

Table 5.3: Comparison of the weight three attack with the best known attacks.

powerful key recovery attack on any filter generator, if enough output symbols are available. It also applies to any filter generator with a weight three feedback polynomial, by the squaring method.

6

Cryptanalysis of Irregularly Clocked Filter Generators

An irregularly clocked filter generator is a generator built on a filter generator as described in the previous chapter. But to prevent correlation attacks, the output from the filter function is decimated in some unpredictable way. A binary LFSR produces a sequence denoted $s_0, s_1, \ldots$. From the LFSR, d input bits are fed into a filter function h. A clock control sequence denoted $c_0, c_1, \ldots$ is generated in some arbitrary way. This clock control sequence is used to decimate the output from the filter function, h, i.e., to remove some symbols from the stream. The decimated sequence is denoted $\boldsymbol{z} = z_0, z_1, \ldots$ and is used as the keystream. The clock control sequence uniquely determines a sequence of integers $k_0, k_1, \ldots$ such that

$$z_t = h(s_{k_t}, s_{k_t+\tau_1}, s_{k_t+\tau_2}, \ldots, s_{k_t+\tau_{d-1}}), \tag{6.1}$$

where $t \geq 0$ and $k_0 < k_1 < k_2 < \ldots$, and τ_i is the distance from the first filter tap to the i^{th} input tap of the LFSR. In other words, after observing $z_0, z_1, \ldots, z_N$ the LFSR has been clocked $k_N = \sum_{t=0}^{N} c_t$ times. See Figure 6.1 for and illustration of irregularly clocked filter generators. As usual, the produced keystream is XORed to the plaintext to produce the ciphertext. Note that many well known designs can be described by this model, e.g., the shrinking generator, self-shrinking generator, alternating step generator, LILI-128 and LILI-II.

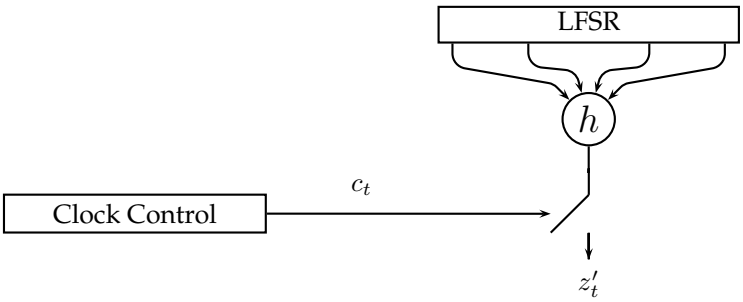


Figure 6.1: Irregularly clocked Filter Generator.

In this chapter we will describe a distinguishing attack that uses a low weight multiple of the linear feedback shift registers (LFSR), i.e., it belongs to the class of linear distinguishers, see [GO94, Gol95]. It collects statistics from sliding windows around the positions of the keystream, where the members of this recursion are likely to appear. The strength of the attack is the updating procedure used when moving the windows, this procedure allows us to receive many new samples with very few operations.

We will discuss some variants of the attack and finally demonstrate the effectiveness on the stream cipher LILI-II. LILI-II [CDF+02] was designed by Clark, Dawson, Fuller, Golić, Lee, Millan, Moon and Simpson and first published in ACISP 2002. It is the improved successor of LILI-128, described in Section 5.4. LILI-II uses a 128 bit key which is expanded and used with a much larger internal state, namely 255 bits instead of the 128 bits used in LILI-128. To distinguish the cipher from a random source we need 2^{104} bits of keystream and the complexity is around 2^{104} operations. This is the first attack on LILI-II faster than exhaustive key search.

The chapter is based in [EJ05] and partly on [EJ06]. The chapter is organized as follows, in Section 6.1 we describe how to build our distinguisher. In Section 6.2 we discuss further improvements. We then show how the proposed attack can be applied to LILI-II in Section 6.3. The chapter is summarized in Section 6.4.

6.1 An Efficient Distinguisher

As we are attacking irregularly clocked ciphers we will not know exactly where the symbols from the LFSR sequence will be located in the output keystream, not even if they appear at all.

Our approach is the following. We will fix one position in the recurrence relation, and around the estimated location of the other symbols we

will use sliding windows. When using windows of adequate size we have a high probability that all the symbols in the relation (if not removed by the irregular decimation) are included. We will then calculate how many symbols from the different windows sum to zero. Only one of these combinations contribute with a bias, the others will appear as random samples. In the following subsections we will describe the different steps we use in our attack.

The way we build the distinguisher is influenced by previous work on distinguishers, see for example [GM03b, GO94, Gol95, CHJ02, EJ02].

6.1.1 Finding a Low Weight Multiple

The success of our attack depends on the use of low weight recurrence relations. Hence the first step is to find a low weight multiple of the LFSR. In the attack we use a multiple of weight three of the original feedback polynomial. It is also possible to mount the attack with multiples of higher weight. Using a multiple of higher weight lowers the degree of the multiple, but it also lowers the probability that all symbols in a recurrence are included after the decimation. In general it also lowers the correlation of the Boolean function. So from now on we assume that we use a weight three recurrence relation. We will use the methods described in Section 3.1.2 to find the multiple.

6.1.2 Calculating the Correlation of h for a Weight Three Recursion

Assume that we have a weight three relation for the LFSR sequence according to

$$s_t \oplus s_{t+\tau_1} \oplus s_{t+\tau_2} = 0. \tag{6.2}$$

Let $\boldsymbol{s_t} = (s_t, s_{t+t_1}, \ldots, s_{t+t_{d-1}})$ denote the input bits to h at time t taken from positions $t, t+t_1, \ldots, t+t_{d-1}$, i.e., $z_t = h(\boldsymbol{s}_t)$. The correlation (see Equation (5.7)) for a weight three recurrence relation of the nonlinear Boolean function, h, is

$$\Pr\Big(h(\boldsymbol{s_t}) \oplus h(\boldsymbol{s_{t+\tau_1}}) \oplus h(\boldsymbol{s_{t+\tau_2}}) = 0 \mid \boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_1}} \oplus \boldsymbol{s_{t+\tau_2}} = \boldsymbol{0},\ \forall t \geq 0\Big) = \\ = \tfrac{1}{2} + \varepsilon_h. \tag{6.3}$$

If the LFSR would be regularly clocked ($c_t = 1,\ \forall t$), the probability above is equivalent to

$$\Pr\big(z_t \oplus z_{t+\tau_1} \oplus z_{t+\tau_2} = 0 \mid \boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_1}} \oplus \boldsymbol{s_{t+\tau_2}} = \boldsymbol{0},\ \forall t \geq 0\big). \tag{6.4}$$

The correlation can be calculated by simply trying all possible input combinations into the function. As we use a weight three recursion some

combinations will not be possible and the distribution will be biased. The correlation of Boolean functions was also discussed in the previous chapter.

6.1.3 The Positions of the Windows

Consider again the weight three relation, but now with irregular clocking. Let $E(C)$ denote the expected value of the clocking sequence, $c_t,\ t \geq 0$. The size of the windows depends on the distance from the fixed position, hence we will fix the center position in the recurrence and use windows around the two other positions. We rewrite the recurrence as

$$s_{t-\tau_1} \oplus s_t \oplus s_{t+\tau_2-\tau_1} = 0. \tag{6.5}$$

The expected distance between the output from h, corresponding to input $\boldsymbol{s}_{t-\tau_1}$ and $\boldsymbol{s}_t$, is $\tau_1/E(C)$ as the sequence is decimated, similarly the distance between $\boldsymbol{s}_t$ and $\boldsymbol{s}_{t+\tau_2-\tau_1}$ is $\frac{\tau_2-\tau_1}{E(C)}$. Figure 6.2 illustrates how we position the windows of size L_1 and L_2 in the case of a weight three recursion.

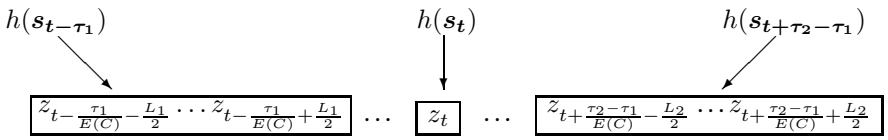


Figure 6.2: Illustration of the window positions in the case with a weight three recurrence relation.

6.1.4 Determining the Size of the Windows

The output sequence from the clock-control part, c_t, is assumed to have a fixed distribution independent of t. By using the central limit theorem we know that the sum of a large number of random variables approaches the normal distribution. So $Y_n = C_1 + C_2 + \ldots + C_n$ is distributed approximately as $\mathrm{N}(n \cdot E(C), \sigma_c\sqrt{n})$, where n denotes the number of observed symbols, $E(C)$ the expected value of the clocking sequence and σ_c the standard deviation of the clocking sequence.

If the windows are sufficiently large, the correct position of the symbol will be located inside the window with a high probability, for example

$$\begin{aligned} \Pr(nE(C) - \sigma_c\sqrt{n} < Y_n < nE(C) + \sigma_c\sqrt{n}) = &\quad 0.682, \\ \Pr(nE(C) - 2\sigma_c\sqrt{n} < Y_n < nE(C) + 2\sigma_c\sqrt{n}) = &\quad 0.954. \end{aligned} \tag{6.6}$$

Continuing, we choose a window size of four standard deviations. The size of window i is denoted by L_i, i.e., $L_i \approx 4\sigma_c\sqrt{n}$.

6.1.5 Estimating the Number of Bits We Need to Observe

The main idea of the distinguishing attack is to create samples of the form

$$z_{p_1} \oplus z_t \oplus z_{p_2}, \tag{6.7}$$

where p_1 is any position in the first window and p_2 is any position in the second window. We will run through all such possible combinations. As will be demonstrated, each sample is drawn according to a biased distribution.

To determine how many bits we need to observe to reliably distinguish the cipher from a random source, we need to make an estimate of the bias. First we consider the case of a regularly clocked cipher. We denote the window sizes by L_1 and L_2. In the following estimations we have to remember that we are calculating samples, and that for every time instant we get $L_1 \cdot L_2$ new samples. Among all of the $L_1 \cdot L_2$ constructed samples, one sample will correspond to the actual recurrence equation and thus sum to zero according to the correlation of the filter function (see (6.3)). Hence, only this one sample contributes with the bias ε_h. All the other $L_1 \cdot L_2 - 1$ samples are assumed to be random samples. The distribution for our samples can roughly be given as

$$\begin{cases} \Pr(z_{p_1} \oplus z_t \oplus z_{p_2} = 0) = \frac{1}{2} + \frac{\varepsilon_h}{L_1 L_2}, \\ \Pr(z_{p_1} \oplus z_t \oplus z_{p_2} = 1) = \frac{1}{2} - \frac{\varepsilon_h}{L_1 L_2}, \end{cases} \tag{6.8}$$

assuming that $s_{t-\tau_1}$ and $s_{t+\tau_2-\tau_1}$ always appear inside the windows.

When we have irregular clocking, the output from the LFSR is decimated, i.e., some terms might not contribute to the output sequence. The probability that the end two terms of a weight three recurrence relation is included in the keystream and in the windows is denoted by p_{dec}. In the approximation we neglect the probability that the result in some cases deviates more than two standard deviations from the expected position, i.e., the component lies outside the window. Hence, the distribution $\mathcal{D}_\mathcal{C}$ of our samples can be estimated as

$$\begin{cases} \mathcal{D}_\mathcal{C}(0) = \frac{1}{2} + \frac{\varepsilon_h p_{dec}}{L_1 L_2}, \\ \mathcal{D}_\mathcal{C}(1) = \frac{1}{2} - \frac{\varepsilon_h p_{dec}}{L_1 L_2}. \end{cases}, \tag{6.9}$$

where we will refer to

$$\varepsilon_{tot} = \frac{\varepsilon_h p_{dec}}{L_1 L_2} \tag{6.10}$$

as the bias. This approximation has been compared with simulation results and works well, see Section 6.3.1. The Squared Euclidean Imbalance of our

samples can be expressed as

$$\Delta(\mathcal{D}_{\mathcal{C}}) = \frac{4\varepsilon_h^2 p_{dec}^2}{L_1^2 L_2^2}. \tag{6.11}$$

In the approximation we have estimated that the probability for the position of the taps inside the windows is uniform. The purpose is to make the updating procedure when moving the windows as efficient as possible, which in fact is the strength of the attack. A better approximation would be to weigh the positions inside the window according to the normal distribution, e.g., see Daemen and Assche [DA06]. This might decrease the number of symbols needed, but would make an efficient updating procedure much more difficult.

As stated in Section 3.1.3.2 the number of samples we need to observe is $N \approx 11/\Delta(\mathcal{D}_{\mathcal{C}})$. Thus, the number of samples our distinguisher require can be estimated as

$$\frac{11}{4} \cdot \frac{L_1^2 L_2^2}{\varepsilon_h^2 p_{dec}^2}. \tag{6.12}$$

At each time instant we receive $L_1 \cdot L_2$ new samples, and hence the total number of bits we need for the distinguisher can be estimated by (6.13).

$$N \approx \frac{11}{4} \cdot \frac{L_1 L_2}{\varepsilon_h^2 p_{dec}^2}. \tag{6.13}$$

The above Equations (6.12-6.13) assume independent samples. This is not true in our case, but the equation is still a good approximation of the number of samples needed in the attack.

6.1.6 Complexity of Calculating the Samples

The strength of the distinguisher is that the calculation of the number of ones and zeros in the windows can be performed very efficiently; when we move the first position from z_t to z_{t+1} we also move the windows one step to the right.

We denote the number of zeros in window one at time instant t by $\mathcal{D}_t^{\boldsymbol{z}_{win_1}}$, and similarly we denote the number of zeros in windows two by $\mathcal{D}_t^{\boldsymbol{z}_{win_2}}$. The number of samples that fulfill $z_{p_1} \oplus z_t \oplus z_{p_2} = 0$ at time t is denoted $\mathcal{D}_t^X$, where p_1, p_2 are some positions in window one respectively window two. Hence, when moving the windows we get the new number of zeros $\mathcal{D}_{t+1}^{\boldsymbol{z}_{win_1}}$ and $\mathcal{D}_{t+1}^{\boldsymbol{z}_{win_2}}$ by subtracting the first bit in the old window and adding the new bit included in the window, e.g., for window one,

$$\mathcal{D}_{t+1}^{\boldsymbol{z}_{win_1}} = \mathcal{D}^{\boldsymbol{z}_{win_1}} - z_{t-\frac{\tau_1}{E(C)}-\frac{L_1}{2}} + z_{t-\frac{\tau_1}{E(C)}+\frac{L_1}{2}+1}, \tag{6.14}$$

and similarly for window two. From the $\mathcal{D}_{t+1}^{\boldsymbol{z}^{win_1}}$ and $\mathcal{D}_{t+1}^{\boldsymbol{z}^{win_2}}$ we can, with very few basic computations calculate $\mathcal{D}_{t+1}^{X}$.

We define one operation as the computations required to calculate $\mathcal{D}_{t+1}^{X}$ from $\mathcal{D}_{t+1}^{\boldsymbol{z}^{win_1}}$ and $\mathcal{D}_{t+1}^{\boldsymbol{z}^{win_2}}$.

Theorem 6.1: The proposed distinguisher requires $N = \frac{11}{4} \cdot \frac{L_1 L_2}{\varepsilon_h^2 p_{dec}^2}$ bits of keystream and uses a computational complexity of approximately N operations.

Although the number of zeros in the windows $\mathcal{D}_{t+1}^{\boldsymbol{z}^{win_1}}, \mathcal{D}_{t+1}^{\boldsymbol{z}^{win_2}}$ are dependent on the previous number of zeros in the window $\mathcal{D}_{t}^{\boldsymbol{z}^{win_1}}, \mathcal{D}_{t}^{\boldsymbol{z}^{win_1}}$, the covariance between the number of samples received at time instant t and $t+1$ is zero, $\mathrm{Cov}(\mathcal{D}_{t+1}^{X}, \mathcal{D}_{t}^{X}) = 0$, see Appendix A.

6.1.7 Hypothesis Testing

The last step in the attack is to determine whether the collected data really is biased. A rough method for the hypothesis test is to check whether the result deviates more than two standard deviations from the expected result in the case when the bits are truly random. The standard deviation for a sum of these samples, can be estimated by $\sigma = \sqrt{N \frac{L_1 L_2}{4}}$, see Appendix A, where L_1 and L_2 are the sizes of the windows and N is the number of bits of keystream we observe.

6.1.8 Summary of the Attack

In Figure 6.3 we summarize the attack, where $\mathcal{D}_{t}^{\boldsymbol{z}^{win_1}}$ denotes the number of zeros in window one, $\mathcal{D}_{t}^{\boldsymbol{z}^{win_2}}$ the number of zeros in window two. The total sum of the samples is denoted by $\mathcal{D}^X$, i.e., $\mathcal{D}^X = \sum_{t=0}^{N-1} \mathcal{D}_t^X$. $L_1,\ L_2$ denotes the sizes of window one respectively window two.

6.2 Possible Improvements Using Vectorial Samples

In the previous section an efficient distinguisher was created by adding binary symbols from different windows in the keystream. The natural step to improve this strategy is to consider a word based attack. Instead of considering the number of zeros and ones inside the windows, we increase the alphabet size and count words (vectors of consecutive bits) inside the windows and around the center member of the weight three recurrence equation.

1. Find a weight three multiple of the LFSR.
2. Calculate the bias ε_h.
3. Determine the positions of the windows.
4. Calculate the sizes $L_1,\ L_2$ of the windows.
5. Estimate the number of bits N we need to observe.
6. **for** t from 0 to N
   if $z_t = 0$
   $\mathcal{D}^X \mathrel{+}= \mathcal{D}_t^{\boldsymbol{z}_{win_1}} \cdot \mathcal{D}_t^{\boldsymbol{z}_{win_2}} + (L_1 - \mathcal{D}_t^{\boldsymbol{z}_{win_1}})(L_2 - \mathcal{D}_t^{\boldsymbol{z}_{win_2}})$
   else if $z_t = 1$
   $\mathcal{D}^X \mathrel{+}= \mathcal{D}_t^{\boldsymbol{z}_{win_1}}(L_2 - \mathcal{D}_t^{\boldsymbol{z}_{win_2}}) + (L_1 - \mathcal{D}_t^{\boldsymbol{z}_{win_1}})\mathcal{D}_t^{\boldsymbol{z}_{win_2}}$
   end if
   Move window and calculate $\mathcal{D}_{t+1}^{\boldsymbol{z}_{win_1}},\ \mathcal{D}_{t+1}^{\boldsymbol{z}_{win_2}}$
   end for
7. **if** $|\mathcal{D}^X - N \cdot \frac{L_1 \cdot L_2}{2}| > \sqrt{N \cdot L_1 \cdot L_2}$
   output **cipher**
   else
   output **random**.

Figure 6.3: Summary of the proposed distinguishing attack.

6.2.1 An Efficient Distinguisher Using Vectors

In this section the idea of using words is applied directly on the keystream, i.e., a length two word would be of the form (z_t, z_{t+1}).

We start by considering the calculation and storage of the number of different words inside the two windows. Let $\mathcal{D}_t^{\boldsymbol{z}_{win_1}}$ now denote the empirical distribution of words in window one and $\mathcal{D}_t^{\boldsymbol{z}_{win_2}}$ the same in window two. The word at the center position of the recurrence relation is denoted $\boldsymbol{z}_t$. Note that we can maintain an effective updating procedure for these distributions, i.e., $\mathcal{D}_{t+1}^{\boldsymbol{z}_{win_1}}$ and $\mathcal{D}_{t+1}^{\boldsymbol{z}_{win_2}}$ are easily updated from $\mathcal{D}_t^{\boldsymbol{z}_{win_1}}$ and $\mathcal{D}_t^{\boldsymbol{z}_{win_2}}$.

The disadvantage of the proposed procedure, compared to the previous method is that the complexity of calculating the overall number of samples of a certain value increases with larger word size. Using a larger alphabet means a higher computational complexity to calculate the number of ways to add the words in the different windows. On the other hand, we expect a higher bias in a word based approach, giving a shorter required keystream.

The main idea of the distinguishing attack using word size b is to create samples of the form

$$\begin{gathered}(z_{p_1}, z_{p_1+1}, \ldots, z_{p_1+b-1}) \oplus (z_t, z_{t+1}, \ldots, z_{t+b-1}) \oplus \\ \oplus (z_{p_2}, z_{p_2+1}, \ldots, z_{p_2+b-1}),\ \ t \geq 0,\end{gathered} \tag{6.15}$$

where the word $(z_{p_1}, z_{p_1+1}, \ldots, z_{p_1+b-1})$ is any word inside the first window and $(z_{p_2}, z_{p_2+1}, \ldots, z_{p_2+b-1})$ is any word inside the second window. The total number of samples created at time t is $L_1 \cdot L_2$, where now L_1 is the number of words inside the first window, etc.

Consider the two stored tables $\mathcal{D}_t^{\boldsymbol{z}^{win_1}}$ and $\mathcal{D}_t^{\boldsymbol{z}^{win_2}}$ containing the number of different words inside each window as empirical distributions. Then at each time instant in the attack we need to perform a convolution of two such distributions. This is a time consuming operation if the distributions are large. A trivial calculation of the convolution of two distributions of size n has complexity $\mathcal{O}(2^{2n})$. However, it has been demonstrated, see for example [MJ05a] that convolutions over bitwise addition of two large distribution can be performed via Fast Hadamard Transform (FHT) with complexity $\mathcal{O}(n \cdot 2^n)$.

The proposed attack using words is summarized in Figure 6.4. The distribution $\mathcal{D}_{\mathcal{C}}(i),\ \forall i \in \mathbb{F}_{2^b}$ is determined through simulation before the attack, and needs only be done once. A convolution of $\mathcal{D}_t^{\boldsymbol{z}^{win_1}}(i)$ and $\mathcal{D}_t^{\boldsymbol{z}^{win_2}}(j)$ is performed at each time instant, to decrease the complexity this step can be performed via the Fast Hadamard Transform.

1. Find a weight three multiple of the LFSR.
2. Determine the positions of the windows.
3. Calculate the sizes $L_1,\ L_2$ of the windows.
4. Estimate the number of bits N we need to observe.
5. **for** t from 0 to N
   for i from 0 to 2^b
   for j from 0 to 2^b
   $\mathcal{D}^X(i \oplus j \oplus \boldsymbol{z}_t) += \mathcal{D}_t^{\boldsymbol{z}^{win_1}}(i) \cdot \mathcal{D}_t^{\boldsymbol{z}^{win_2}}(j)$
   end for
   end for
   Move window and calculate $\boldsymbol{z}_{t+1},\ \mathcal{D}_{t+1}^{\boldsymbol{z}^{win_1}}$ and $\mathcal{D}_{t+1}^{\boldsymbol{z}^{win_2}}$.
   end for
6. Calculate $LLR = \sum_{i \in \mathbb{F}_2^b} \mathcal{D}^X(i) \cdot \log_2\left(\frac{\mathcal{D}_{\mathcal{C}}(i)}{2^{-b}}\right)$
   if $LLR \geq 0$
   output **cipher**
   else
   output **random**.

Figure 6.4: Summary of the proposed distinguishing attack using words.

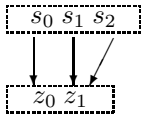
Simulation results show that using vectors indeed results in a more bi-

ased distribution. However, the biggest contribution seem to come from an increased dependence between the vectors, i.e., even for a random stream some small bias might be found. The proposed method seems to offer little advantage compared to the binary method, considering the increase in computational complexity required. However, there might be specific cases where the word based attack offers a significant improvement.

6.2.2 Considering Words In The Undecimated Sequence

Instead of considering words in the keystream sequence we can estimate the distribution of words in the undecimated sequence of LFSRs. These words can have a different word size than in the decimated sequence. A word in the keystream is as before denoted $\boldsymbol{z_t} = (z_t, z_{t+1}, \ldots, z_{t+b-1})$, and a corresponding word of size d in the undecimated LFSR stream is denoted by $\boldsymbol{s_t} = (s_t, s_{t+1}, \ldots, s_{t+d-1})$.

Consider the output from a Step 1/Step 2 generator depicted in Figure 6.5. In this figure we consider keystream words of size two and the decima-



$$
\begin{aligned}
\Pr(\boldsymbol{s} = z_0 z_1 z_1 | \boldsymbol{z} = z_0 z_1) &= 0.5 \\
\Pr(\boldsymbol{s} = z_0 z_1 z_1' | \boldsymbol{z} = z_0 z_1) &= 0.25 \\
\Pr(\boldsymbol{s} = z_0 z_1' z_1 | \boldsymbol{z} = z_0 z_1) &= 0.25 \\
\Pr(\boldsymbol{s} = z_0 z_1' z_1' | \boldsymbol{z} = z_0 z_1) &= 0 \\
\Pr(\boldsymbol{s} = z_0' z_1 z_1 | \boldsymbol{z} = z_0 z_1) &= 0 \\
\Pr(\boldsymbol{s} = z_0' z_1 z_1' | \boldsymbol{z} = z_0 z_1) &= 0 \\
\Pr(\boldsymbol{s} = z_0' z_1' z_1 | \boldsymbol{z} = z_0 z_1) &= 0 \\
\Pr(\boldsymbol{s} = z_0' z_1' z_1' | \boldsymbol{z} = z_0 z_1) &= 0
\end{aligned}
$$

Figure 6.5: Keystream generated using $c_t \in \{1, 2\}$.

tion is done according to $c_t \in \{1, 2\}$ with equal probability. The distribution of the undecimated words will be skew, e.g., $\Pr(\boldsymbol{s} = 111 | \boldsymbol{z} = 00) = 0$. The probabilities $\Pr(\boldsymbol{s_t} | \boldsymbol{z_t})$, $\forall \boldsymbol{s_t}, \boldsymbol{z_t}$, can be precomputed. For simplicity, our considerations are purely combinatorial. Each observed $\boldsymbol{z}$ vector in a window will give rise to a number of samples for the $\boldsymbol{s}$ vector. In the previous example, an observed $\boldsymbol{z} = (00)$ would give 4 possible samples for $\boldsymbol{s}$, namely two samples with value $\boldsymbol{s} = (000)$ one with value $\boldsymbol{s} = (001)$ and one with value $\boldsymbol{s} = (010)$.

When performing the attack, as in the previous section, windows are placed around the expected positions of members in the recurrence relation. Let us denote the empirical distributions of the corresponding undecimated words of size d as $\mathcal{D}_t^{\boldsymbol{s}_{win_1}}$, $\mathcal{D}_t^{\boldsymbol{s}_{win_2}}$ and $\mathcal{D}_t^{\boldsymbol{s}_{center}}$.

The empirical distribution of undecimated words in the windows, $\mathcal{D}_t^{\boldsymbol{s}_{win_1}}$

and $\mathcal{D}_t^{\boldsymbol{s}_{win_2}}$ are calculated and, as before, the update of these tables can be done very efficiently when moving the windows. The update is performed for the two windows at each time instant t. For the center word the table $\mathcal{D}_t^{\boldsymbol{s}_{center}}$ is obtained by a simple table lookup.

In the cipher case, we are looking at an equation of the form

$$\boldsymbol{s}_{t-\tau_1} \oplus \boldsymbol{s}_t \oplus \boldsymbol{s}_{t+\tau_2-\tau_1} = 0, \;\; t \geq 0. \tag{6.16}$$

The final procedure is almost exactly as in the previous section. We have an empirical distribution of $\boldsymbol{s}_{t-\tau_1}$ in the array $\mathcal{D}_t^{\boldsymbol{s}_{win_1}}$, etc., and we estimate the distribution of $\boldsymbol{s}_{t-\tau_1} \oplus \boldsymbol{s}_t \oplus \boldsymbol{s}_{t+\tau_2-\tau_1}$ through the convolution of $\mathcal{D}_t^{\boldsymbol{s}_{win_1}}$, $\mathcal{D}_t^{\boldsymbol{s}_{win_2}}$ and $\mathcal{D}_t^{\boldsymbol{s}_{center}}$. This distribution is denoted $\mathcal{D}_t^X$. By computing the convolution via Fast Hadamard Transform the complexity of these calculations can, as mentioned in Section 6.2.1, be significantly lowered.

Finally, by summing over all t, we obtain the empirical distribution $\mathcal{D}^W$. As before, this needs to be checked against the uniform distribution and possibly some experimentally verified $\mathcal{D}_{\mathcal{C}}$. An outline of the proposed attack is depicted in Figure 6.6.

1. Find a weight three multiple of the LFSR.
2. Determine the positions of the windows.
3. Calculate the sizes $L_1,\ L_2$ of the windows.
4. Estimate the number of bits M we need to observe.
5. **for** t from 0 to M
   for i from 0 to 2^d
   for j from 0 to 2^d
   for k from 0 to 2^d
   $\mathcal{D}^X(i \oplus j \oplus k) += \mathcal{D}_t^{\boldsymbol{s}_{win_1}}(i) \cdot \mathcal{D}_t^{\boldsymbol{s}_{win_2}}(j) \cdot \mathcal{D}_t^{\boldsymbol{s}_{center}}(k)$
   end for
   end for
   end for
   Move window and calculate $\mathcal{D}_{t+1}^{\boldsymbol{s}_{win_1}}$, $\mathcal{D}_{t+1}^{\boldsymbol{s}_{win_2}}$, and $\mathcal{D}_{t+1}^{\boldsymbol{s}_{center}}$
   end for
6. Calculate $LLR = \sum_{i \in \mathbb{F}_{2^b}} \mathcal{D}^X(i) \cdot \log_2\left(\frac{\mathcal{D}_{\mathcal{C}}(i)}{2^{-b}}\right)$
   if $LLR \geq 0$
   output **cipher**
   else
   output **random**.

Figure 6.6: The attack considering undecimated sequences.

Similarly as for using the vectors in the keystream, simulations on a Step

1/Step 2 generator have shown that the gain of using vectors in the undecimated sequence is small. However, on other ciphers the gain might be larger.

6.3 Cryptanalysis of LILI-II

LILI-II [CDF+02] is the successor of the NESSIE candidate stream cipher LILI-128 [CDF+00]. Many attacks where found on LILI-128 such as the attack described in Section 5.4, but also [MH04, JJ02, CM03, Cou03]. The many attacks on LILI-128 motivated a larger internal state, which is the biggest difference between the two ciphers, LILI-II also use a nonlinear Boolean function, h_d, with 12 input bits instead of 10 as in LILI-128.

Both the members of the LILI family are binary stream cipher that use irregular clocking. They consists of an LFSR$_c$, that via a nonlinear function clocks a second LFSR, called LFSR$_d$, irregularly. The structured can be viewed in Figure 6.7. LILI-II use a key length of 128 bits, the key is ex-

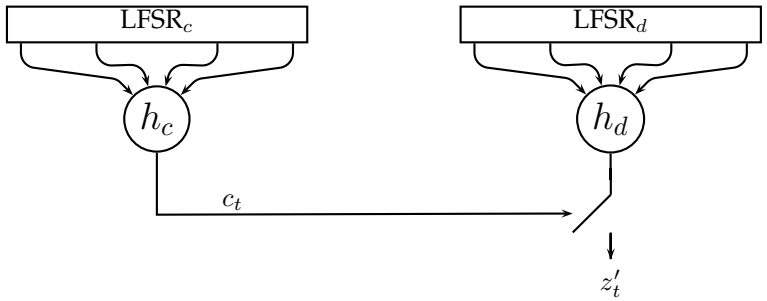


Figure 6.7: LILI-128.

panded and used to initialize the two LFSRs. The first shift register, LFSR$_c$ is a primitive polynomial of length 128, and hence has a period of $2^{128} - 1$. The feedback polynomial for LFSR$_c$ is given by

$$\begin{aligned} f_c(x) = \; & x^{128} + x^{126} + x^{125} + x^{124} + x^{123} + x^{122} + x^{119} + x^{117} + x^{115} + x^{111} + \\ & x^{108} + x^{106} + x^{105} + x^{104} + x^{103} + x^{102} + x^{96} + x^{94} + x^{90} + x^{87} + x^{82} + \\ & x^{81} + x^{80} + x^{79} + x^{77} + x^{74} + x^{73} + x^{72} + x^{71} + x^{70} + x^{67} + x^{66} + x^{65} + \\ & x^{61} + x^{60} + x^{58} + x^{57} + x^{56} + x^{55} + x^{53} + x^{52} + x^{51} + x^{50} + x^{49} + x^{47} + \\ & x^{44} + x^{43} + x^{40} + x^{39} + x^{36} + x^{35} + x^{30} + x^{29} + x^{25} + x^{23} + x^{18} + x^{17} + \\ & x^{16} + x^{15} + x^{14} + x^{11} + x^{9} + x^{8} + x^{7} + x^{6} + x^{1} + 1. \end{aligned} \tag{6.17}$$

The Boolean function, $h_c : \mathbb{F}_2^2 \rightarrow \{1, 2, 3, 4\}$, takes two input bits from LFSR$_c$. It is specified as

$$h_c(x_0, x_{126}) = 2 \cdot x_0 + x_{126} + 1 \, . \tag{6.18}$$

The output of this function is used to clock LFSR$_d$ irregularly. The output sequence from h_c is denoted c_t and $c_t \in \{1, 2, 3, 4\}$, i.e., LFSR$_d$ is clocked at least once and at most four times between consecutive outputs. The expected value of the clock sequence, denoted $E(C)$, is $E(C) = 2.5$.

LFSR$_d$ is chosen to have a primitive polynomial of length 127 which produces a maximal-length sequence with a period of $T = 2^{127} - 1$. The original polynomial was found not to be primitive, see [CDF+04], and has been changed to

$$\begin{aligned} f_d(x) = \; & x^{127} + x^{121} + x^{120} + x^{114} + x^{107} + x^{106} + x^{103} + x^{101} + x^{97} + x^{96} + \\ & x^{94} + x^{92} + x^{89} + x^{87} + x^{84} + x^{83} + x^{81} + x^{76} + x^{75} + x^{74} + x^{72} + x^{69} + \\ & x^{68} + x^{65} + x^{64} + x^{62} + x^{59} + x^{57} + x^{56} + x^{54} + x^{52} + x^{50} + x^{48} + x^{46} + \\ & x^{45} + x^{43} + x^{40} + x^{39} + x^{37} + x^{36} + x^{35} + x^{30} + x^{29} + x^{28} + x^{27} + x^{25} + \\ & x^{23} + x^{22} + x^{21} + x^{20} + x^{19} + x^{18} + x^{14} + x^{10} + x^{8} + x^{7} + x^{6} + x^{4} + \\ & x^{3} + x^{2} + 1. \end{aligned} \tag{6.19}$$

Twelve bits are taken from LFSR$_d$ as input to the function h_d, these bits are taken from the positions (0,1,2,7,12,20,30,44,65,80,96,122) of the LFSR. The function h_d is given as a truth table. Note that also the Boolean function described in the original proposal was weak and has been replaced, see [CDF+04].

We will now apply the technique described in Section 6.1 step-by-step on LILI-II to show the effectiveness of the procedure.

(*i*) **Finding a Low Weight Multiple:** The first step in the attack is to find a low weight multiple $g(x)$ of $f_d(x)$ such that $g(x) = f_d(x)k(x)$ for some polynomial $k(x)$. We will focus on using a weight three multiple in this section. According to Section 3.1.2 weight three multiples, i.e., $w_g = 3$, of an LFSR of degree $r_f = 127$, will start to appear at the degree $r_g = 2^{64}$. The complexity to find the multiple is approximately 2^{70}, using the polynomial residue technique described in the same section.

If we instead would mount the attack with a weight four multiple the expected degree of the polynomial would be $r_f = 2^{43.19}$, the complexity to find a weight four multiple is about $2^{91.81}$.

(*ii*) **Correlation of h_d:** In Table 6.1 some examples are presented from two clock controlled ciphers, these results are based on a weight three and a weight four recursion. In the case of LILI-128 and LILI-II the correlation of h_d are approximately the same for a weight three relation as for a weight four relation. The probability in (6.3) decreases significantly, i.e., ε_{h_d} decreases, if we use multiples of higher weight than four;

Generator	Number of input bits	ε_{h_d}	
		$w_g = 3$	$w_g = 4$
LILI-128	10	$2^{-9.00}$	$2^{-9.07}$
LILI-II	12	$2^{-13.22}$	$2^{-12.36}$

Table 6.1: The correlation ε_{h_d} of Boolean functions of some clock controlled generators, using weight three and weight four recursions.

(*iii*) **Position of the Windows:** The multiple $g(x)$ is expected to have a degree of approximately 2^{64}, i.e., $\tau_1 \approx 2^{63}$ and $\tau_2 \approx 2^{64}$, and hence $\tau_2 - \tau_1 \approx 2^{63}$. The output sequence from the clock-control part denoted by c_t in Figure 6.7 takes the values $c_t \in \{1, 2, 3, 4\}$ with equal probability, i.e., a geometric distribution. Thus in the case of LILI-II we know that $E(C) = 2.5$ and $\sigma_c = \sqrt{7.5}$. The center positions of the windows will be positioned approximately at $t - 2^{61.68}$ and $t + 2^{61.68}$, where t denotes the position of the center symbol in the recurrence.

(*iv*) **Determine the Size of the Windows:** As stated in previously we know that $E(C) = 2.5$ and $\sigma_c = \sqrt{7.5}$ for LILI-II. We will use a window size of four standard deviations, i.e., $L = 4\sqrt{7.5 \cdot n}$.

Using the expected positions of the windows for LILI-II from previous section the expected window sizes for a weight three relation are $L = 4\sqrt{7.5 \cdot 2^{61.68}} = 2^{34.29}$.

(*v*) **Estimate the Number of Bits We Need to Observe:** If we use the estimated numbers from the previous section and Equation (6.13) we get the following estimate on the number of bits we need to observe to distinguish LILI-II from a random source. For $w = 3$,

$$N \approx \frac{11}{4} \cdot \frac{2^{34.29 \cdot 2}}{2^{(-13.22) \cdot 2} \cdot 2^{(-4) \cdot 2}} \approx 2^{104.48}. \tag{6.20}$$

6.3.1 Simulations on a Scaled Down Version of LILI-II

To verify the correctness of the attack we performed the attack on a scaled down versions of LILI-II. In the scaled down version we kept the original clock control part unchanged, but used a weaker data generation part. Instead of the original LFSR_d we used the primitive trinomial,

$$x^{3660} + x^{1637} + 1, \tag{6.21}$$

as feedback polynomial.

We fix the center member of the feedback polynomial, and the center position for window one will be positioned at

$$\begin{aligned} &t - \tau_1/E(C) = t - \tfrac{3660-1637}{2.5} = t - 809, \text{ and at} \\ &t + \tfrac{\tau_2-\tau_1}{E(C)} = t - \tfrac{1637}{2.5} = t + 655 \end{aligned} \tag{6.22}$$

for window two. We use window sizes of four standard deviations, i.e., $L_1 = 4\sqrt{7.5 \cdot 809} = 312$ and $L_2 = 4\sqrt{7.5 \cdot 655} = 280$.

The Boolean function, h_d, was replaced with the 3-resilient 7-input plateaued function also used in [LZGB03],

$$h_d(x) = 1+x_1+x_2+x_3+x_4+x_5+x_6+x_1x_7+x_2(x_3+x_7)+x_1x_2(x_3+x_6+x_7). \tag{6.23}$$

The correlation of this Boolean function for a weight three relation is $\varepsilon_{h_d} = -2^{-4}$, i.e.,

$$\Pr\Big(h(\boldsymbol{s_t}) \oplus h(\boldsymbol{s_{t+\tau_1}}) \oplus h(\boldsymbol{s_{t+\tau_2}}) \ = 0 \mid \boldsymbol{s_t} \oplus \boldsymbol{s_{t+\tau_1}} \oplus \boldsymbol{s_{t+\tau_2}} = \mathbf{0},\ \forall t\Big) = \frac{1}{2} - 2^{-4}.$$

For a weight three relation the probability that all bits are included in the keystream is $p_{dec} = 2^{-4}$. The number of bits we need to observe can now be estimated as

$$N \approx \frac{11}{4} \cdot \frac{312 \cdot 280}{(2^{-4})^2(2^{-4})^2} = 2^{34.46}. \tag{6.24}$$

We used $N = 2^{36.8054}$ in a simulated attack where we tried to verify the theoretical bias. The number of combinations fulfilling the recurrence equation in the simulation was

$$\mathcal{D}^X = 2^{52.2201} - 2^{29.2829}, \tag{6.25}$$

where $2^{52.2201}$ is half of the total number of collected samples. This gives a deviation of $2^{29.2829}$ from the expected value of a random sequence, and hence a simulated bias of $\varepsilon_{tot} = 2^{-23.9372}$. This can be compared with the theoretically derived bias which is $\varepsilon_{tot} = 2^{-24.4147}$. The standard deviation can be calculated as $\sigma = \sqrt{N\frac{L_1L_2}{4}} = 2^{25.6101}$.

We also implemented the attack and performed 100 attacks on both keystream and random data, the results of the attacks are summarized in Table 6.2. The results matched the theory well.

6.3.2 Results

In this section we summarize the results of the attack applied on LILI-II, we also show the results for the attack if performed on LILI-128. Observe that

N	$1-\alpha$	$1-\beta$
2^{33}	0.88	0.97
2^{34}	0.99	0.98

Table 6.2: Simulated confidence levels $1-\alpha$ and power levels $1-\beta$ of the scaled down version of LILI-II.

there exist many better attacks on LILI-128. These attacks all use the fact that one of the LFSRs only has degree 39, if this degree would be increased the attacks would become significantly less effective, the complexity of our attack would not be affected at all.

In Table 6.3 we list the sizes on the windows used to attack the generator and the total number of keystream bits we need to observe to reliably distinguish the ciphers from a random source. The results in the table is calculated for a weight three recurrence relation.

Function	L_1	L_2	$\sharp$ bits needed
LILI-128	$2^{25.45}$	$2^{25.45}$	$2^{76.95}$
LILI-II	$2^{34.29}$	$2^{34.29}$	$2^{103.02}$

(6.26)

Table 6.3: The number of bits needed for the distinguisher for two members of the LILI family.

6.4 Summary

In this chapter we have described a distinguisher applicable to irregularly clocked filter generators. The attack has been applied on a member of the LILI family, namely LILI-II. The attack on LILI-II needed 2^{104} bits of keystream and a computational complexity of approximately 2^{104} operations to reliably distinguish the cipher from random data. This is the best known attack of this kind so far. We have also proposed two possibles improvements which might offer some advantages over the basic attack in some specific cases.

7

Distinguishing Attacks on the Pomaranch Family of Stream Ciphers

Pomaranch is a family of stream ciphers in the eSTREAM stream cipher project, it was designed by Jansen, Helleseth and Kholosha. The Pomaranch family consists of several versions and variants that were submitted to Profile 2 of the eSTREAM process, i.e., it is mainly designed to be fast and compact in hardware. The first two versions have been cryptanalyzed in [CGJ06, Kha05, HJ06]. For each new version, the cipher has been changed such that the attacks on the previous versions would not be successful. The third versions of the Pomaranch ciphers were included as phase 3 candidates in eSTREAM.

In this chapter we present a framework of general distinguishing attacks. The statistical distinguisher, which can be applied to all Pomaranch-like ciphers having one or several types of jump registers and either linear or nonlinear filter function. We improve the computational complexity of all known distinguishers on Version 1 and Version 2. Our attack is also applied to Pomaranch Version 3 and it is shown that the attack will succeed on the 80-bit variant with computational complexity $2^{71.10}$, significantly less than

exhaustive key search. For the 128-bit variant, the attack will have computational complexity 2^{126}, almost that of exhaustive key search. The expressions for the complexity of the attack given in this chapter can be interpreted as design criteria for subsequent versions of Pomaranch.

The chapter is based on parts of [EHJ07b] and the outline of the chapter is as follows. Section 7.1 will describe the Pomaranch stream ciphers and Section 7.2 will briefly describe the previous attack on Pomaranch. In Section 7.3 we will describe some properties of Pomaranch ciphers that we will later use in our attack. In Section 7.4 we describe our distinguisher for different design principles used in Pomaranch ciphers. We then demonstrate our distinguisher on all existing version and variants of proposed Pomaranch ciphers in Section 7.5. Section 7.6 summarizes the chapter.

7.1 Description of Pomaranch

In this section, we will give a brief overview of the design of Pomaranch. There are three versions of Pomaranch. First the overall design idea is presented and then we give the specific parameters for the different versions. The attacks described in this chapter are independent of the initialization procedure and thus, only the keystream generation will be described here. For more details we refer to the respective design documents, see [JHK05, JHK06a, JHK06b].

Pomaranch is a synchronous stream cipher designed primarily for being efficient in hardware. The general structure resembles that of the nonlinear combination generator. The design of Pomaranch is illustrated in Figure 7.1. Pomaranch is based on a cascade of n *jump registers* (JR). A jump register can be seen as an irregularly clocked linear finite state machine (LFSM). The clocking of a register can be done in one of two ways, either it jumps c_0 steps or it jumps c_1 steps. The clocking is decided by a binary jump control sequence, denoted $j_t^{(i)}$ for register i at time t,

$$j_t^{(i)} = \begin{cases} 0, & \text{JR}_i \text{ is clocked } c_0 \text{ times} \\ 1, & \text{JR}_i \text{ is clocked } c_1 \text{ times} \end{cases} . \tag{7.1}$$

The jump registers are implemented using two different kinds of delay shift cells, S-cells and F-cells. The S-cell is a normal D-element and the F-cell is a D-element where the output is fed back and XORed with the input. Half of the cells are implemented as S-cells and half are implemented as F-cells. When the register is clocked it will jump c_0 steps. When the jump control $j_t^{(i)}$ is one, all cells in JR_i are switched to the opposite mode, i.e., all S-cells become F-cells and vice versa, see Figure 7.2 for an explanation of how to construct such a cell. This switch of cells results in a jump through the state

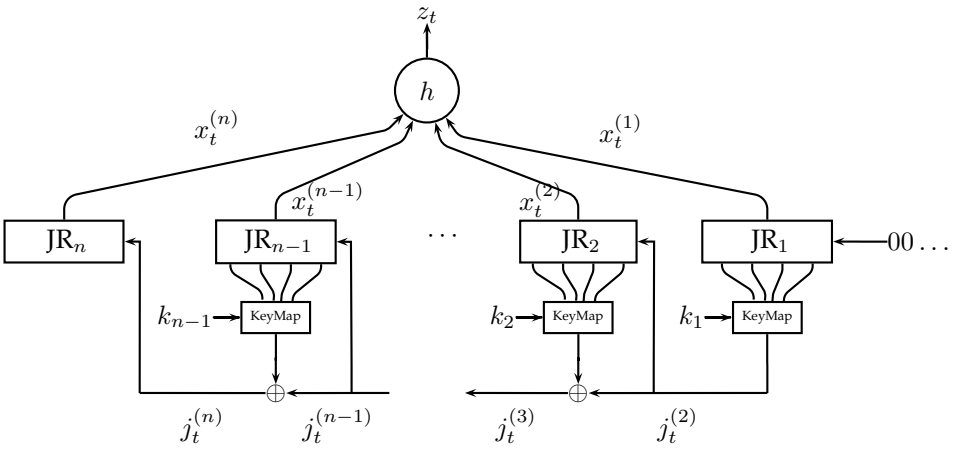


Figure 7.1: The general structure of the Pomaranch family of stream ciphers.

space corresponding to c_1. The jump index of a register is the number of c_0 clockings that is equivalent to one c_1 clocking.

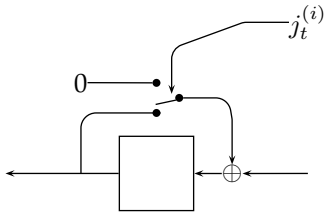


Figure 7.2: Jump register cell.

Notations. As seen in Figure 7.1 we denote jump register i by JR$_i$ and its period by $T^{(i)}$. The key is denoted by K and the subkey used for JR$_i$ is denoted k_i. The length of one registers is denoted by L, it is assumed that all registers are of the same length. The filter function is denoted by h (more specifically we will use the notation $h^{[1]}, h_{80}^{[2]}, h_{128}^{[2]}, h_{80}^{[3]}, h_{128}^{[3]}$ for the different filter functions, where the superscript denotes the version of Pomaranch of interest, and the subscript denotes the key length used in the version). Similarly, we will denote the number of registers used by $n^{[1]}, n_{80}^{[2]}, n_{128}^{[2]}, n_{80}^{[3]}, n_{128}^{[3]}$. The bit taken from the register i as input to the filter function at time t is de-

noted by $x_t^{(i)}$.

7.1.1 Pomaranch Version 1

The first version of Pomaranch was introduced in [JHK05]. In this version, 128 bit keys were used together with an IV in the range of 64 to 112 bits. The cipher is built upon $n^{[1]} = 9$ identical jump registers. Each register uses 14 memory cells together with a characteristic polynomial with the jump index 5945, i.e., $(c_0, c_1) = (1, 5945)$. From register i, nine bits are taken as input to a key dependent function, denoted KeyMap in Figure 7.1. The output of this function is XORed to the jump sequence, $j_t^{(i)}$, from register $i-1$, to produce the jump sequence, $j_t^{(i+1)}$, for the next register. The keystream bit z is given as the XOR of the bit in cell 13 from all the registers, i.e.,

$$z = h^{[1]}\big(x^{(1)}, \ldots, x^{(9)}\big) = x^{(1)} + x^{(2)} + \ldots + x^{(9)}. \tag{7.2}$$

7.1.2 Pomaranch Version 2

The second version of Pomaranch [JHK06a], comes in two variants, one 80-bit with $n_{80}^{[2]} = 6$, and one 128-bit key variant using $n_{128}^{[2]} = 9$ registers. The registers are still 14 memory cells long but to prevent the attacks on the first version, different tap positions are used as input to the KeyMap function in Figure 7.1. The characteristic polynomial is also changed and the new jump index is 13994. A new initialization procedure was introduced to prevent the attacks in [CGJ06, HKK05]. The keystream is still taken as the XOR of the bits in cell 13 of all registers and is given by

$$z = \begin{cases} h_{80}^{[2]}\big(x^{(1)}, \ldots, x^{(6)}\big) & = x^{(1)} + x^{(2)} + \ldots + x^{(6)} \\ h_{128}^{[2]}\big(x^{(1)}, \ldots, x^{(9)}\big) & = x^{(1)} + x^{(2)} + \ldots + x^{(9)} \end{cases}. \tag{7.3}$$

7.1.3 Pomaranch Version 3

As for the second version, there are two variants of Pomaranch Version 3 [JHK06b], one 80-bit and one 128-bit key variant. The number of registers used is still the same as in Pomaranch Version 2, i.e., $n_{80}^{[3]} = 6$ respectively $n_{128}^{[3]} = 9$. In the case of Pomaranch Version 3, two different jump registers are used, the first using jump index 84074 which is referred to as type I and the second using jump index 27044, referred to as type II. The type I registers are used for odd numbered sections in Figure 7.1, and type II for the even numbered sections. Both registers are built on 18 memory cells, and have primitive characteristic polynomials, i.e., when clocked only with zeros or

only with ones they have a period of $2^{18} - 1$. From each register one bit is taken from cell 17 and is fed into h. The filter functions used are

$$z = \begin{cases} h_{80}^{[3]}\big(x^{(1)}, \ldots, x^{(6)}\big) & = G\big(x^{(1)}, \ldots, x^{(5)}\big) + x^{(6)} \\ h_{128}^{[3]}\big(x^{(1)}, \ldots, x^{(9)}\big) & = x^{(1)} + x^{(2)} + \ldots + x^{(9)} \end{cases}, \tag{7.4}$$

where

$$\begin{aligned} G(x^{(1)}, \ldots, x^{(5)}) \quad = \quad & x^{(1)} + x^{(2)} + x^{(5)} + x^{(1)}x^{(3)} + x^{(2)}x^{(4)} + x^{(1)}x^{(3)}x^{(4)} + \\ & + x^{(2)}x^{(3)}x^{(4)} + x^{(3)}x^{(4)}x^{(5)}, \end{aligned} \tag{7.5}$$

is a 1-resilient Boolean function. The keystream length per IV/key pair is limited to 2^{64} bits in Version 3.

7.2 Previous Attacks on the Pomaranch Stream Ciphers

The first attack on Pomaranch by Cid, Gilbert and Johansson [CGJ06] was an attack on the initialization procedure. Soon after, an attack on the keystream generation was presented by Khazaei [Kha05]. This attack considered the best linear approximation of bits distance $L + 1$ apart. Register JR_1 was exhaustively searched as this register is always fed with the all zero jump control sequence and the linear approximation is not valid for this register. When the state of JR_1 was known, JR_2 and 16 bits of the key was guessed. This was iterated until the full key was recovered. A distinguisher was not explicitly mentioned by Khazaei [Kha05] but it is very easy to see that if the attack is stopped after JR_1 is recovered, it would be equivalent to a distinguishing attack. This distinguisher needs $2^{72.8}$ keystream bits and computational complexity $2^{86.8}$.

Pomaranch Version 2 was designed to resist the attacks in [CGJ06,Kha05]. However, it was still possible to find biased linear approximations by looking at keystream bits further apart. This was done by Hell and Johansson [HJ06] and the new approximation made it possible to mount distinguishing and key recovery attacks on both the 80-bit and the 128-bit variants. The distinguisher for the 80-bit variant needs $2^{48.39}$ keystream bits and a computational complexity of about $2^{62.39}$. For the 128-bit variant the complexities are about $2^{77.33}$, and $2^{91.33}$ respectively.

7.3 Properties Used in the Attack

In this section we will present general distinguishers for Pomaranch-like ciphers, in particular we will study the constructions that have been proposed

in Pomaranch Version 1-3. We will give general results for these cipher families that can be used as design criteria for future Pomaranch ciphers.

7.3.1 The Period of Registers

The first jump register, denoted JR_1 in Figure 7.1, is during keystream generation mode fed by the jump sequence only containing zeros. Hence JR_1 is a LFSM. The period of the register is denoted by T_1, hence $x_t^{(1)} = x_{t+T_1}^{(1)}$ with $T_1 = 2^L - 1$.

From JR_1 a jump control sequence is calculated which controls the jumping of JR_1. Assume that after T_1 clocks of JR_1, register JR_1 has jumped C steps. Then after T_1^2 clocks JR_2 has jumped CT_1 steps, a multiple of T_1 and is thus back to its initial state. If primitive characteristic polynomials are used for the registers, it can be shown that the period for register JR_i is

$$T_i = T_1^i, \tag{7.6}$$

and hence

$$x_t^{(i)} = x_{t+T_1^i}^{(i)}. \tag{7.7}$$

Consequently, at time t and $t + T_1^p$, the filter function h has p inputs with exactly the same value, namely the contribution from registers $\text{JR}_1, \ldots, JR_p$. This observation will be used in our attack.

7.3.2 The Filter Function

The filter 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

$$z_t = h\big(x_t^{(1)}, \ldots, x_t^{(n)}\big). \tag{7.8}$$

Using the results from Section 7.3.1 and taking our samples as $z_t \oplus z_{t+T_1^p}$ we can write the expression for the samples as

$$z_t \oplus z_{t+T_1^p} = h\big(x_t^{(1)}, \ldots, x_t^{(n)}\big) \oplus h\big(x_{t+T_1^p}^{(1)}, \ldots, x_{t+T_1^p}^{(n)}\big). \tag{7.9}$$

- **Linear Filter Function**. When the filter function h is linear, i.e.,

$$h(x^{(1)}, \ldots, x^{(n)}) = \bigoplus_{i=1}^{n} x^{(i)}, \tag{7.10}$$

and our samples are taken as $z_t \oplus z_{t+T_1^p}$ we know from Section 7.3.1 that p inputs to the filter function are the same at time t and at time

$t + T_1^p$, hence we can rewrite (7.9) as

$$z_t \oplus z_{t+T_1^p} = \bigoplus_{i=p+1}^{n} x_t^{(i)} \oplus x_{t+T_1^p}^{(i)}. \tag{7.11}$$

- **Nonlinear Filter Function**. When the filter function h is nonlinear, $x_t^{(i)}$ and $x_{t+T_1^p}^{(i)}$ will not cancel out in the keystream with probability one, as in the case of a linear filter function. But, the input to h at time t and $t+T_1^p$ has p inputs $x^{(1)}, \ldots, x^{(p)}$ with exactly the same value. This might lead to a biased distribution,

$$\begin{aligned} &\Pr\left(h\big(x_t^{(1)}, \ldots, x_t^{(n)}\big) \oplus h\big(x_{t+T_1^p}^{(1)}, \ldots, x_{t+T_1^p}^{(n)}\big) = 0 | x_t^{(i)} = x_{t+T_1^p}^{(i)}, 1 \le i \le p\right) \\ &= \Pr\left(z_t \oplus z_{t+T_1^p} = 0 \mid x_t^{(i)} = x_{t+T_1^p}^{(i)},\ 1 \le i \le p\right) = \tfrac{1}{2} + \epsilon, \end{aligned} \tag{7.12}$$

  where ϵ denotes the bias and $|\epsilon| \le 1$.

7.3.3 Linear Approximations of jump registers

In our attack, we need to find a linear approximation for the output bits of the jump registers that is biased, i.e., that holds with probability different from one half. We assume that all states of the register are equally probable, except for the all zero state which has probability 0. Further, it is assumed that all jump control sequences have the same probability. Finding the best linear approximation can then be done by exhaustive search. Under certain circumstances much faster approaches can be used, see [HJ06]. We search for a set $\mathcal{T}$ of size w such that

$$\Pr\big(\bigoplus_{i \in \mathcal{T}} x_{t+i} = 0\big) = \frac{1}{2} + \varepsilon, \quad |\varepsilon| \le 0.5, \tag{7.13}$$

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

It is assumed that jump register JR_1 will always have the all zero jump control sequence. Hence, the linear approximation will never apply for this register.

In the case when not all registers use the same kind of jump registers, 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 all types of registers at the same time. This is much harder to find than a single approximation for one register. This is the case in Pomaranch Version 3.

7.4 Attacking Different Versions of Pomaranch

A Pomaranch stream cipher can be designed using one or several types of jump registers. It can also use a linear or a nonlinear Boolean filter function. The attack used on all versions is in essence the same. We start by finding a linear approximation for one or several jump registers. We then calculate the cipher distribution $\mathcal{D}_{\mathcal{C}}$. We make an estimate on the amount of keystream, N, needed in the attack using the Squared Euclidean Imbalance. Samples are then collected from the keystream as

$$Q_t = \bigoplus_{i \in \mathcal{T}} z_{t+i} \oplus \bigoplus_{i \in \mathcal{T}} z_{t+i+T_1^p}. \tag{7.14}$$

Finally, we perform a hypothesis test according to Section 3.1.3.2 using the log-likelihood ratio. The attack is summarized in Figure 7.3.

1. Find a linear approximation for the jump registers, i.e., find $\mathcal{T}$ such that
   $\Pr(\bigoplus_{i \in \mathcal{T}} x_{t+i} = 0) = \frac{1}{2} + \varepsilon, \quad |\varepsilon| \leq 0.5$
2. Calculate the distribution $\mathcal{D}_{\mathcal{C}}$.
3. Calculate the number of samples, N, we need to observe.
4. **for** $t = 0 \ldots N$
   $Q_t = \bigoplus_{i \in \mathcal{T}} z_{t+i} \oplus \bigoplus_{i \in \mathcal{T}} z_{t+i+T_1^p}$
   end for
5. Calculate $LLR = \sum_{t=0}^{N} \log_2 \left(\frac{\mathcal{D}_{\mathcal{C}}(Q_t)}{1/2} \right)$.
6. **if** $(LLR \geq 0)$
   output **cipher**.
   else
   output **random**.

Figure 7.3: Summary of the new distinguishing attack on the Pomaranch family of stream ciphers.

We will now take a closer look at 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 criteria for Pomaranch-like stream ciphers.

7.4.1 One Type of Registers with Linear Filter Function

In this family of Pomaranch stream ciphers we assume that all jump register sections use the same type of register and that the filter function h is linear, i.e., $h(x^{(1)}, \ldots, x^{(n)}) = \sum_{i=1}^{n} x^{(i)}$. 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 7.3.3, of weight w of the register used. We consider samples at t and $t + T_1^p$ such that p positions into h are the same according to Section 7.3.1. Our samples will be taken as

$$\begin{aligned} \bigoplus_{i\in\mathcal{T}} z_{t+i} \oplus \bigoplus_{i\in\mathcal{T}} z_{t+i+T_1^p} &= \bigoplus_{j=1}^{n} \bigoplus_{i\in\mathcal{T}} \left(x_{t+i}^{(j)} \oplus x_{t+i+T_1^p}^{(j)}\right) \\ &= \bigoplus_{j=p+1}^{n} \bigoplus_{i\in\mathcal{T}} \left(x_{t+i}^{(j)} \oplus x_{t+i+T_1^p}^{(j)}\right). \end{aligned} \tag{7.15}$$

The distribution of these samples are denoted by $\mathcal{D}_{\mathcal{C}}$, i.e.,

$$\mathcal{D}_{\mathcal{C}}(x) = \Pr\Big(\bigoplus_{i\in\mathcal{T}} z_{t+i} \oplus \bigoplus_{i\in\mathcal{T}} z_{t+i+T_1^p} = x\Big),\ x \in \{0,1\}. \tag{7.16}$$

If we assume that this distribution is skew we can write the distribution as

$$\begin{cases} \mathcal{D}_{\mathcal{C}}(0) = \frac{1}{2} + \varepsilon_{tot} \\ \mathcal{D}_{\mathcal{C}}(1) = \frac{1}{2} - \varepsilon_{tot} \end{cases}. \tag{7.17}$$

The bias of $\bigoplus_{i\in\mathcal{T}} x_{t+i}^{(i)}$ is ε and we have $2(n-p)$ such relations. Using the piling-up lemma introduced in Equation 4.7, the total bias of the samples is given by

$$\varepsilon_{tot} = 2^{2(n-p)-1}\varepsilon^{2(n-p)}. \tag{7.18}$$

We once again use the Squared Euclidean Imbalance in (3.33) from Section 3.1.3.2. This equation states that we need $11/\Delta(\mathcal{D}_{\mathcal{C}})$ samples to distinguish the distribution from the random distribution. We can express the SEI as

$$\Delta(\mathcal{D}_{\mathcal{C}}) = 2^{2(n-p)}\varepsilon^{2(n-p)}. \tag{7.19}$$

We can now state the following theorem.

Theorem 7.1: The computational complexity and the number N of keystream bits needed to reliably distinguish the Pomaranch family of stream

ciphers using a linear filter function and n jump registers of the same type are both bounded by

$$T_1^p + \frac{11}{(2\varepsilon)^{4(n-p)}}, \quad p > 0, \tag{7.20}$$

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

7.4.2 Different Registers with Linear Filter Function

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

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 JR$_{p+1}, \ldots, JR_n$. 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 (7.15). If we denote the bias for the approximation of register i by ε_i, then the total bias will be given as

$$\varepsilon_{tot} = 2^{n-p-1} \prod_{i=p+1}^{n} \varepsilon_i^2. \tag{7.21}$$

This gives us the Squared Euclidean Imbalance

$$\Delta(\mathcal{D}_\mathcal{C}) = 2^{2(n-p)} \prod_{i=p+1}^{n} \varepsilon_i^4. \tag{7.22}$$

We can now give the following theorem

Theorem 7.2: Assume that 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 filter function and n jump registers of different types are both bounded by

$$T_1^p + \frac{11}{2^{4(n-p)} \prod_{i=p+1}^{n} \varepsilon_i^4}, \quad p > 0, \tag{7.23}$$

where ε_i is the bias of jump register JR$_i$.

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 I and registers in even positions are of type II. In this case we only have to search for a linear approximation that is biased for type I and type II registers at the same time. The bias of $\bigoplus_{i\in\mathcal{T}} x_{t+i}^{(i)}$ is denoted $\varepsilon_{type\ I}$ and $\varepsilon_{type\ II}$, respectively, for the different registers. In total we have $2\lceil\frac{n-p}{2}\rceil$ type I relations and $2\lfloor\frac{n-p}{2}\rfloor$ type II relations when n is odd. Hence, the total bias of the samples is given by

$$\varepsilon_{tot} = 2^{n-p-1}\varepsilon_{type\ I}^{2\lceil\frac{n-p}{2}\rceil}\varepsilon_{type\ II}^{2\lfloor\frac{n-p}{2}\rfloor}. \tag{7.24}$$

If we apply Theorem 7.2 to the 128-bit variant of Pomaranch Version 3, the number of samples in the distinguisher is given by

$$N = T_1^p + \frac{11}{2^{4(n-p)}\varepsilon_{type\ I}^{4\lceil\frac{n-p}{2}\rceil}\varepsilon_{type\ II}^{4\lfloor\frac{n-p}{2}\rfloor}}. \tag{7.25}$$

7.4.3 Nonlinear Filter Function

Now we consider the case when the Boolean filter function is a nonlinear function. We only consider the case when the filter function h can be written in the form

$$h(x^{(1)},\ldots,x^{(n)}) = G(x^{(1)},\ldots,x^{(n-1)}) \oplus x^{(n)}. \tag{7.26}$$

The attack can easily be extended to filter 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 in this form, we only consider this special case here.

Attacks on this family use a biased linear approximation of JR_n, see Section 7.3.3, 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 7.3.2.

Let ϵ denote the bias of $G\big(x_t^{(1)},\ldots,x_t^{(n-1)}\big) \oplus G\big(x_{t+T_1^p}^{(1)},\ldots,x_{t+T_1^p}^{(n-1)}\big)$, and ε the bias of our linear approximation for JR_n, $\bigoplus_{i\in\mathcal{T}} x_{t+i}^{(i)}$. The samples are taken as

$$\begin{aligned}\bigoplus_{i\in\mathcal{T}} z_{t+i} \oplus \bigoplus_{i\in\mathcal{T}} z_{t+i+T_1^p} =& \bigoplus_{i\in\mathcal{T}} x_{t+i}^{(n)} \oplus \bigoplus_{i\in\mathcal{T}} x_{t+i+T_1^p}^{(n)} \\ \oplus \bigoplus_{i\in\mathcal{T}} G\big(x_{t+i}^{(1)},\ldots,x_{t+i}^{(n-1)}\big) &\oplus G\big(x_{t+i+T_1^p}^{(1)},\ldots,x_{t+i+T_1^p}^{(n-1)}\big)\end{aligned} \tag{7.27}$$

and the bias of the samples is given by

$$\varepsilon_{tot} = 2^{w+1}\varepsilon^2\epsilon^w. \tag{7.28}$$

This relation tells us that we need to keep the weight of the linear approximation of JR_n as low as possible, i.e., there is a trade off between the bias ε of the approximation and the number of terms w in the relation.

Theorem 7.3: The computational complexity and the number N of keystream bits needed to reliably distinguish the Pomaranch family of stream ciphers using a filter function of the form (7.26) are both bounded by

$$T_1^p + \frac{1}{(2^{w+1}\varepsilon^2\epsilon^w)^2}, \tag{7.29}$$

where ε is the bias of the approximation of weight w of register JR_n and ϵ is the bias of $G\big(x_t^{(1)}, \ldots, x_t^{(n-1)}\big) \oplus G\big(x_{t+T_1^p}^{(1)}, \ldots, x_{t+T_1^p}^{(n-1)}\big)$.

Note that in this presentation it does not matter if all registers are of the same type or if they are of different types. As only register JR_n is completely linear in the output function H, we only need to have an approximation of this register.

7.5 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 attack proposed in this chapter, we show that we can find distinguishers with better complexity than previously known for *all* five ciphers.

7.5.1 Pomaranch Version 1

In Pomaranch Version 1 all registers are the same, so the attack will be according to Section 7.4.1. The best known linear approximation for this register, as given in [Kha05], is

$$\varepsilon = \big|\Pr(x_t \oplus x_{t+8} \oplus x_{t+14} = 0) - 1\big| = 2^{-5.286}. \tag{7.30}$$

Using Theorem 7.1 for different values of p we get Table 7.1. We see that the best attack is achieved when $p = 5$. The computational complexity and the amount of keystream needed is then $2^{72.35}$.

p	1	2	3	4	5	6	7
$N^{[1]}$	$2^{140.61}$	$2^{123.47}$	$2^{106.32}$	$2^{89.18}$	$2^{72.35}$	$2^{83.99}$	$2^{97.99}$

Table 7.1: Number of samples needed to distinguish Pomaranch Version 2 from a random source according to Theorem 7.1.

7.5.2 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 7.4.1. The best bias of a linear approximation for the registers used was found in [HJ06] and is given by

$$\varepsilon = \left|\Pr(x_t \oplus x_{t+2} \oplus x_{t+6} \oplus x_{t+18} = 0) - 1\right| = 2^{-4.788}. \tag{7.31}$$

Using Theorem 7.1 for different values of p gives Table 7.2. For the 80-bit variant the computational complexity and the number of samples is $2^{56.00}$ and for the 128-bit variant it is $2^{80.07}$.

p	1	2	3	4	5	6
$N_{80}^{[2]}$	$2^{99.22}$	$2^{80.07}$	$2^{60.91}$	$2^{56.00}$	$2^{70.00}$	$2^{84.00}$
$N_{128}^{[2]}$	$2^{156.68}$	$2^{137.52}$	$2^{118.37}$	$2^{99.22}$	$2^{80.07}$	$2^{84.00}$

Table 7.2: Number of samples needed to distinguish Pomaranch Version 2 from a random source according to Theorem 7.1.

7.5.3 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 non-linear filter function, the attack will hence follow the procedure described in Section 7.4.3.

  We started by estimating the bias of

$$G\big(x_t^{(1)}, \ldots, x_t^{(5)}\big) \oplus G\big(x_{t+T_1^p}^{(1)}, \ldots, x_{t+T_1^p}^{(5)}\big). \tag{7.32}$$

The results for different p are summarized in Table 7.3. The keystream per IV/key pair of Pomaranch Version 3 is limited to 2^{64}. Because of this we limited p to $p \in \{1, 2, 3\}$, otherwise $T_1^p > 2^{64}$. We looked for a linear relation of JR_6 that, together with a value of $p \in \{1, 2, 3\}$, minimizes the amount of keystream needed as given by Theorem 7.3. The best approximation found was

$$\Pr\big(x_t^{(6)} \oplus x_{t+5}^{(6)} \oplus x_{t+7}^{(6)} \oplus x_{t+9}^{(6)} \oplus x_{t+12}^{(6)} \oplus x_{t+18}^{(6)} = 0\big) = \frac{1}{2}(1 - 2^{-8.774}), \quad (7.33)$$

using $p = 3$. The total bias of our samples using this approximation is

$$\varepsilon_{tot} = (2^{6+1}2^{-9.774})^2 \cdot (2^{-4})^6 = 2^{-35.548}, \quad (7.34)$$

according to (7.28). The samples used in the attack are taken according to

$$\bigoplus_{i \in \mathcal{T}} z_{t+i} \oplus \bigoplus_{i \in \mathcal{T}} z_{t+i+T_1^3}, \quad (7.35)$$

where $\mathcal{T} = \{0, 5, 7, 9, 12, 18\}$. According to Theorem 7.3, the amount of keystream needed is $2^{54} + 2^{74.56} = 2^{74.56}$. 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 as all samples will have this bias regardless of the key and IV used. We only need to consider 2^{64} keystream bits from $\lceil 2^{10.56} \rceil = 1510$ different key/IV pairs.

p	1	2	3	4	5
ϵ	0	2^{-4}	2^{-3}	2^{-2}	1

Table 7.3: The bias of $G\big(x_t^{(1)}, \ldots, x_t^{(5)}\big) \oplus G\big(x_{t+T_1^p}^{(1)}, \ldots, x_{t+T_1^p}^{(5)}\big)$ in the 80 bit variant of Pomaranch Version 3 for different values of p.

- **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

$$x_t \oplus x_{t+1} \oplus x_{t+2} \oplus x_{t+5} \oplus x_{t+7} \oplus x_{t+11} \oplus x_{t+12} \oplus x_{t+15} \oplus x_{t+21}, \quad (7.36)$$

which has the same bias for both types of registers, namely

$$\varepsilon_{even} = \varepsilon_{odd} = 2^{-10.934}. \quad (7.37)$$

Using (7.25) for different values of p we get Table 7.4. 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 as the distance between the bits in each sample has to be $2^{126.00}$. As the frame length is limited to 2^{64} it will not be possible to get any biased samples at all with $p = 7$.

p	1	2	3	4	5	6	7	8
$N_{128}^{(3)}$	$2^{353.35}$	$2^{309.61}$	$2^{265.88}$	$2^{222.14}$	$2^{178.40}$	$2^{134.67}$	$2^{126.00}$	$2^{144.00}$

Table 7.4: Number of samples needed to distinguish the 128-bit variant of Pomaranch Version 3 according to (7.25).

7.6 Summary

In this chapter we have presented a framework for statistical distinguishers on the Pomaranch family of stream ciphers. We have demonstrated how to use the framework on all proposed Pomaranch ciphers. The results can also be seen as a design criteria for future design proposals of Pomaranch-like ciphers. The amount of keystream and the computational complexity needed in our attack on the existing versions and variants of Pomaranch ciphers are summarized in Table 7.5.

Attack Complexities		Keystream	Comp. Complexity
Pomaranch v1	128 bit	2^{72}	2^{72}
Pomaranch v2	80 bit	2^{56}	2^{56}
	128 bit	2^{80}	2^{80}
Pomaranch v3	80 bit	2^{75}	2^{75}
	128 bit	$* \ 2^{126}$	2^{126}

* Without frame length restriction

Table 7.5: Computational complexity and keystream needed for the proposed distinguishers for all versions and variants of Pomaranch.

8

Find the Dragon

Dragon is a word oriented stream cipher submitted as a Profile 1 candidate to the eSTREAM project, i.e., it is designed to be very fast in software. Dragon is one of eight primitives in Profile 1 that have been selected for phase 3 of the project. The design is based on a linear block (a counter) and a *nonlinear feedback shift register* (NLFSR) with a very large internal state of 1088 bits. The state is updated by a nonlinear function. This function is also used as a filter function to produce the keystream.

In this chapter we show how statistical weaknesses in the F function can be used to create a distinguisher for Dragon. However, it is important to note that our attack is only successful if we remove the constraint that the cipher has to be resynchronized after 2^{64} keystream words. Our distinguishing attack requires around 2^{155} words of keystream from Dragon, it has time complexity 2^{187} and needs 2^{32} of memory. An alternative method is also presented that only requires time complexity 2^{155} but needs 2^{96} of memory. The attack shows that Dragon does not provide full security when a key of size 256 bits is used.

The chapter is based on [EM05] and the outline of the chapter is the following. In Section 8.1 a short description of the stream cipher Dragon is given. Afterward, in Section 8.2, we derive linear relations, and build our distinguisher in Section 8.3. Then in Section 8.4 we describe how to calculate the noise distribution. In Section 8.5 we summarize different attack scenarios on Dragon, and finally, in Section 8.6 we present our results, make conclusions and discuss possible ways to overcome the attack.

8.1 A Description of Dragon

Dragon is a stream cipher constructed from a large nonlinear feedback shift register, an update function, F, and a memory denoted as M. It is a word oriented cipher that operates on 32 bit words, the NLFSR is 1024 bits long, i.e., 32 words long. The words in the internal state are denoted as $B_i,\ 0 \leq i \leq 31$. The memory M (counter) contains 64 bits, which is used as a linear part with the period of 2^{64}. The cipher handles two key sizes, namely 128 bit keys and 256 bit keys. In our attack we disregard the initialization procedure and just assume that the initial state of the NLFSR is randomly selected.

In each round the F function takes six words as input and produces six words of output, as shown in Figure 8.1. In the figure $\oplus$ denotes 32 bit parallel XOR, and and $\boxplus$ arithmetical addition modulo 2^{32}. These six

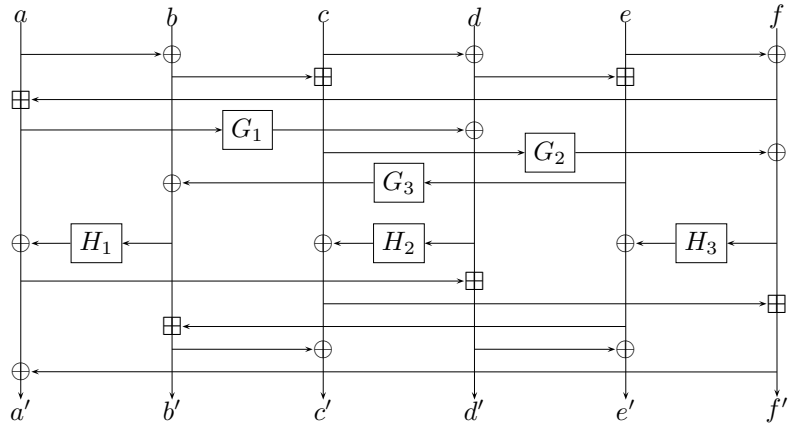


Figure 8.1: F-function.

words, denoted as a, b, c, d, e, f, are formed from words of the NLFSR and the memory register M, as explained in (8.1), where $M = (M^{(L)}, M^{(R)})$,

$$\begin{array}{lll} a = B_0, & b = B_9, & c = B_{16}, \\ d = B_{19}, & e = B_{30} \oplus M_L, & f = B_{31} \oplus M^{(R)}. \end{array} \tag{8.1}$$

The F function uses six $\mathbb{Z}_{2^{32}} \rightarrow \mathbb{Z}_{2^{32}}$ S-boxes $G_1,\ G_2,\ G_3,\ H_1,\ H_2$ and H_3. The purpose of them is to provide high algebraic immunity and nonlinearity. These S-boxes are constructed from two other fixed $\mathbb{Z}_{2^8} \rightarrow \mathbb{Z}_{2^{32}}$

S-boxes, S_1 and S_2, as shown below,

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

where 32 bits of input, $\boldsymbol{x}$, is represented by its four bytes as $\boldsymbol{x} = (x_0, x_1, x_2, x_3)$.

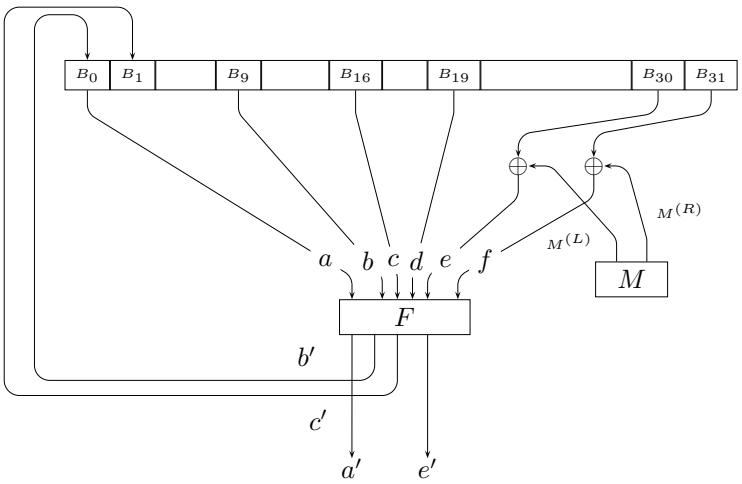


Figure 8.2: A overview of the Dragon keystream generator.

The exact specification of the S-boxes can be found in [CHM+05]. The output of the function F is denoted as (a', b', c', d', e', f'), from which the two words a' and e' forms 64 bits of keystream as $z = (a', e')$. Two other output words from the filter function are used to update the NLFSR as

$$
\begin{cases}
B_0 = b',\\
B_1 = c',\\
B_i = B_{i-2},\ 2 \le i \le 31.
\end{cases}
\tag{8.2}
$$

A short description of the keystream generation function is summarized in Figure 8.3.

Input= $\{B_0, \ldots, B_{31}, M\}$
1. $(M^{(L)}, M^{(R)}) = M$.
2. $a = B_0,\ b = B_9,\ c = B_{16},\ d = B_{19},\ e = B_{30} \oplus M^{(L)},\ f = B_{31} \oplus M^{(R)}$.
3. $(a', b', c', d', e', f') = F(a, b, c, d, e, f)$.
4. $B_0 = b',\ B_1 = c'$
5. $B_i = B_{i-2},\ 2 \le i \le 31$.
6. $M++$
7. $z = (a', e')$

Output= $\{k, B_0, \ldots, B_{31}, M\}$

Figure 8.3: Dragon's Keystream Generation Function.

8.2 Linear Approximation of the Function F

We will start the description of our attack by giving a description of how to linearly approximate the non-linear function F.

Recall that at time t the input to the function F is a vector of six words

$$(a, b, c, d, e, f) = (B_0, B_9, B_{16}, B_{19}, B_{30} \oplus M^{(L)}, B_{31} \oplus M^{(R)}). \tag{8.3}$$

The output of F are the words (a', b', c', d', e', f'). To simplify further expressions, let us introduce the new variables

$$\begin{cases} b'' &= b \oplus a = B_9 \oplus B_0, \\ c'' &= c \boxplus (a \oplus b) = B_{16} \boxplus (B_9 \oplus B_0), \\ d'' &= d \oplus c = B_{19} \oplus B_{16}, \\ f'' &= f \oplus e = B_{30} \oplus B_{31} \oplus M^{(L)} \oplus M^{(R)}. \end{cases} \tag{8.4}$$

If the words denoted as B are independent, then these new variables will also be independent (as B_{19} is independent of B_{16} and random, then d'' is independent and random as well; similarly, independence of B_{16} lead to the independence of c'', etc.).

The output from F can be expressed via $(a, b'', c'', d'', e, f'')$ as follows,

$$\begin{aligned} a' &= (a \boxplus f'') \oplus H_1(b'' \oplus G_3(e \boxplus d'')) \oplus \\ &\oplus \Big((f'' \oplus G_2(c'')) \boxplus \Big(c'' \oplus H_2(d'' \oplus G_1(a \boxplus f'')) \Big) \Big), \\ e' &= (e \boxplus d'') \oplus H_3(f'' \oplus G_2(c'')) \oplus \\ &\oplus \Big((d'' \oplus G_1(a \boxplus f'')) \boxplus \Big((a \boxplus f'') \oplus H_1(b'' \oplus G_3(e \boxplus d'')) \Big) \Big). \end{aligned} \tag{8.5}$$

Let us now analyze the expression for a'. The variable b'' appears only once (in the input of H_1), which means that this input is independent from other terms of the expression, i.e., the term $H_1(.)$ can be substituted by $H_1(r_1)$, where r_1 is some independent and uniformly distributed random variable. Then, the same will happen with the input for H_2.

We would like to approximate the expression for a' as

$$a' = a \oplus N^{(a)}, \tag{8.6}$$

where $N^{(a)}$ is some nonuniformly distributed noise variable. If we XOR both sides with a and then substitute a' with the expression from (8.5), we derive

$$N^{(a)} = a \oplus (a \boxplus f'') \oplus H_1(r_1) \oplus \Big((f'' \oplus G_2(c'')) \boxplus (c'' \oplus H_2(r_2)) \Big). \tag{8.7}$$

Despite the fact that G and H are $\mathbb{Z}_{2^{32}} \rightarrow \mathbb{Z}_{2^{32}}$ functions, they are not likely to be one-to-one mappings, considering the way the S-boxes are used as $\mathbb{Z}_{2^8} \rightarrow \mathbb{Z}_{2^{32}}$ functions [1] . This means that even if the input to a G or a H function is completely random, then the output will still be biased. Moreover, the output from the expressions ($x \oplus G_i(x)$ and similarly $x \oplus H_i(x)$) are also biased, as x in these expressions plays a role of an approximation of the G_i and the H_i functions. These observations mean that the noise variable $N^{(a)}$, is also biased if the input variables are independent and uniformly distributed.

By a similar observation, the expression for e' can also be approximated as follows.

$$e' = e \oplus N^{(e)}, \tag{8.8}$$

where $N^{(e)}$ is the noise variable. The expression for $N^{(e)}$ can similarly be derived as

$$N^{(e)} = e \oplus (e \boxplus d'') \oplus H_3(r_3) \oplus \Big((d'' \oplus G_1(a'')) \boxplus (a'' \oplus H_1(r_4) \Big), \tag{8.9}$$

where $a'' = a \boxplus f''$ is a new random variable, which is also independent as it has f'' as its component, and f'' does not appear anywhere else in the expression (8.9). The two new variables r_3 and r_4 are also independent and uniformly distributed random variables by similar reasons.

8.3 Constructing a Distinguisher

The key observation for our distinguisher, is that one of the input words to the filter function F, at time t is partially repeated as input to F at time

[1] The cipher Turing uses similar $\mathbb{Z}_{2^{32}} \rightarrow \mathbb{Z}_{2^{32}}$ functions based on $\mathbb{Z}_{2^8} \rightarrow \mathbb{Z}_{2^{32}}$ S-boxes, which can be regarded as a source of weakness. However, no attack has been found on Turing so far.

$t + 14$, i.e.,

$$e_{t+14} = a_t \oplus M^{(L)}_{t+14}. \tag{8.10}$$

Let $x_t \in \mathcal{X}$ be a sample at time t where $\mathcal{X} = \{0, \ldots, 2^{32} - 1\}$, then we will construct our samples as the sum of two keystream words

$$\begin{aligned} x_t = &\quad a'_t \oplus e'_{t+14} = \\ = &\quad (a_t \oplus N^{(a)}_t) \oplus (a_t \oplus M^{(L)}_{t+14} \oplus N^{(e)}_{t+14}) = \\ = &\quad \underbrace{N^{(a)}_t \oplus N^{(e)}_{t+14}}_{N^{(tot)}_t} \oplus M^{(L)}_{t+14} \end{aligned} . \tag{8.11}$$

Let $\mathcal{D}_\mathcal{C}$ denote the distribution of the noise variable $N^{(tot)}$, and let $\mathcal{D}_\mathcal{U}$ denote the uniform distribution. The collected samples $x_t \in \mathcal{X}$ form the empirical distribution $\mathcal{D}_X$. To perform the hypothesis test we use the log-likelihood ratio introduced in Equation (3.21).

In Section 3.1.3.2 we introduced the statistical distance as a means to estimate the sample size need to distinguish two distributions $\mathcal{D}_\mathcal{C}$ and $\mathcal{D}_\mathcal{U}$. It was defined as

$$\varepsilon = |\mathcal{D}_\mathcal{C} - \mathcal{D}_\mathcal{U}| = \frac{1}{2} \sum_{x=0}^{2^{32}-1} |\mathcal{D}_\mathcal{C}(x) - \mathcal{D}_\mathcal{U}(x)|, \tag{8.12}$$

where $\mathcal{D}_\mathcal{U}(x) = 1/2^{32}$, $\forall x \in \mathcal{X}$. As mentioned in the same section, we need

$$N \approx 1/\varepsilon^2, \tag{8.13}$$

samples to distinguish $\mathcal{D}_\mathcal{C}$ from $\mathcal{D}_\mathcal{U}$.

We will now discuss a set of possible solutions for how to deal with the the unknown counter value $M^{(L)}$.

(*i*) One possible solution would be to guess the initial state of the counter $M_0 = (M^{(L)}_0, M^{(R)}_0)$ (in total 2^{64} combinations), and then construct 2^{64} empirical distribution from the given keystream. However, it will increase the time complexity of the distinguisher by 2^{64} times;

(*ii*) Another possibility is to guess the first 32 bits $M^{(R)}_0$ of the initial value of the counter M_0, i.e., 2^{32} values. If we do so, then we always know when the upper 32 bits $M^{(L)}$ are increased, i.e., at any time t we can express $M^{(L)}_t$ as follows.

$$M^{(L)}_t = M^{(L)}_0 \boxplus \Delta_t, \tag{8.14}$$

where Δ_t is known at each time, as $M_t^{(R)}$ is known. Recall from (8.11), that the noise variable $N_t^{(tot)}$ is expressed as $x_t \oplus M_{t+14}^{(L)}$. However, this expression can also be approximated as

$$x_t \oplus (M_0^{(L)} \boxplus \Delta_{t+14}) \rightarrow x_t \oplus (M_0^{(L)} \oplus \Delta_{t+14} \oplus N_2), \tag{8.15}$$

where N_2 is a new noise variable due to the approximation of the kind "$\boxplus \Rightarrow \oplus$". As $M_0^{(L)}$ can be regarded as a constant for every sample x_t, it only "shifts" the distribution, but will not change the bias. A shift of the uniform distribution is again the uniform distribution, so, the distance between the noise and the uniform distribution will remain the same. This solution requires 2^{32} guesses, and also introduce a new noise variable N_2;

(*iii*) A third solution could be to consider the sum of two consecutive samples $x_t \oplus x_{t+1}$. As $M^{(L)}$ changes slowly, then with probability $(1-2^{-32})$ we have $M_t^{(L)} = M_{t+1}^{(L)}$, and this term will be eliminated from the expression for the new sample. Unfortunately, this method will decrease the bias significantly, and then the number of required samples N will be much larger than in the previous cases.

In our attack we tried different solutions, and based on simulations we decided to choose solution (2) for our attack, as it has the lowest attack complexity.

The remaining questions is how to calculate the noise distribution $\mathcal{D}_{\mathcal{C}}$.

8.4 Calculation of the Noise Distribution

Consider the expression for the noise variable $x_t \oplus M_{t+14}^{(L)} = N_t^{(a)} \oplus N_{t+14}^{(e)}$. For simplicity in the formula, we omit time instances for variables,

$$\begin{aligned} N_t^{(tot)} = \;& N_t^{(a)} \oplus N_{t+14}^{(e)} = \\ = \;& (a \boxplus f'') \oplus \Big(\big(f'' \oplus G_2(c'')\big) \boxplus \big(c'' \oplus H_2(r_2)\big) \Big) \oplus \\ & \oplus (a \boxplus d'') \oplus \Big(\big(d'' \oplus G_1(a'')\big) \boxplus \big(a'' \oplus H_1(r_4)\big) \Big) \oplus \\ & \oplus H_1(r_1) \oplus H_3(r_3). \end{aligned} \tag{8.16}$$

We propose two ways to calculate the distribution of the total noise random variable $N_t^{(tot)}$. Let us truncate the word size by n bits (when we consider the expression modulo 2^n). Then in the first case the computational

complexity is $\mathcal{O}(2^{4n})$. This complexity is too high and, therefore, requires the noise variable to be truncated by some number of bits $n \ll 32$, much less than 32 bits. The second solution has a better complexity $\mathcal{O}(n2^n)$, but introduces two additional approximations into the expression, which makes the calculated bias smaller than the real value, i.e., by this solution we can verify the lower bound for the bias of the noise variable. Below we describe two methods and give our simulation results on the bias of the noise variable $N_t^{(tot)}$.

8.4.1 Truncate the word size

Consider the expression (8.16) taken by modulo 2^n, for some $n = 1 \dots 32$. Then the distribution of the noise variable can be calculated by the following steps.

(*i*) Construct four distributions, two of them are conditioned

$$\begin{aligned} &\mathcal{D}_{(G_2(c'') \mod 2^n | c'')}, \\ &\mathcal{D}_{(G_1(a'') \mod 2^n | a'')}, \\ &\mathcal{D}_{(H_1(r_4) \mod 2^n)}, \\ &\mathcal{D}_{(H_2(r_2) \mod 2^n)}. \end{aligned} \tag{8.17}$$

If the inputs to the H_i functions are random, their distributions are the same, i.e., $\mathcal{D}_{(H_1(r_4) \mod 2^n)} = \mathcal{D}_{(H_2(r_2) \mod 2^n)}$. Hence, it is sufficient to determine one of them. The algorithm requires one loop for c'' (a'' and x) of size 2^{32}. The time required is $3 \cdot 2^{32}$;

(*ii*) Afterward, construct two more conditioned distributions

$$\begin{aligned} &\mathcal{D}_{((f'' \oplus G_2(c'')) \boxplus (c'' \oplus H_2(r_2)) \mod 2^n | f'')}, \\ &\mathcal{D}_{((d'' \oplus G_1(a'')) \boxplus (a'' \oplus H_1(r_4)) \mod 2^n | d'')}. \end{aligned} \tag{8.18}$$

This requires four loops for $f'', c'', G_2(c'') \mod 2^n$, and $H_2(r_2) \mod 2^n$, which takes time $\mathcal{O}(2^{4n})$ (and similar for the second distribution);

(*iii*) Then, calculate another two conditioned distributions

$$\begin{aligned} \mathcal{D}_{(\mathtt{Expr}_1 | a)} &= \mathcal{D}_{((a \boxplus f'') \oplus (f'' \oplus G_2(c'')) \boxplus (c'' \oplus H_2(r_2)) \mod 2^n | a)}, \\ \mathcal{D}_{(\mathtt{Expr}_2 | a)} &= \mathcal{D}_{((a \boxplus d'') \oplus (d'' \oplus G_1(a'')) \boxplus (a'' \oplus H_1(r_4)) \mod 2^n | a)}. \end{aligned} \tag{8.19}$$

Each takes time $\mathcal{O}(2^{3n})$;

(*iv*) Finally, combine the results, partially using FHT, and then calculate the bias of the noise:

$$\mathcal{D}_{\mathcal{C}} = \mathcal{D}_{N^{(tot)}} = \mathcal{D}_{(\mathtt{Expr}_1|a)} \oplus \mathcal{D}_{(\mathtt{Expr}_2|a)} \oplus \mathcal{D}_{H_1} \oplus \mathcal{D}_{H_3}. \tag{8.20}$$

This will take time $\mathcal{O}(2^{3n} + 3n \cdot 2^n)$.

This algorithm calculates the exact distribution of the noise variable taken modulo 2^n, and has the complexity $\mathcal{O}(2^{4n})$. Due to such a high computational complexity we could only manage to calculate the bias of the noise when $n = 8$ and $n = 10$:

$$\begin{aligned} \varepsilon_I|_{n=8} &= 2^{-80.59}, \\ \varepsilon_I|_{n=10} &= 2^{-80.57}. \end{aligned} \tag{8.21}$$

8.4.2 Approximate $\boxplus$ with $\oplus$

Consider two additional approximations of the second $\boxplus$ to $\oplus$ in (8.16). Then, the total noise can be expressed as

$$\begin{aligned} N_t^{(tot)} = \quad & H_1(r_1) \oplus H_2(r_2) \oplus H_3(r_3) \oplus H_1(r_4) \oplus \\ & \oplus \big(G_2(c'') \oplus c''\big) \oplus \big(G_1(a'') \oplus a''\big) \oplus \\ & \oplus N_3 \oplus N_{2,a} \oplus N_{2,e}, \end{aligned} \tag{8.22}$$

where

$$N_3 = (a \boxplus f'') \oplus (a \boxplus d'') \oplus f'' \oplus d'',$$

and $N_{2,a}$ and $N_{2,e}$ are two new noise variables due to the approximation $\boxplus \Rightarrow \oplus$, i.e., $N_{2,a} = (x \boxplus y) \oplus (x \oplus y)$, for some random inputs x and y, and similar for $N_{2,e}$. Introduction of two new noise variables will statistically make the bias of the total noise variable smaller, but it can give us a lower bound of the bias, and also allow us to operate with distributions of size 2^{32}.

First, calculate the distributions $\mathcal{D}_{(H_i)}$, $\mathcal{D}_{(G_1(a'') \oplus a'')}$ and $\mathcal{D}_{(G_1(c'') \oplus c'')}$, each takes time 2^{32}. The expressions for $N_{2,a}, N_{2,e}$ and N_3 belong to the class of *pseudo-linear functions modulo* 2^n (PLFM), which was introduced in [MJ05b]. In the same paper, algorithms for construction of their distributions were also provided, which take time around $\mathcal{O}(\delta \cdot 2^n)$, for some small δ. The last step is to perform the convolution of precomputed distribution tables via FHT in time $\mathcal{O}(n2^n)$. Algorithms (PLFM distribution construction and computation of convolutions) and data structures that operates on large distributions are given in [MJ05b]. If we consider $n = 32$, then the total time

complexity to calculate the distribution table for $N^{(tot)}$ will be around 2^{38} operations, which is feasible for a common PC. It took us a few days to accomplish such calculations on a usual PC with memory 2Gb and 2×200Gb of HDD, and the received bias of $N^{(tot)}$ was

$$\varepsilon_{II}|_{n=32} = 2^{-74.515}. \tag{8.23}$$

If we also approximate $(M_0^{(L)} \boxplus \Delta_t) \to (M_0^{(L)} \oplus \Delta_t) \oplus N_2$, and add the noise N_2 to $N^{(tot)}$, we receive the bias

$$\varepsilon_{II}^{\Delta}|_{n=32} = 2^{-77.5}, \tag{8.24}$$

which is the lower bound. This means that our distinguisher requires approximately 2^{155} words of the keystream, according to 8.12.

8.5 Attack Scenarios

In the previous section we have shown how to sample from the given keystream, where 32 bit samples are drawn from the noise distribution with the bias $\varepsilon_{II}^{\Delta}|_{n=32} = 2^{-77.5}$. Our distinguisher needs around 2^{155} words of the keystream to successfully distinguish the cipher from random. Unfortunately, we have to guess the lower 32 bits $M_0^{(R)}$ to construct the empirical distribution correctly. This guess increases the time complexity of our attack to 2^{187}, and requires memory 2^{32}. The algorithm of our distinguisher for Dragon is given in Figure 8.4.

The time complexity can easily be reduced down to 2^{155}, if memory of size 2^{96} is available. Assume that we first construct a special table $T[\Delta][x] = \#\{t \equiv \Delta \mod 2^{64}, x_t = x\}$, where the samples are taken as $x_t = a'_t \oplus e'_{t+14}$. Afterward, for each guess of $M_0^{(L)}$, the empirical distribution $\mathcal{D}_X$ is constructed from the table T in time 2^{96}. Hence, the total time complexity will be $2^{155} + 2^{32} \cdot 2^{96} \approx 2^{155}$. This scenario is given in Figure 8.5. In the figure $(\Delta \boxplus M_0^{(R)}) \gg 32$ denotes a binary shift of $\Delta \boxplus M_0^{(R)}$32 steps to the right.

8.6 Summary

Two versions of a distinguishing attack on Dragon were presented. The first scenario required a computational complexity of 2^{187} and needed memory only 2^{32}. The second scenario had a lower time complexity around 2^{155}, but required a larger amount of memory 2^{96}. These attacks show that Dragon does not provide full security and can successfully be broken much faster than the exhaustive search, when a key of 256 bits is used.

for $0 \leq M_0^{(R)} < 2^{32}$

$\mathcal{D}_X(x) = 0, \ \ \forall x \in \mathbb{Z}_{2^{32}}$

$\Delta = 0$ (or $= -1$, if $M_0^{(R)} = 0$)

for $t = 0, 1, \ldots, 2^{155}$

if $(M_0^{(R)} \boxplus t) = 0$ **then** $\Delta = \Delta \boxplus 1$

$x_t = a'_t \oplus e'_{t+14} \oplus \Delta$

$\mathcal{D}_X(x_t) = \mathcal{D}_X(x_t) + 1$

end for

$$LLR = \sum_{x \in \mathbb{Z}_{2^{32}}} \mathcal{D}_X(x) \cdot \log_2 \left(\frac{\mathcal{D}_{\mathcal{C}}(x)}{2^{-32}} \right)$$

If $LLR \geq 0$ **break and output:** Dragon

end for

output: Random source

Figure 8.4: The distinguisher for Dragon (Scenario I).

From the specification of Dragon we also note that the amount of keystream for an unique pair initialization vector (IV) and key is limited to 2^{64}. This makes our attacks infeasible, but our results show that there are some structural weaknesses in Dragon.

Several new stream cipher proposals are based on NLFSRs. It is important to study such primitives, as it could be an interesting replacement for the widely used LFSR based stream ciphers.

for $0 \leq t < 2^{155}$

$T[t \mod 2^{64}][a'_t \oplus e'_{t+14}] + +$

end for

for $M_0^{(R)} = 0, \ldots, 2^{32} - 1$

for $\Delta = 0, \ldots, 2^{64} - 1$

for $x = 0, \ldots, 2^{32} - 1$

$\mathcal{D}_X\left(x \oplus \left((\Delta \boxplus M_0^{(R)}) \gg 32\right)\right) += T[\Delta][x]$

end for

end for

$$LLR = \sum_{x \in \mathbb{Z}_{2^{32}}} \mathcal{D}_X(x) \cdot \log_2\left(\frac{\mathcal{D}_{\mathcal{C}}(x)}{2^{-32}}\right)$$

If $LLR \geq 0$ **break** and **output:** Dragon

end for

output: Random source

Figure 8.5: Distinguisher for Dragon with lower time complexity (Scenario II).

9

A Square Root Resynchronization Attack

It is well known that a block cipher in either counter mode or OFB mode can be easily distinguished from a random sequence using $2^{n/2}$ keystream blocks, where n is the block size. In this chapter we give an extension to the attack scenario for the generic distinguishing attacks on block ciphers in OFB mode, we also show that the attack is applicable to the eSTREAM candidates Lex and the Pomaranch family of stream ciphers.

Assume that resynchronization of the primitive is performed using a fixed resync, see Section 3.6 i.e., we can perform an attack under the known IV/known plaintext scenario, see Section 1.3.1. Under this assumption we show that we can not only distinguish the keystream from a random sequence, we can also recover some plaintext if only ciphertext is given. Hence, the attack can be used for instance deduction.

The chapter is based on the papers [EHJ07a, EHJ07b] and the outline is as follows. We first describe the classical distinguishing attack on block ciphers in OFB mode in Section 9.1. We describe the new attack scenario in Section 9.2. In Section 9.3 we describe the attack on an eSTREAM candidate called LEX. In Section 9.4 we describe the attack on a family of eSTREAM candidates called the Pomaranch family. Finally we summarize the chapter in Section 9.5.

9.1 Classical Distinguishing Attack on OFB mode

In this section we describe the generic distinguishing attack that applies to block ciphers used in OFB. We use n to denote the block size in bits of the underlying cipher.

Output feedback mode (OFB) is a block cipher mode that turns the block cipher into a synchronous stream cipher as described in Section 1.2.2.1. The n-bit keystream words $(z_0, z_1, z_2 \ldots)$ are generated by repeatedly encrypting an n-bit IV. Let $z_{-1} = IV$, then

$$z_i = E_K(z_{i-1}), \quad i \geq 0. \tag{9.1}$$

As a block cipher defines a permutation over all n-bit blocks, we expect the average period of the keystream to be in the order of 2^{n-1} blocks. Thus, we expect that there is no collision in the first 2^{n-1} keystream blocks. On the other hand, if we have a collision, then we know that all subsequent blocks will be the same. I.e., if $z_i = z_j$ $(i \neq j)$, then $z_{i+k} = z_{j+k}$ $(k \geq 0)$. According to the birthday paradox, in a truly random sequence we expect to find a collision after observing $2^{n/2}$ n-bit blocks. This suggests that the keystream sequence can be distinguished from random by observing approximately $2^{n/2}$ keystream blocks. The distinguisher is given in Figure 9.1.

Input($z_0, z_1, \ldots, z_{2^{n/2}}$)
if ($z_i = z_j$ and $z_{i+1} \neq z_{j+1}$ for some $i \neq j$)
    return Random
else
    return OFB Mode

Figure 9.1: Distinguisher for OFB mode

9.2 New Attack Scenario

In this section we extend the basic distinguishing scenario given in the previous section. We show that in our scenario the adversary will get much more information than from a distinguishing attack. Some part of the plaintext will actually be recovered. The attack is generic for any stream cipher. Let us divide the internal state of the cipher into two parts,

$$State = (State_K,\ State_{K+IV}), \tag{9.2}$$

where $State_K$ is a part of the state that is only affected by the key and $State_{K+IV}$ is a part of the state that is affected by 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. For simplicity we denote $L = |State_{K+IV}|$.

Consider a resynchronization scenario. 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 N plaintext bits corresponding to every ciphertext. Our goal is to recover the rest of the plaintext for one of the messages.

A sequence of l consecutive keystream bits is called a *sample* and is denoted by x, i.e., $x_t = (z_t, z_{t+1}, \ldots, z_{t+l-1})$. Let us first store k samples from IV_0 in a table. Assume that the cipher is initialized using several IVs and we know at least the l first bits of the corresponding keystream. After how many initializations do we expect to produce a sample which is already stored in our table? This problem corresponds to the second birthday problem described in Section 3.1.1. In Section 3.1.1 it was showed that if $k = \mathcal{O}(\sqrt{L})$ and $N = \mathcal{O}(\sqrt{L})$, then probability of a collision is approximately

$$1 - e^{-\frac{kN}{L}}. \tag{9.3}$$

Denote the IV producing the collision by IV_c. 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. In other words, by knowing the first l corresponding plaintext bits of a ciphertext, we can decrypt the rest of the ciphertext without knowing the keys. The attack is visualized in Figure 9.2.

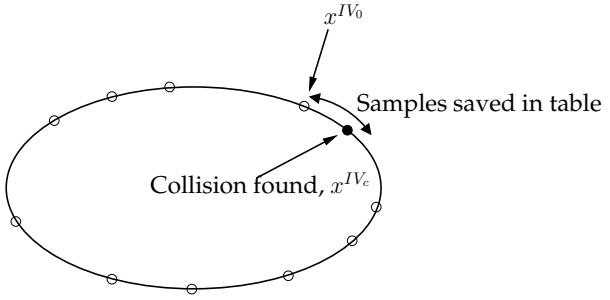


Figure 9.2: State graph for a fixed key, samples are visualized by circles.

This resynchronization scenario will not apply to a block cipher in counter mode. In counter mode there is no state that is updated pseudo randomly.

The value that is encrypted is a counter and thus, it will never repeat. This is the reason why counter mode can be distinguished from a random sequence but also the reason why the resynchronization state collision attack described above will fail.

However, the above scenario applies to block ciphers used in OFB mode as z_i can be seen as the part of the state depending on both the key and the IV, $State_{K+IV}$ and the key used in the block cipher can be seen as the state only depending on the key, $State_K$. By initializing OFB mode with a new IV, the cipher will enter a new random state after every encryption.

Assume that $2^{\beta L}$ ($0 \leq \beta \leq 1$) samples of length l from a keystream sequence of $2^{\beta L} + l$ bits, originating from IV_0 and key K, is saved in a table. The table is then sorted with complexity $\mathcal{O}(\beta L \cdot 2^{\beta L})$. This table covers a fraction of $2^{-(1-\beta)L}$ of the entire cycle. The number of samples (IVs with l known keystream bits) is according to Equation (9.3) approximately $2^{(1-\beta)L}$. For each sample, a logarithmic search with complexity $\mathcal{O}(\beta L)$ 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 $l \approx L$. The attack complexities are then given by

$$\begin{aligned} Keystream: &\quad 2^{\beta L} + L \text{ bits from one IV and} \\ &\quad L \text{ bits from } 2^{(1-\beta)L} \text{ IVs} \\ Time: &\quad \beta L 2^{\beta L} + \beta L 2^{(1-\beta)L} \\ Memory: &\quad L 2^{\beta L} \end{aligned}$$

where $0 \leq \beta \leq 1$. 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 L.

The proposed attack is summarized in Figure 9.3. In the figure, $x_t^{IV_i}$

```
for i = 1, ..., 2^{βL}
   T[i] = x_{t+i}^{IV_0}
end for
sort T
for i = 1, ..., 2^{(1-β)L}
   if x_t^{IV_i} ∈ T
      return cipher
end for
return random
```

Figure 9.3: Summary of square root attack.

represents a sample from the keystream from IV_i at time t, and T represents the table where samples are stored.

9.2.1 Distinguished Points

In practice the limiting factor of the algorithm is to store the table of size $2^{\frac{L}{2}}$ in memory. To reduce the size of the table one can use distinguished points, i.e., not every word is saved in the table. For example, only words that end with t zeros are saved in the table. This would reduce the size of the table to in average $2^{\frac{L}{2}-t}$. On the other hand the price we have to pay is higher computational complexity in the attack phase as we have to clock the cipher on average 2^t times for every IV before an occurrence with t zeros at the end occur in a word. Also, we loose the power to predict future keystream symbols and the attack becomes a pure distinguishing attack. The attack using distinguished points is summarized in Figure 9.4.

```
j=0
for i = 1, ..., 2^(L/2)
   if x_{t+i}^{IV_0} fulfills requirement
      T[j] = x_{t+i}^{IV_0}
      j++
end for
sort T
for i = 1, ..., 2^(L/2)
   k = 0
   while x_{t+k}^{IV_i} do not fulfill requirement
      k++
   end while
   if x_{t+k}^{IV_i} ∈ T
      return cipher
end for
return random
```

Figure 9.4: Summary of square root attack using distinguished points.

9.3 Analysis of Leak Extraction (LEX)

LEX [Bir06] is a stream cipher submitted as a Profile 1 candidate to the eSTREAM project. It is very similar to a block cipher used in output feedback mode. Instead of taking the result of a block cipher encryption as keystream, the keystream is taken as a part of the state after each round. The idea behind LEX can be applied to many different block ciphers and can thus in some sense be seen as a mode of operation. However, leaking the wrong part of the state can be devastating for the security of the cipher and how to leak the state must be carefully considered depending on which block cipher to use. LEX in the eSTREAM project uses AES as block cipher. The state of consists of 16 bytes, denoted by $b_t^{(i,j)},\ 0 \leq i,j \leq 3$ at time t. One round of AES is performed, using round key K_t, to produce the next state. This is illustrated in Figure 9.5. In the figure $K_i,\ 0 \leq i \leq 9$ denotes the

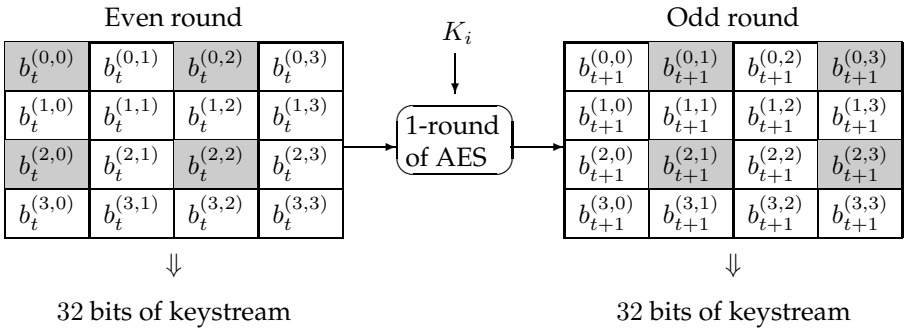


Figure 9.5: One round of the stream cipher LEX, at each time instant 32 bits of keystream is leaked from the internal state. The bytes leaked at each round is marked with gray, it is assumed in the figure that t is even. One round of AES is applied to the state to determine the next state.

round key where $i = t \pmod{10}$, there are in total ten 128-bit round keys in AES. Lex leaks 32 bits, i.e., 4 bytes, of the state in each round. These bytes are leaked according to

$$\boldsymbol{z_t} = \begin{cases} \left(b_t^{(0,0)}, b_t^{(2,0)}, b_t^{(0,2)}, b_t^{(2,2)}\right), & t \text{ odd} \\ \left(b_t^{(0,1)}, b_t^{(2,1)}, b_t^{(0,3)}, b_t^{(2,3)}\right), & t \text{ even} \end{cases} \tag{9.4}$$

128-bit AES uses 10 rounds to output 128 bits and thus, LEX leaks 320 bits in 10 rounds. Hence, LEX is 2.5 times faster than AES in e.g., OFB mode.

Consider the generic distinguishing attack on OFB mode where 2^{64} blocks of keystream can be used to distinguish the sequence from random. In LEX this generic distinguishing attack is not possible as the mapping from internal state to output is not one-to-one and thus collisions can occur.

On the other hand, consider the resynchronization scenario given in Section 9.2. The internal state consists of the 128-bit block in AES together with the secret key K. As keystream bits are output in every AES round it make sense to include the 10 rounds of AES in the internal state. The part of the state that is not fixed is thus $128 + \log_2 10 = 131.32$ bits. By knowing $2^{65.66}$ keystream bits from one IV and the first ≈ 132 bits from $2^{65.66}$ other IVs, we can decrypt the rest of the ciphertext corresponding to one IV, IV_c.

This attack does not stem from a structural weakness of LEX or AES. Instead it stems from the fact that the part of the state that holds the key bits is not updated unless the key is changed. Thus it would be applicable to any block cipher used as a stream cipher by leak extraction.

9.4 Analysis of Pomaranch

In this section we will look at the existing versions of Pomaranch, see Section 7.1 for a description of the Pomaranch family of ciphers. We show that the square root IV 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 only 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 each of length 18 bits, i.e., $L = 36$. Assume we choose $\beta = 0.5$ and a sample length of $l = 36$ bits. In order to collect 2^{18} samples, we need approximately $2^{18} + 36$ keystream bits from IV_0. These samples are stored in a table. We then samples from 2^{18} different IVs in order to find a collision. The simulation results are summarized in Table 9.2. We also verified the attack using three registers. The attack given in this section suggests a new design criteria for the Pomaranch family of stream ciphers, namely that the total register length must be twice the keysize.

Cipher		Table size	Number of IVs needed
Version 1		2^{63}	2^{63}
Version 2	80 bit	2^{42}	2^{42}
	128 bit	2^{63}	2^{63}
Version 3	80 bit	2^{54}	2^{54}
	128 bit	2^{64}	2^{98}

Table 9.1: Summary of attack complexities on some Pomaranch ciphers.

		Number of IVs				
Success rate (%)		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 9.2: Simulation results using 2 register Pomaranch version 3 with linear filter function, the table summarizes how many times the attack succeeds out of 100 attacks for a specific table size and number of IVs.

9.5 Summary

We have presented a general IV attack that works for all ciphers where the keysize is larger than half of the state size, when the part of the state only affected by the key is not considered.

We show that the attack is applicable to block ciphers in OFB mode and also to the eSTREAM candidate LEX. The attack was demonstrated on all versions and variants of Pomaranch with complexity far below exhaustive search. The attack will not recover the key but is still much stronger than a distinguishing attack. By using keystream from multiple IVs is is possible to recover the plaintext corresponding to one IV if only the ciphertext together with the first few keystream bits are known. The attack scenario here is somewhat similar to the attack scenarios in disk encryption, see e.g., [Gjø05], where the adversary has write access to the disk encryptor and read access to the storage medium.

10

A Framework for Chosen IV Statistical Analysis of Stream Ciphers

Depending on the protocol, different resynchronization mechanisms can be used as described in Section 3.6. In this chapter we will consider resynchronization using requested resync, i.e., one of the participants of a transmission requests a resynchronization. In such a system, an assailant is assumed to be able to actively choose the IV. Hence, the cipher should be designed to resist chosen-IV-known-plaintext attacks.

Daemen, Govaerts and Vandewalle [DGV94] presented an resynchronization attack on nonlinear filter generators with linear resynchronization and filter function with few inputs. This attack is extended to the case where the filter function is unknown by Golić and Morgari [GM03a]. More extensions of resynchronization attacks was done by Armknecht, Lano and Preneel [ALP04].

To avoid such attacks, the initialization of stream ciphers in which the internal state variables are determined using the secret key and the public IV should be designed carefully. In most ciphers, first the key and IV is loaded into the state variables, then a next state function is applied to the internal state iteratively for a number of times without producing any output. The number of iterations play an important role on both security and the

efficiency of the cipher. It should be chosen so that each key and IV bit affect each initial state bit in a complex way. On the other hand, using a large number of iterations is inefficient and may hinder the speed for applications requiring frequent resynchronizations.

Filiol introduced tests to evaluate the statistical properties of symmetric ciphers using the number of the monomials in the Boolean functions that simulate the action of a given cipher [Fil01]. Saarinen recently proposed to extend these ideas to a chosen IV statistical attack, called the d-monomial test, and used it to find weaknesses in several proposed stream ciphers [Saa06] .

In this chapter we generalize this idea and propose a framework for chosen IV statistical attacks using a polynomial description. The statistical tests are general distinguishing attacks performed under the chosen IV/known plaintext scenario. The basic idea is to select a subset of IV bits as variables. Assuming all other IV values as well as the key being fixed, we can write a keystream symbol as a Boolean function of the selected IV variables. By running through all possible values of these bits and creating a keystream output for each of them, we create the truth table of this Boolean function. We now hope that this Boolean function has some statistical weaknesses that can be detected. We describe the d-monomial test in this framework, and then we propose two new tests, called the monomial distribution test and the Maximal Degree Monomial Test.

We then apply them on some eSTREAM stream cipher proposals, and give some conclusions regarding the strength of their IV initialization. In particular, we experimentally detected statistical weaknesses in the keystream of Grain-128 with IV initialization reduced to 192 rounds as well as in the keystream of Trivium using an initialization reduced to 736 rounds. Furthermore, we repeat our experiments to study the statistical properties of internal state bits. Here we could detected statistical weaknesses in some state bits of Grain-128 with full IV initialization. In the context, we also propose alternative initial loadings for some of the ciphers so that the diffusion is satisfied in fewer rounds.

The chapter is based on the paper [EJT07] and the outline of the chapter is the following. Section 10.1 gives some background information about Boolean functions. In Section 10.2, the suggested framework for chosen IV statistical attacks is presented. A generalized approach to statical chosen IV analysis is proposed in Section 10.3. We present a test called the Monomial distribution test in Section 10.4 and a test called the Maximal Degree Monomial Test in Section 10.5. Some results are presented for reduced round initializations of the ciphers Grain [HJMM06], Trivium [CP05], Decim [BBC$^+$06] and LEX [Bir06] in Section 10.6. Finally we summarize the chapter in Section 10.7.

```
COMPUTE ANF(v)

for i = 1, ..., n
   for j = 1, ..., 2^{n-1}
      t_j = v_{2j-1}
      u_j = v_{2j-1} ⊕ v_{2j}
   end for
   v = (t, u)
end for
```

Figure 10.1: Algorithm to compute the ANF in vector v from the truth table in v.

10.1 Boolean Functions

Assume that the truth table of an n-variable Boolean function is represented in a vector $\boldsymbol{v}$ of size 2^n and the ANF of the Boolean function can be calculated with complexity $O(n2^n)$ using the algorithm presented in Figure 10.1, which uses two auxiliary vectors $\boldsymbol{t}$ and $\boldsymbol{u}$, both of size 2^{n-1}.

Let $f : \mathbb{F}_2^n \to \mathbb{F}_2$ be a Boolean function, and let M denote the number of monomials in the ANF of f. If f is randomly chosen, each monomial is included with probability one half, i.e., a Bernoulli distribution. The sum of Bernoulli distributed random variables is binomially distributed, hence $M \sim$Bin$(2^n, \frac{1}{2})$, with expected value $E(M) = 2^{n-1}$. Let us denote the number of monomials of degree k by M_k, i.e., $M = \sum_{k=0}^{n} M_k$. The distribution of M_k is Bin$(\binom{n}{k}, \frac{1}{2})$ with $E(M_k) = \frac{1}{2}\binom{n}{k}$.

Let m_k be an observation from M_k, let α be the significance level of the test, and let $n+1$ be the number of degrees of freedom of the test. Then

$$\chi^2 = \sum_{k=0}^{n} \frac{\left(m_k - \frac{1}{2}\binom{n}{k}\right)^2}{\frac{1}{2}\binom{n}{k}} \xrightarrow{d} \chi^2_\alpha(n+1), \text{ when } \binom{n}{k} \to \infty. \tag{10.1}$$

If $\binom{n}{k}$ is large enough, methods described in Section 3.1.3.1 can be used to perform a hypothesis test to decide if the function in question has a deviant number of monomials of degree k.

10.2 A Framework for Chosen IV Statistical Attacks

Different tests have been introduced to evaluate statistical properties of sequences from symmetric ciphers and hash functions. The tests are usually based on taking one long keystream sequence and then applying different statistical tests, like the NIST statistical test suit used in the AES evaluation [NIS].

However, several researchers have recently noted the possibility to instead generate many short keystream sequences, from different chosen IV values and look at the statistical properties of, say, only the first output symbol of each keystream. One such example is the observation by Shamir and Mantin that the second byte in RC4 is strongly biased [FMS01a].

Based on work by Filiol [Fil01], Saarinen [Saa06] recently proposed the d-monomial chosen IV distinguisher, introducing a test called the d-monomial test. The behavior of the keystream is analyzed using a function of n IV bits, i.e., $z = f(iv_0, \ldots, iv_{n-1})$. All other IV and key bits are considered to be constants. For a chosen parameter d (set to be a small value), the d-monomial test counts the number of monomials of Hamming weight d, denoted m_d, $0 \leq d \leq n$, in the ANF of f and compares it to its expected value $\frac{1}{2}\binom{n}{d}$, using the χ^2 goodness-of-fit test with one degree of freedom. In this chapter, we will use a slightly altered d-Monomial test where we sum the test statistics for each d and evaluate the result using $n + 1$ degrees of freedom. The algorithm for the d-Monomial test is summarized in Figure 10.2.

The complexity of this attack is $O(n2^n)$ operations and it needs memory $O(n2^n)$. The downside of this method is that statistical deviations for lower and higher degree monomials are hard to detect as their numbers are few. So even if the maximal degree monomial never occurs, the test does not detect this anomaly. In the next section we will present alternative attacks that solves this problem.

10.3 A Generalized Approach

We suggest to use a generalized approach. Instead of analyzing just one function in ANF form, we can study the behavior of more polynomials so that monomials that are more (or less) probable than others can be detected.

Let us select n IV bit variables, denoted $iv_0, \ldots, iv_{n-1}$, as our *variables*. The remaining IV values as well as key bits are kept constant. Using the first output symbol, $z_0 = f_1(iv_0, \ldots, iv_{n-1})$, for each choice of $iv_0, \ldots, iv_{n-1}$, the ANF of f_1 can be constructed.

d-MONOMIAL TEST

for $iv = 1, \ldots, 2^n - 1$
  Initialize cipher with $\boldsymbol{iv}$
  $\boldsymbol{v}[\boldsymbol{iv}]$=first keystream bit after initialization
end for
Compute ANF of vector $\boldsymbol{v}$ and store result in $\boldsymbol{v}$.
for $i = 1, \ldots, 2^n - 1$
  if $\boldsymbol{v}[i] = 1$
    d= hamming weight of monomial i
    m_d++
end for
for $d = 0, \ldots, n$
  $\chi^2 += \frac{\left(m_d - \frac{1}{2}\binom{n}{d}\right)^2}{\frac{1}{2}\binom{n}{d}}$.
if $\chi^2 > \chi^2_\alpha(n+1)$
  return *cipher*
else
  return *random*

Figure 10.2: Summary of the d-monomial test, complexity $O(n2^n)$.

The new approach is now to do the same again, but using some other choice on IV values outside the IV variables. Running through each choice of $iv_0, \ldots, iv_{n-1}$ in this case gives us a new function f_2. Continuing in this way, we derive P different Boolean functions $f_1, f_2, \ldots, f_P$ in ANF form. In some situations, it might also be possible to obtain polynomials from different keys, where the same IVs have been used.

Having P different polynomials in our possession we can now design any test that looks promising, taken over all polynomials. The d-monomial test would appear for the special case $P = 1$, and the test being counting the number of weight d monomials. We now propose in detail two different tests.

10.4 The Monomial Distribution Test

The attack scenario is similar to the d-monomial test, but instead of counting the number of monomials of a certain degree, we generate P polynomials and calculate in how many of the polynomials each monomial is present. That is, we generate P polynomials of the form (10.2) and count the number

of occurrences of $a_i = 1,\ 0 \leq i \leq 2^n - 1$, where

$$f = a_0 + a_1x_1 + \ldots + a_{n+1}x_1x_2 + \ldots + a_{2^n-1}x_1x_2 \ldots x_{n-1}x_n \qquad (10.2)$$

Denote the number of occurrences of coefficient a_i by m_{a_i}. As each monomial should be included in a function with probability $1/2$, i.e., $P(a_i = 1) = 0.5,\ 0 \leq i \leq 2^n - 1$, the number of occurrences is binomially distributed with expected value $E(M_{a_i}) = P/2$ for each monomial. We will as previously perform a χ^2 goodness-of-fit test with 2^n degrees of freedom, as described by Equation (10.3),

$$\chi^2 = \sum_{i=0}^{2^n-1} \frac{(m_{a_i} - \frac{P}{2})^2}{\frac{P}{2}}. \qquad (10.3)$$

If the observed amount is larger than some tabulated limit $\chi^2_\alpha(2^n)$, for some significance level α and 2^n degrees of freedom, we can distinguish the cipher from a random one. The pseudo-code of the monomial distribution test is given in Figure 10.3.

This algorithm has a higher computational complexity than the d-Monomial attack, $O(Pn2^n)$, and needs the same amount of memory, $O(n2^n)$. On the other hand, if for a cipher some monomials are highly non-randomly distributed, the attack may be successful with less number of IV bits, i.e., smaller n, compared to the d-monomial test. Additionally, although this attack is originally proposed for the chosen IV scenario of a fixed unknown key, it is also possible to apply the test for different key values, if the same IV bits are considered.

10.5 The Maximal Degree Monomial Test

A completely different and very simple test is to see if the maximal degree monomial can be produced by the keystream generator. The coefficient of the maximal degree monomial is the product of all IV bits and can hence only occur if all the IV bits have been properly mixed. In hardware oriented stream ciphers the IV loading is usually as simple as possible to save gates, e.g., the IV bits are loaded into different memory cells. The update function is then performed a number of steps to produce proper diffusion of the bits. Intuitively it will take many clockings before all IV bits meet in the same memory cell and even more clockings before they spread to all the memory cells and become nonlinearly mixed. The aim of the Maximal Degree Monomial Test is to check in a simple way whether the number of initial clockings is sufficient. As the maximal degree monomial is unlikely to exist if lower degree monomials do not exist, this is our best candidate to study. Hence, the existence of the maximal degree term in ANFs is a good indication to the satisfaction of diffusion criteria, especially completeness.

MONOMIAL DISTRIBUTION TEST

for $j = 1, \ldots, P$
  for $iv = 1, \ldots, 2^n - 1$
    Initialize cipher with iv
    $v[iv]$=first keystream bit after initialization
  end for
  Compute ANF of vector v and store result in v.
  for $i = 1, \ldots, 2^n - 1$
    if $v[i] = 1$
      $m_{a_i} + +$
  end for
end for
for $d = 0, \ldots, 2^n - 1$
  $\chi^2 + = \frac{(m_{a_d} - \frac{P}{2})^2}{\frac{P}{2}}$.
end for
if $\chi^2_\alpha(2^n)$
  return *cipher*
else
  return *random*

Figure 10.3: Summary of Monomial distribution test, complexity $O(Pn2^n)$.

According to the Reed-Muller transform [FL03], the maximal degree monomial can be calculated as the XOR of all entries in the truth table. So the test is similar to the previous tests performed by initializing the cipher with all possible combinations for n IV bits, $z^{iv_0,\ldots,iv_{n-1}} = f(iv_0, \ldots, iv_{n-1})$, all other bits are considered to be constants. The existence of the maximal degree monomial can be checked by XORing the first keystream bit from each initialization, this is equivalent to determining a_{2^n-1} in Equation (10.2),

$$a_{2^n-1} = \bigoplus_{iv_0,\ldots,iv_{n-1}} z^{iv_0,\ldots,iv_{n-1}}. \tag{10.4}$$

By for example changing some other IV bit we receive a new polynomial and perform the same procedure again, this is repeated for P polynomials. If the maximal degree polynomial never occurs in any of the polynomials or if it occurs in all of the polynomials we successfully distinguish the cipher. Hence we can, with low complexity and almost no memory, check whether the maximal degree monomial can exist in the output from the cipher. It

MAXIMAL DEGREE MONOMIAL TEST

for $j = 1, \ldots, P$
  $a_{2^n-1} = 0$
  for $iv = 1, \ldots, 2^n - 1$
    Initialize cipher with iv
    $z =$ first keystream bit after initialization
    $a_{2^n-1} = a_{2^n-1} \oplus z$
  end for
  if $a_{2^n-1} = 1$
    ones++
end for
if ones=0 or ones=P
  return *cipher*
else
  return *random*

Figure 10.4: Summary of Maximal Degree Test, complexity $O(P2^n)$.

is possible, with the same complexity, to consider other weak monomials, the coefficient can be calculated according to the Reed-Muller transform. The complexity of the Maximal Degree Monomial test is $O(P2^n)$ and it only requires $O(1)$ memory. The description of the test is given in Figure 10.4.

10.5.1 Other Possible Tests

We have proposed two specific tests that we will use in the sequel to analyze different stream ciphers. Our framework gives us the possibility to design many other interesting tests. As an example, a monomial distribution test restricted to only monomials with very high weight could be an interesting test. We have also performed some basic tests where we study the Walsh spectra of each polynomial.

10.6 Experimental Results

We applied the proposed tests described above on some of the Phase III eSTREAM candidates to evaluate their efficiency of initializations. We evaluated their security margin by testing reduced round versions of the ciphers. We also presented some results on the statistical properties of the internal

state variables.

The significance level of the hypothesis tests is chosen to be approximately $1-\alpha = 1-2^{-10}$. The tabulated results have a success rate of at least 90%. The required number of IVs, polynomials and the amount of memory needed to attack the ciphers are given in tables. Also, the results for initial state variables are presented with the percentage of weak initial state variables.

Hardware oriented stream ciphers use simple initial key and IV loading compared to software oriented ciphers. Generally, key and IV bits affect one initial state variable. Therefore, they require a large number of clockings to satisfy the diffusion of each input bit on each state bit. We repeated some of our simulations using alternative key/IV loadings in which each IV bit is assigned to more than one internal state bit and compared the results to the original settings. In the alternative loadings the hardware complexity is slightly higher, however on the other hand the cipher has more resistance to chosen IV attacks.

10.6.1 Grain-128

Grain-128 [HJMM06] is a hardware oriented stream cipher submitted as a Profile 2 candidate to the eSTREAM network of excellence. It uses an LFSR and a NLFSR together with a nonlinear filter function. In the initialization of Grain, a 128 bit key is loaded into the NLFSR and a 96 bit IV is loaded into the first 96 positions of the LFSR, the rest of the LFSR is filled with ones. The cipher is then clocked 256 times and for each clock the output bit is fed back into both the LFSR and the NLFSR.

	Rounds	P	IV bits	Memory
d-Monomial test	160	1	14	2^{14}
	192	1	25	2^{25}
Monomial distr. test	160	2^6	7	2^7
	192	2^6	22	2^{22}
Max. degree test	160	2^5	11	1
	192	2^5	22	1

Table 10.1: Number of IV bits needed to attack the first keystream bit of Grain-128 for different number of rounds in the initialization (out of 256 rounds).

In Table 10.1, the results obtained for a reduced version of Grain are given. The highest number of rounds we succeeded to break is 192 out of the original 256, which corresponds to the 75% of the initialization phase.

Figure 10.5 illustrates the growth in the number of IV bits needed to successfully distinguish the output for an increasing number of initialization clocking. Assume that we model the relationship between the number of

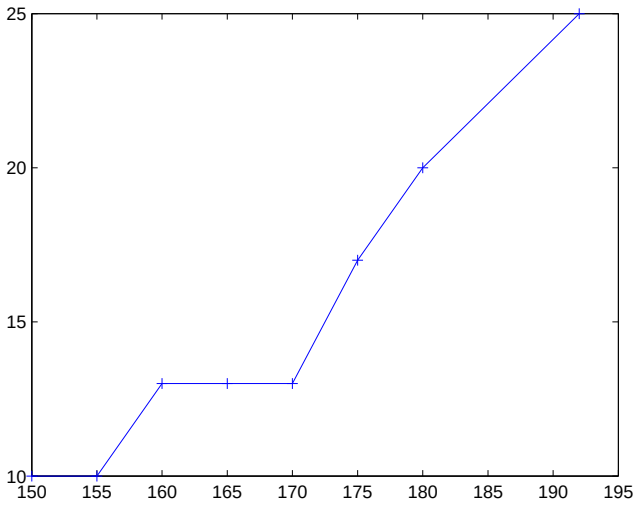


Figure 10.5: Number of IV bits needed to attack the first keystream bit of Grain-128 using the d-Monomial test for different number of rounds in the initialization (out of 256 rounds).

IVs and number of rounds using linear regression. Then the trend equation is obtained as $y = 0.3981x - 52.2195$ where y represents the number of IVs and x represents the number of rounds in initialization. The correlation coefficient of the model is 0.9685. Using this model, the prediction of required number of IVs to attack Grain with the d-monomial test is $y \approx 50$. However, the accuracy of a linear model is highly uncertain, more points need to be calculated before a conclusion about the security can be drawn.

In Table 10.2, the results of the experiments for initial state variables are presented. The number of weak initial state variables are three times better in the maximum degree test compared to the d-monomial test. The statistical deviations in state bits remain even after full initialization. These weak state bits are located in the left most positions of the feedback shift regis-

ters. To remove the statistical deviations in state variables, at least 320 initial clockings are needed. It is possible that if we use a larger number of IV bits, the weaknesses in state variables may also be observed from the keystream bits.

	Rounds	P	IV bits	Memory	Fraction
d-Monomial test	256	1	14	2^{14}	$33/256$
	256	1	20	2^{20}	$56/256$
	288	1	20	2^{20}	$0/256$
Monomial distr. test	256	2^6	8	2^8	$20/256$
	256	2^6	15	2^{15}	$44/256$
	288	2^6	20	2^{20}	$0/256$
Max. degree test	256	2^5	14	1	$108/256$
	256	2^5	16	1	$120/256$
	288	2^5	20	1	$73/256$

Table 10.2: Number of IV bits needed to attack the initial state variables Grain-128 for different number of rounds in the initialization (out of 256 rounds).

10.6.1.1 Alternative Key/IV Loading for Grain-128

Here we propose an alternative key/IV loading in which only the loading of the first 96 bits of the NLFSR is different from the original. Instead of directly assigning the key, we assign the modulo 2 summation of IV and the first 96 bits of the key. The proposed loading is very similar to the original and the increase in number of gates required is approximately 10-15%. In an environment where many resynchronizations are expected, one can reduce the number of initial clockings by using some more gates in the hardware implementation. In the new loading, each IV bit affects two internal state variables. We repeated our experiments using the new loading and the results are given in Table 10.3 and Table 10.4. Using alternative loading, Grain shows more resistance to the presented attacks, but still the statistical deviations in the state bits remain after full initialization.

	Rounds	P	IV bits	Memory
d-Monomial test	160	1	19	2^{19}
Monomial distr. test	160	2^6	20	2^{20}
Max. degree test	160	2^5	21	1

Table 10.3: Number of IV bits needed to attack the first keystream bit of Grain-128 with alternative key/IV loading for different number of rounds in the initialization (out of 256 rounds).

	Rounds	P	IV bits	Memory	Fraction
d-Monomial test	256	1	14	2^{14}	$1/256$
	256	1	16	2^{16}	$5/256$
	288	1	20	2^{20}	$0/256$
Monomial distr. test	256	2^6	8	2^8	$4/256$
	256	2^6	10	2^{10}	$10/256$
	288	2^6	20	2^{20}	$0/256$
Max. degree test	256	2^5	14	1	$100/256$
	256	2^5	16	1	$108/256$
	288	2^5	20	1	$47/256$

Table 10.4: Number of IV bits needed to attack the initial state variables of Grain-128 with alternative key/IV loading for different number of rounds in the initialization (out of 256 rounds).

10.6.2 Trivium

Trivium [CP05] is another hardware oriented stream cipher submitted to eSTREAM in Profile 2. It is based on NLFSRs an the state is divided into three registers which in total stores 288 bits. During the initialization the 80-bit key is inserted into the first register while an 80-bit IV is inserted into the second register. The cipher is clocked four full cycles before producing any keystream, i.e., 1152 clockings.

The results for Trivium are given in Table 10.5 and Table 10.6. For attacks

	Rounds	P	IV bits	Memory
d-Monomial test	608	1	12	2^{12}
	640	1	15	2^{15}
	672	1	20	2^{20}
	704	1	27	2^{27}
Monomial distr. test	608	2^5	9	2^9
	640	2^6	13	2^{13}
	672	2^6	18	2^{18}
	704	2^6	23	2^{23}
Max. degree test	608	2^5	9	1
	640	2^5	13	1
	672	2^5	18	1
	704	2^5	24	1

Table 10.5: Number of IV bits needed to attack the first keystream bit of Trivium for different number of rounds in the initialization (out of 1152 rounds).

on 736 and more rounds, the d-Monomial and the Monomial distribution attacks suffer from too large memory requirements. The Maximal Degree Monomial Test can be used to attack even 736 rounds (approximately 64% of initialization) using 33 IV bits. The attack on 736 rounds has only been performed a handful of times so the success rate is still an open issue in this case.

The percentage of weak initial state variables for Trivium are approximately the same using d-monomial and Maximal Degree Monomial Tests.

10.6.2.1 Alternative Key/IV Loading for Trivium

In the original key/IV loading, 128 bits of the initial state are assigned to constants and the key and IV bits affect only one state bit. Here, we propose an alternative initial key/IV loading in which the first register is filled with the modulo 2 summation of key and IV, the second register is filled with IV and the last register is filled with the complement of key plus IV. In this

	Rounds	P	IV bits	Memory	Fraction
d-Monomial test	608	1	12	2^{12}	$144/256$
	640	1	12	2^{12}	$57/256$
	672	1	15	2^{15}	$87/256$
	704	1	20	2^{20}	$74/256$
Monomial distr. test	608	2^5	12	2^{12}	$105/256$
	640	2^5	12	2^{12}	$29/256$
	672	2^5	15	2^{15}	$0/256$
	704	2^5	20	2^{20}	$12/256$
Max. degree test	608	2^5	12	1	$169/256$
	640	2^5	12	1	$86/256$
	672	2^5	15	1	$108/256$
	704	2^5	20	1	$76/256$

Table 10.6: Number of IV bits needed to attack the initial state variables of Trivium for different number of rounds in the initialization (out of 1152 rounds).

setting, each IV bit affects three internal state bits, therefore the diffusion of IV bits to the state bits is satisfied in less number of clockings. We repeated the tests using the alternative loading and obtained the results given in Table 10.7 and Table 10.8. In the alternative loading, the required number of IV bits and memory needed to attack Trivium are approximately 50 percent more compared to the original loading.

10.6.3 Decim

Decim-v2 [BBC$^+$06] is also a hardware oriented stream cipher based on a nonlinearly filtered LFSR and the irregularly decimation mechanism, ABSG. The internal state size of Decim-v2 is 192 bit and it is loaded with 80 bit key and 64 bit IV. The first 80 bits of the LFSR are filled with the key, the bits between 81 and 160 are filled with linear functions of key and IV and the last 32 bits are filled with a linear function of IV bits.

The results we obtained for Decim-v2 are given in Table 10.9 and Table

	Rounds	P	IV bits	Memory
d-Monomial test	608	1	18	2^{18}
Monomial distr. test	608	2^5	22	2^{22}
Max. degree test	608	2^5	17	1

Table 10.7: Number of IV bits needed to attack the first keystream bit of Trivium with alternative key/IV loading for different number of rounds in the initialization (out of 1152 rounds).

	Rounds	P	IV bits	Memory	Fraction
d-Monomial test	608	1	12	2^{12}	$4/256$
	640	1	18	2^{12}	$17/256$
	672	1	20	2^{15}	$0/256$
Monomial distr. test	608	2^5	12	2^{12}	$2/256$
	640	2^5	18	2^{12}	$19/256$
	672	2^5	20	2^{15}	$0/256$
Max. degree test	608	2^5	12	1	$21/256$
	640	2^5	18	1	$24/256$
	672	2^5	20	1	$0/256$

Table 10.8: Number of IV bits needed to attack the initial state variables of Trivium with alternative key/IV loading for different number of rounds in the initialization (out of 1152 rounds).

10.10. The security margin for Decim against chosen IV attacks is very large, the cipher can only be broken when not more than about 3% of the initialization is used. This is mainly because of the initial loading of key and IV in which each IV bits affect 3 state variables and the high number of quadratic terms in the filter function. The weakness in initial state variables can be observed for higher number of clockings. The number of weak initial state variables are approximately the same for all attacks.

	Rounds	P	IV bits	Memory
d-Monomial test	20	1	16	2^{16}
Monomial distr. test	20	2^5	13	2^{13}
Max. degree test	20	2^5	19	1

Table 10.9: Number of IV bits needed to attack the first keystream bit of Decim-v2 for different number of rounds in the initialization (out of 768 rounds).

	Rounds	P	IV bits	Memory	Fraction
d-Monomial test	160	1	12	2^{12}	$47/192$
	192	1	20	2^{20}	$18/192$
Monomial distr. test	160	2^5	17	2^{17}	$47/192$
	192	2^5	20	2^{20}	$13/192$
Max. degree test	160	2^5	12	1	$44/192$
	192	2^5	20	1	$17/192$

Table 10.10: Number of IV bits needed to attack the initial state variables of Decim-v2 for different number of rounds in the initialization (out of 768 rounds).

10.6.4 Lex

Lex [Bir06] is a software oriented stream cipher that is based on the block cipher AES. It uses 128 bit keys and IVs, the IV is taken as the initial state and to initialize the cipher one full AES round, i.e., 10 round operations, is applied to the IV before producing any keystream, then one more round is performed before the first leakage of keystream bits.

The Maximal Degree Test on Lex relates to the saturation attack on AES. This attack uses the property that the sum of one output byte for all possible inputs to one byte of the cipher is zero after three rounds of AES. According to the Reed-Muller transform, the maximal degree monomial is calculated as the sum of all entries in the truth table, this means that a degree 8 polynomial can never occur from the same byte after 3 round operations. This

also means that we can easily create a chosen IV distinguisher by considering eight IV bits that go into the same byte, the maximal degree of the output polynomial for three rounds is seven. After four rounds this property in general disappears, but the attack can be extended to attack 6 rounds of AES by using guess and determine techniques on other rounds.

	Rounds	P	IV bits	Memory
d-Monomial test	2	1	8	2^8
	3	1	18	2^{18}
Monomial distr. test	2	2^6	2	2^2
	3	2^6	8	2^8
Max. degree test	2	2^5	2	1
	3	2^5	8	1

Table 10.11: Number of IV bits needed to attack the first keystream of Lex for different number of rounds in the initialization (out of 11 rounds).

As we see, the Monomial Distribution Test and the Maximal Degree Monomial Test performs best on Lex, a slight advantage for the Maximum Degree Monomial attack because of the low memory requirement. The d-Monomial test fails to find the anomaly that the degree 8 monomial can not exist as there is only one monomial of this degree.

10.7 Summary

In this study, we generalize the idea of d-monomial attacks and propose a framework for chosen IV statistical analysis. The proposed framework can be used as an instrument for designing good initialization procedures. It can be used to verify the effectiveness of the initialization, but also to help designing a well-balanced initialization, e.g., prevent an unnecessary large number of initial clockings or even reduce the number of gates used in an hardware implementation by being able to use a simpler loading procedure.

Also, we propose two new statistical attacks. The attacks work better than the d-Monomial attack as described by Saarinen. In particular, the Maximal Degree Monomial test improves on the analysis of state variables in Grain. The attacks are applied on some eSTREAM proposals, and we give some conclusions regarding the strength of their initialization procedure.

We experimentally detected statistical weaknesses in the keystream for reduced versions Grain and Trivium where 75% respectively 64% of the initial clocking were used. In Decim we could only find statistical weaknesses when the number of initial clockings was reduced to 3%. Similiar meassurements were performed on the internal state variables. We showed that is possible to dectect weakness in 54% of the state variable of Grain-128 with full IV initialization. It is an open question how to utilize these weaknesses of state bits to attack the cipher. Decim also seemed more vulnerable to this kind of analysis, statistical deviations could be found in state bits when 25% of the number of initial clockings were used. Trivium however has a linear output function and the state variables offer a comparable restistance to this type of analysis as the keystream bits.

The tests suggest that the initialization procedure of Grain should be extended with some more initial clockings. The initialization procedure of Trivium seems to offer a good trade off between security and speed. For the ciphers Grain and Trivium, we propose alternative initialization schemes with slightly higher hardware complexity. In the proposed loadings, each IV and key bit affects more than one state bit and the resistance of the ciphers to the proposed attacks increases about 50%.

Decim seems to have a high security margin and it is probable that the number of round in the initialization procedure could be greatly reduced. It is also an interesting question whether a simpler loading procedure could be used in Decim. A simpler loading procedure could mean a smaller footprint in hardware.

11

Concluding Remarks

In this thesis we have studied distinguishing attacks on stream ciphers. The distinguishing attack is relatively young type of attack and should be further studied. It is clear that they are important since they highlight weaknesses of a cipher. We have also showed that distinguishing attacks in some cases can be turned into key recovery attacks and instance deduction.

In this thesis distinguishing attacks have been presented in different attack scenarios. We have studied the statistics of one single long keystream, but also considered distinguishing attack under resynchronization using many keystreams.

Distinguishing attacks have been described on the classical stream cipher constructions such as filter generator and the irregularly clocked filter generators. Though these constructions are not often used in practice, many primitives are closely related to these constructions. We also demonstrated how our attacks could be applied to two modern constructions.

A large part of the thesis has been spent on investigating the security of some eSTREAM candidates. Confidence in a new stream cipher design can only be gained by thorough analysis of the primitive. Hence, it is very important to study the security of these candidates before they can be widely deployed.

A

Variance of the number of combinations

Let $\mathcal{D}_t^{z_{win_1}}$ and $\mathcal{D}_t^{z_{win_2}}$ denote the number of zeros in window one respectively window two. Also, let $L_i,\ i \in \{1, 2\}$ denote the size of window i, then

$$E(\mathcal{D}_t^{z_{win_1}}) = \tfrac{r_1}{2} \quad E\big((\mathcal{D}_t^{z_{win_1}})^2\big) = \tfrac{r_1^2+r_1}{4} \quad V(\mathcal{D}_t^{z_{win_1}}) = \tfrac{r_1}{4}$$

$$E(\mathcal{D}_t^{z_{win_2}}) = \tfrac{L_2}{2} \quad E\big((\mathcal{D}_t^{z_{win_2}})^2\big) = \tfrac{r_2^2+L_2}{4} \quad V(\mathcal{D}_t^{z_{win_2}}) = \tfrac{L_2}{4}$$

Let z_t denote the bit in the center position, and W_t' the number of samples fulfilling the recurrence relation. To make the computations a bit simpler we denote $W_t = W_t' - \frac{L_1 L_2}{2}$, i.e., we subtract the expected value of W_t', hence $E(W_t) = 0$. We also introduce the symbol $A_t = \mathcal{D}_t^{z_{win_1}}\mathcal{D}_t^{z_{win_2}} + (L_1 - \mathcal{D}_t^{z_{win_1}})(L_2 - \mathcal{D}_t^{z_{win_2}}) - L_1L_2/2$.

$$W_t = \begin{cases} \underbrace{\mathcal{D}_t^{z_{win_1}}\mathcal{D}_t^{z_{win_2}} + (L_1 - \mathcal{D}_t^{z_{win_1}})(L_2 - \mathcal{D}_t^{z_{win_2}}) - L_1L_2/2}_{A_t} & \text{if } z_t = 0, \\ -\underbrace{\big(\mathcal{D}_t^{z_{win_1}}\mathcal{D}_t^{z_{win_2}} + (L_1 - \mathcal{D}_t^{z_{win_1}})(L_2 - \mathcal{D}_t^{z_{win_2}}) - L_1L_2/2\big)}_{A_t} & \text{if } z_t = 1. \end{cases}$$

We define W as the sum of W_t for N bits, $W = \sum_{t=0}^{N-1} W_t$. Hence

$$E(W) = E(\sum_{t=0}^{N-1} W_t) = \sum_{t=0}^{N-1} E(W_t) = 0.$$

We are trying to calculate

$$V(\sum_{i=0}^{N-1} W'_t) = V(\sum_{i=0}^{N-1} W_t + N\frac{L_1 L_2}{2}) = V(\sum_{i=0}^{N-1} W_t) = V(W).$$

We will use that $V(W) = E(W^2) - E(W)^2$.

$$E(W^2) = E\big((\sum_{t_1=0}^{N-1} W_{t_1})(\sum_{t_2=0}^{N-1} W_{t_2})\big) = \sum_{t_1=0}^{N-1}\sum_{t_2=0}^{N-1} E(W_{t_1} \cdot W_{t_2})$$

- For $t_1 \neq t_2$

$$\begin{aligned} E(W_{t_1}W_{t_2}) = \quad & \tfrac{1}{4}\Big(\quad E(W_{t_1}W_{t_2}|z_{t_1}=0, z_{t_2}=0)+ \\ & + \quad E(W_{t_1}W_{t_2}|z_{t_1}=0, z_{t_2}=1)+ \\ & + \quad E(W_{t_1}W_{t_2}|z_{t_1}=1, z_{t_2}=0)+ \\ & + \quad E(W_{t_1}W_{t_2}|z_{t_1}=1, z_{t_2}=1)\Big) = \\ = \quad & \tfrac{1}{4}E\big(A_{t_1}A_{t_2} - A_{t_1}A_{t_2} - A_{t_1}A_{t_2} + (-A_{t_1})(-A_{t_2})\big) = 0 \ . \end{aligned}$$

- For $t_1 = t_2$

$$\begin{aligned} E(W_t^2) = \quad & \tfrac{1}{2}\big(E(W_t^2|z_t=0) + E(W_t^2|z_t=1)\big) = E(A^2) = \\ = \quad & 4E(X^2)E(Y^2) + 4L_1L_2E(X)E(Y) - 4L_1E(X)E(Y^2) - \\ & -4L_2E(X^2)E(Y) + \tfrac{L_1^2L_2^2}{4} - L_1^2L_2E(Y) - \\ & -L_1L_2^2E(X) + L_1^2E(Y^2) + L_2^2E(X) = \\ = \quad & \tfrac{L_1L_2}{4} \end{aligned}$$

So

$$E(W^2) = \sum_{t_1=0}^{N-1}\sum_{t_2=0}^{N-1} E(W_{t_1}W_{t_2}) = \sum_{t=0}^{N-1} E(W_t^2) = \sum_{t=0}^{N-1} \frac{L_1L_2}{4} = N \cdot \frac{L_1L_2}{4}.$$

Finally we can give and expression or the variance.

$$V(W) = E(W^2) - E(W)^2 = N \cdot \frac{L_1 L_2}{4}.$$

B

Notations

α	Significance level.
$1-\alpha$	Confidence level.
$\mathcal{A}$	Symbol alphabet.
$\mathcal{A}_d$	Space of affine functions of d variables.
$\lvert\mathcal{A}\rvert$	Cardinality of a space.
$\mathcal{AR}$	Acceptance region.
$1-\beta$	Power level.
$\mathcal{B}_d$	Space of Boolean functions of d variables.
c_t	Cipher text at time t.
$\mathcal{C}$	Ciphertext space.
$d_H(h_1, h_2)$	Hamming distance between function h_1 and h_2.
$deg(h)$	Algebraic degree of Boolean function $h(\boldsymbol{x})$.
$d_K(m)$	Decryption of one ciphertext symbol, i.e., $d_K : \mathcal{A} \to \mathcal{A}$.
$D_K(\boldsymbol{m})$	Decryption function i.e., $D_K : \mathcal{C} \to \mathcal{M}$.
$\mathcal{D}_X(x)$	Probability mass function of random variable X.
$\lvert\mathcal{D}_0 - \mathcal{D}_1\rvert$	Statistical distance between $\mathcal{D}_0$ and $\mathcal{D}_1$.
$\Delta(\mathcal{D}_\mathcal{C})$	Squared Euclidean Imbalance (SEI) between $\mathcal{D}_\mathcal{C}$ and the uniform distribution.
e_t	Noise variable.
$\varepsilon,\ \epsilon$	Statistical bias.
$e_K(m)$	Encryption of one plaintext symbol, i.e., $e_K : \mathcal{A} \to \mathcal{A}$.
$E_K(\boldsymbol{m})$	Encryption function, i.e., $E_K : \mathcal{M} \to \mathcal{C}$.
$f(x)$	Polynomial (mostly used for feedback polynomials of LFSRs).

$f_X(x)$	Probability density function of random variable X.
$F_X(x)$	Distribution function of random variable X.
$\mathbb{F}_q$	Field of size q, q is a prime number.
$\mathbb{F}_q[x]$	Polynomial with coefficients in $\mathbb{F}_q$.
$\mathbb{F}_{q^n}$	Extension field of $\mathbb{F}_q$.
$\mathbb{F}_q^n$	Vector of n elements, where each element is from $\mathbb{F}_q$
$h(x_0, \ldots, x_d)$	Boolean function of d variables.
$h(\boldsymbol{x}), \boldsymbol{x} \in \mathbb{F}_2^d$	Boolean function of d variables.
IV	Initialization vector (IV).
iv_i	IV bit i .
$\mathcal{IV}$	IV space.
K	Key (asymmetric cryptography: K_e-encryption key and K_d-decryption key).
$\|K\|$	Key size, i.e., number of bits used.
$\mathcal{K}$	Key space.
k_i	Keybit i.
LLR	Log-likelihood ratio.
m_t	Message symbol at time t.
$\boldsymbol{m}$	Message.
$\mathcal{M}$	Message space.
N	Number of samples needed in attack.
$nl(h)$	Nonlinearity of Boolean function $h(\boldsymbol{x})$
P	Number of feedback polynomials needed in attack.
P_e	Overall error probability.
$\mathcal{P}_X(x)$	Distribution function of random variable X.
r_f	Degree of polynomial $f(x)$.
s_t	LFSR symbol at time t.
σ	Internal state of cipher.
T	Period.
$w_H(\boldsymbol{x})$	Hamming weight of vector $\boldsymbol{x}$.
W_f	Weight of $f(x)$.
$\mathcal{W}_h$	Walsh transform of Boolean function h.
$\boldsymbol{x} \in \mathbb{F}_2^d$	Vector of d elements, i.e., $\boldsymbol{x} = (x_0, \ldots, x_{d-1}),\ x_i \in \mathbb{F}_2$
z_t	Keystream symbol at time t.

Bibliography

[ALP04] F. Armknecht, J. Lano, and B. Preneel. On the resynchronization attack. In H. Handschuh and M.A. Hasan, editors, *Selected Areas in Cryptography—SAC 2004*, volume 3357 of *Lecture Notes in Computer Science*, pages 19–38. Springer-Verlag, 2004.

[aQUoTiA] Information Security Research Centre at Queensland University of Technology in Australia. Crypt-X. *http://www.isi.qut.edu.au/resources/cryptx/*.

[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+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.

[Ber06] D.J. Bernstein. Salsa20. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/025, 2006. available at *http://www.ecrypt.eu.org/stream*.

[Bir06] A. Biryukov. A new 128 bit key stream cipher : Lex. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/013, 2006. available at *http://www.ecrypt.eu.org/stream*.

[BJV04] T. Baignères, P. Junod, and S. Vaudenay. How Far Can We Go Beyond Linear Cryptanalysis? In *Advances in Cryptology—ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 432–450, 2004.

[BL05] A. Braeken and J. Lano. On the (im)possibility of practical and secure nonlinear filters and combiners. In *Selected Areas in Cryptography—SAC 2004*, volume 3897 of *Lecture Notes in Computer Science*, pages 159–174. Springer-Verlag, 2005.

[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.

[CDF+00] A. Clark, E. Dawson, J. Fuller, J. Golic, H-J. Lee, William Millan, S-J. Moon, and L. Simpson. The LILI-128 keystream generator. In *Selected Areas in Cryptography—SAC 2000*, volume 2012 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[CDF+02] A. Clark, E. Dawson, J. Fuller, J. Golic, H-J. Lee, W. Millan, S-J. Moon, and L. Simpson. The LILI-II keystream generator. In L. Batten and J. Seberry, editors, *Information Security and Privacy: 7th Australasian Conference, ACISP 2002*, volume 2384 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2002.

[CDF+04] A. Clark, E. Dawson, J. Fuller, J. Golic, H-J. Lee, W. Millan, S-J. Moon, and L. Simpson. LILI-II design. Available at *http://www.isrc.qut.edu.au/resource/lili/lili2design.php*, Accessed November 10, 2004, 2004.

[CGJ06] C. Cid, H. Gilbert, and T. Johansson. Cryptanalysis of Pomaranch. *IEE Proceedings - Information Security*, 153(2):51–53, June 2006.

[CHJ02] D. Coppersmith, S. Halevi, and C.S. Jutla. Cryptanalysis of stream ciphers with linear masking. In M. Yung, editor, *Advances in Cryptology—CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 515–532. Springer-Verlag, 2002.

[CHM+05] K. Chen, M. Henricksen, W. Millan, J. Fuller, L. Simpson, E. Dawson, H. Lee, and S. Moon. Dragon: A fast word based stream cipher. *ECRYPT Stream Cipher Project Report 2005/006*, 2005.

[CJS00] V. Chepyzhov, T. Johansson, and B. Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In B. Schneier, editor, *Fast Software Encryption—FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 181–195. Springer-Verlag, 2000.

[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.

[CM03] N. Courtois and WS. 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.

[Cou03] 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.

[CP05] C. De Cannière and B. Preneel. Trivium - specifications. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030, 2005. available at *http://www.ecrypt.eu.org/stream*.

[CS91] V. Chepyzhov and B. Smeets. On a fast correlation attack on certain stream ciphers. In D.W. Davies, editor, *Advances in Cryptology—EUROCRYPT'91*, volume 547 of *Lecture Notes in Computer Science*, pages 176–185. Springer-Verlag, 1991.

[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.

[DA06] J. Daemen and G.V. Assche. Distinguishing stream ciphers with convolutional filters, 2006.

[DGV94] J. Daemen, R. Govaerts, and J. Vandewalle. Resynchronization weaknesses in synchronous stream ciphers. In T. Helleseth, editor, *Advances in Cryptology—EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*, pages 159–167. Springer-Verlag, 1994.

[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.

[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—FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 2004.

[EHJ07a] 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.

[EHJ07b] H. Englund, M. Hell, and T. Johansson. Two general attacks on Pomaranch-like keystream generators. In A. Biryukov, editor, *Fast Software Encryption—FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 274–289. Springer-Verlag, 2007.

[EJ02] P. Ekdahl and T. Johansson. Distinguishing attacks on SOBER-t16 and SOBER-t32. In J. Daemen and V. Rijmen, editors, *Fast Software Encryption—FSE 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 210–224. Springer-Verlag, 2002.

[EJ04] H. Englund and T. Johansson. A new simple technique to attack filter generators and related ciphers. In H. Handschuh and A. Hasan, editors, *Selected Areas in Cryptography—SAC 2004*, volume 3357 of *Lecture Notes in Computer Science*, pages 39–53. Springer-Verlag, 2004.

[EJ05] H. Englund and T. Johansson. A new distinguisher for clock controlled stream ciphers. In *Fast Software Encryption—FSE 2005*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[EJ06] H. Englund and T. Johansson. Three ways to mount distinguishing attacks on irregularly clocked stream ciphers. In *International Journal of Security and Networks*, volume 1. Inderscience Enterprise Ltd, 2006.

[EJT07] H. Englund, T. Johansson, and M. Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In C.P. Rangan, editor, *Progress in Cryptology—INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages ??–?? Springer-Verlag, 2007.

[EM05] H. Englund and A. Maximov. Attack the Dragon. In S. Maitra, V. Madhavan, and R. Venkatesan, editors, *Progress in Cryptology—INDOCRYPT 2005*, volume 3797 of *Lecture Notes in Computer Science*, pages 130–142. Springer-Verlag, 2005.

[Fil01] E. Filiol. A new statistical testing for symmetric ciphers and hash functions. In V. Varadharajan and Y. Mu, editors, *International Conference on Information, Communications and Signal Processing*, volume 2119 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 2001.

[FL03] B.J. Falkowski and C.C. Lozano. Generation and properties of fastest transform matrices over gf(2). In *Proceedings of the 2003 International Symposium on—ISCAS'03.*, volume 4 of *Circuits and Systems*, pages 740–743, 2003.

[FMS01a] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. *Lecture Notes in Computer Science*, 2259:1–24, 2001.

[FMS01b] S.R. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. In *Selected Areas in Cryptography—SAC 2001*, pages 1–24, 2001.

[Gjø05] K. Gjøsteen. Security notions for disk encryption. In *Computer Security – ESORICS 2005*, volume 3679 of *Lecture Notes in Computer Science*, pages 455–474. Springer-Verlag, 2005.

[GM03a] J. D. Golic and G. Morgari. On the resynchronization attack. In T. Johansson, editor, *Fast Software Encryption—FSE 2003*, volume 2887 of *Lecture Notes in Computer Science*, pages 100–110. Springer-Verlag, 2003.

[GM03b] J.D. Golić and R. Menicocci. A new statistical distinguisher for the shrinking generator. Available at *http://eprint.iacr.org/2003/041*, Accessed September 29, 2003, 2003.

[GO94] J.D. Golić and L. O'Connor. A unified Markov approach to differential and linear cryptanalysis. In *Advances in Cryptology—ASIACRYPT'94*, Lecture Notes in Computer Science, pages 387–397. Springer-Verlag, 1994.

[Gol93] J.D. Golić. Correlation via linear sequential circuit approximation of combiners with memory. In R.A. Rueppel, editor, *Advances in Cryptology—EUROCRYPT'92*, volume 658 of *Lecture Notes in Computer Science*, pages 113–123. Springer-Verlag, 1993.

[Gol94] J.D. Golić. Intrinsic statistical weakness of keystream generators. In *Advances in Cryptology—ASIACRYPT'94*, volume 917 of *Lecture Notes in Computer Science*, pages 91–103. Springer-Verlag, 1994.

[Gol95] J.D. Golić. Towards fast correlation attacks on irregularly clocked shift registers. In L.C. Guillou and J-J. Quisquater, editors, *Advances in Cryptology—EUROCRYPT'95*, volume 921 of *Lecture Notes in Computer Science*, pages 248–262. Springer-Verlag, 1995.

[Gol96] J.D. Golić. Computation of low-weight parity-check polynomials. *Electronic Letters*, 32(21):1981–1982, October 1996.

[Gol97] 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.

[Hel80] M. Hellman. A cryptanalytic time-memory trade-off. In *IEEE Transactions on Information Theory*, volume 26, pages 401–406, 1980.

[HJ06] M. Hell and T. Johansson. On the problem of finding linear approximations and cryptanalysis of Pomaranch version 2. Selected Areas in Cryptography—SAC 2004, 2006. Preproceedings.

[HJMM06] M. Hell, T. Johansson, A. Maximov, and W. Meier. A stream cipher proposal: Grain-128. Information Symposium in Information Theory—ISIT 2006, 2006. available at *http://www.ecrypt.eu.org/stream*.

[HKK05] M. Hasanzadeh, S. Khazaei, and A. Kholosha. On IV setup of Pomaranch. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/082, 2005. *http://www.ecrypt.eu.org/stream*.

[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.

[JHK05] C.J.A. Jansen, T. Helleseth, and A. Kholosha. Cascade jump controlled sequence generator (CJCSG). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/022, 2005. *http://www.ecrypt.eu.org/stream*.

[JHK06a] C.J.A. Jansen, T. Helleseth, and A. Kholosha. Cascade jump controlled sequence generator and Pomaranch stream cipher (version 2). eSTREAM, ECRYPT Stream Cipher Project, Report 2006/006, 2006. *http://www.ecrypt.eu.org/stream*.

[JHK06b] C.J.A. Jansen, T. Helleseth, and A. Kholosha. Cascade jump controlled sequence generator and Pomaranch stream cipher (version 3). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2006. *http://www.ecrypt.eu.org/stream*.

[JJ99] T. Johansson and F. Jönsson. Fast correlation attacks based on turbo code techniques. In M.J. Wiener, editor, *Advances in Cryptology—CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 181–197. Springer-Verlag, 1999.

[JJ00] T. Johansson and F. Jönsson. Fast correlation attacks through reconstruction of linear polynomials. In M. Bellare, editor, *Advances in Cryptology—CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 300–315. Springer-Verlag, 2000.

[JJ02] T. Johansson and F. Jönsson. A fast correlation attack on LILI-128. In *Information Processing Letters*, volume 81, pages 127–132, 2002.

[JM06] A. Joux and F. Müller. Chosen-ciphertext attacks against MOSQUITO. In M. Robshaw, editor, *Fast Software Encryption—FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 390–404. Springer-Verlag, 2006.

[Jun03] P. Junod. On the optimality of linear, differential and sequential distinguishers. In *Advances in Cryptology—EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2003.

[Kha05] S. Khazaei. Cryptanalysis of Pomaranch (CJCSG). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/065, 2005. *http://www.ecrypt.eu.org/stream*.

[LZGB03] S. Leveiller, G. Zémor, P. Guillot, and J. Boutros. A new cryptanalytic attack for pn-generators filtered by a boolean function. In K. Nyberg and H. Heys, editors, *Selected Areas in Cryptography—SAC 2002*, volume 2595 of *Lecture Notes in Computer Science*, pages 232–249. Springer-Verlag, 2003.

[Mar] G. Marsaglia. DIEHARD statistical tests.

[Mas69] J.L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15:122–127, 1969.

[Mat94] M. Matsui. Linear cryptanalysis method for DES cipher. In T. Helleseth, editor, *Advances in Cryptology—EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer-Verlag, 1994.

[McE87] R.J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1987.

[MH04] H. Molland and T. Helleseth. An improved correlation attack against irregular clocked and filtered keystream generators. In *Advances in Cryptology—CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 373–389. Springer-Verlag, 2004.

[MJ05a] A. Maximov and T. Johansson. Fast computation of large distributions and its cryptographic applications. In *Advances in Cryptology—ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 313–332. Springer-Verlag, 2005.

[MJ05b] 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.

[Mol04] H. Molland. Improved linear consistency attack on irregular clocked keystream generators. In *Fast Software Encryption—FSE 2004*, 2004.

[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.

[MvOV97] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[NIS] NIST. NIST statistical test suite. *http://csrc.nist.gov/rng*.

[Pas03] E. Pasalic. *On Boolean Functions in Symmetric-Key Ciphers*. PhD thesis, Lund University, Department of Information Technology, P.O. Box 118, SE–221 00, Lund, Sweden, 2003.

[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*.

[Saa02] M.-J.O. Saarinen. A time-memory tradeoff attack against LILI-128. In J. Daemen and V. Rijmen, editors, *Fast Software Encryption—FSE 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 231–236. Springer-Verlag, 2002.

[Saa06] M.-J.O. Saarinen. Chosen-IV statistical attacks on eSTREAM stream ciphers. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/013, 2006. *http://www.ecrypt.eu.org/stream*.

[Sha49] C.E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 27:656–715, 1949.

[Sie84] T. Siegenthaler. Correlation-immunity of non-linear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, 30:776–780, 1984.

[TSS+05] Y. Tsunoo, T. Saito, M. Shigeri, H. Kubo, and K. Minematsu. Shorter bit sequence is enough to break stream cipher LILI-128. *IEEE Transactions on Information Theory*, 51:4312–4319, 2005.

[TWP07] E. Tews, R.P. Weinmann, and A. Pyshkin. Breaking 104 bit WEP in less than 60 seconds. Cryptology ePrint Archive, Report 2007/120, 2007. *http://eprint.iacr.org/*.

[Wag02] D. Wagner. A generalized birthday problem. In M. Yung, editor, *Advances in Cryptology—CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 2002.