Stream Cipher Design

An evaluation of the eSTREAM candidate Polar Bear

J O H N   M A T T S S O N

KTH Computer Science
and Communication

Master of Science Thesis
Stockholm, Sweden 2006

Stream Cipher Design

An evaluation of the eSTREAM candidate Polar Bear

J O H N   M A T T S S O N

Master's Thesis in Computer Science (20 credits)
at the School of Engineering Physics
Royal Institute of Technology year 2006
Supervisor at CSC was Johan Håstad
Examiner was Johan Håstad

TRITA-CSC-E 2006:111
ISRN-KTH/CSC/E--06/111--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se

Abstract

As a response to the lack of efficient and secure stream ciphers, ECRYPT (a 4-year Network of Excellence funded by the European Union) manages and coordinates a multi-year effort called eSTREAM to identify new stream ciphers suitable for widespread adoption. Polar Bear, one of the eSTREAM candidates, is a new synchronous stream cipher proposed by Johan Håstad, and Mats Näslund. In this thesis, the first known attack is presented. It is a guess-and-determine attack with a computational complexity of $O(2^{78.8})$ that recovers the initial state. We propose that this weakness is fixed by adding a key-dependent pre-mixing of the dynamic permutation in conjunction with the key schedule. Further suggested tweaks strengthen the security and improves performance on long sequences. The updated Polar Bear specification that will be sent to eSTREAM before June 30, 2006, is based on tweaks suggested in this thesis. We have also optimized the source code of Polar Bear, which enables it to run almost twice as fast. We have not found any other weaknesses in Polar Bear, and it seems resistant to all known generic attacks.

Konstruktion av strömkrypton

En utvärdering av eSTREAM-kandidaten Polar Bear

Sammanfattning

På grund av bristen på säkra och effektiva strömkrypton driver ECRYPT (ett fyra år långt EU-projekt) ett delprojekt kallat eSTREAM för att identifiera nya strömkrypton lämpliga för implementation. Polar Bear, en av kandidaterna, är ett nytt strömkrypto skapat av Johan Håstad och Mats Näslund. I denna uppsats presenterar vi den första kända attacken. Det är en *guess-and-determine attack* som bestämmer det initiala tillståndet med en tidkomplexitet på $O(2^{78.8})$. Vi föreslår att liknande attacker undviks genom att den dynamiska permutationen blandas under nyckelschemat. Ytterligare förslag förstärker säkerheten och förbättrar prestandan vid långa sekvenser. Den uppdaterade Polar Bear-specifikationen kommer att bygga på dessa förslag. Vi har också optimerat källkoden, vilket nästan fördubblar prestandan. Vi har inte hittat några andra svagheter i designen, och Polar Bear verkar stå emot alla kända typer av attacker.

Acknowledgments

I like to thank the people at the Communication Security Lab at Ericsson Research in Kista for making my time there as enjoyable as it was. Especially my supervisor Mats Näslund that always answered the questions that I had. And Eva Gustafsson and Rolf Blom for letting my go to the SASC workshop in Leuven, Belgium.

I also want to thank Johan Håstad, my supervisor and examiner at the Royal Institute of Technology.

Contents

1 Introduction **1**
1.1 Classification of Ciphers . . . 2
1.2 This Master's Project and eSTREAM . . . 4
1.3 Purpose . . . 5
1.4 Outline of the Thesis . . . 5
1.5 Definitions and Notation . . . 6

2 Stream Cipher Design **7**
2.1 Design Approaches . . . 7
2.1.1 Information-theoretic approach . . . 7
2.1.2 System-theoretic approach . . . 8
2.1.3 Complexity-theoretic approach . . . 8
2.1.4 Randomized stream ciphers . . . 8
2.2 Shift Registers . . . 8
2.2.1 Linear feedback shift register . . . 8
2.2.2 Boolean functions . . . 9
2.2.3 Nonlinear feedback shift register . . . 10
2.2.4 The 2-adic integers . . . 10
2.2.5 Feedback shift register with carry . . . 10
2.3 Stream Ciphers Based on LFSRs . . . 11
2.3.1 Nonlinear combination generators . . . 11
2.3.2 Nonlinear filter generators . . . 12
2.3.3 Clock-controlled generators . . . 13
2.4 Stream Ciphers Based on Block Ciphers . . . 13
2.4.1 Cipher feedback mode . . . 13
2.4.2 Output feedback mode . . . 13
2.4.3 Counter mode . . . 13

3 Basic Characteristics **15**
3.1 Statistical Properties . . . 15
3.1.1 Period . . . 16
3.1.2 Golomb's randomness postulates . . . 16
3.1.3 Statistical tests . . . 16

3.2 Measures of Complexity . . . 17
3.2.1 Linear complexity . . . 18
3.2.2 Quadratic span . . . 18
3.2.3 Maximum order complexity . . . 19
3.2.4 2-adic span . . . 19
3.2.5 Ziv-lempel complexity . . . 19
3.3 Key, State and IV size . . . 20

4 Generic Attacks on Stream Ciphers **21**
4.1 Exhaustive Key Search . . . 22
4.2 Time Memory Trade-offs . . . 22
4.3 Distinguishing Attacks . . . 23
4.4 Guess-and-Determine attacks . . . 24
4.5 Correlation Attacks . . . 25
4.5.1 Basic correlation attack . . . 25
4.5.2 Fast correlation attack . . . 25
4.6 Algebraic Attacks . . . 27
4.7 Side channel attacks . . . 28
4.8 Implementation Issues . . . 29

5 Description of Polar Bear **30**
5.1 Rijndael . . . 30
5.1.1 Attacks . . . 30
5.2 RC4 . . . 31
5.2.1 Attacks . . . 31
5.3 Polar Bear . . . 32
5.3.1 Description . . . 32
5.3.2 The output cycle . . . 33

6 Results **35**
6.1 Permutation Error . . . 35
6.2 A Guess-and-Determine Attack . . . 36
6.2.1 An improved attack . . . 37
6.2.2 Analysis . . . 38
6.3 Evaluation with Respect to Other Attacks . . . 39
6.3.1 Time-memory trade-offs . . . 39
6.3.2 Algebraic attacks . . . 39
6.3.3 Correlation attacks . . . 39
6.4 A new version of Polar Bear - Pig Bear . . . 39
6.4.1 Key schedule . . . 40
6.4.2 The output cycle . . . 40
6.4.3 Security analysis . . . 41
6.5 Statistical Testing . . . 42
6.6 Optimization and Performance . . . 42

6.6.1 Further performance tweaks 44

7 Conclusions **46**

7.1 Future Work 47

Bibliography **48**

List of Figures

1.1 *Initiation and output cycle of a synchronous stream cipher* 3
1.2 *Additive stream cipher* 4

2.1 *Feedback shift register with carry* 11
2.2 *Nonlinear combination generator and nonlinear filter generator* 12

5.1 *Key and IV schedule* 33
5.2 *The output cycle* 34

List of Tables

1.1 *The eSTREAM timetable* 5

6.1 *Performance figures* 43
6.2 *Performance figures for the speed tweaks* 44

Chapter 1

Introduction

The need to keep information secret, especially in communications, is obvious in many circumstances such as military, diplomatic, and business affairs. One early example found in The Code Book by Simon Singh [50], is the story about Histiaeus that, in 499 BC was planning a revolt in Ionia. His solution to the problem of secrecy was that he shaved the head of his messenger, tattooed a message on his head, and then waited for the hair to grow back before he sent him away. Such secret communication, achieved by hiding the existence of a message, is known as *steganography*. The word is derived from the Greek words *steganos*, meaning “covered,” and *graphein*, meaning “to write” [50]. This way of secret communication has its limits as the message is revealed if it is found. Hence, in parallel with the development of steganography, there was the evolution of *cryptography*, derived from the Greek word *kryptos*, meaning “hidden”. The aim of cryptography is not to hide the existence of a message, but rather to hide its meaning [50].

A specific method to hide the meaning of a message is called a *cipher*, and the process of transforming the message or *plaintext* to a coded message, or *ciphertext*, is known as *encryption*. The reverse process of transforming the ciphertext back to the original plaintext is known as *decryption*. A cipher can be seen as a combination of a general cryptographic *algorithm* and a *key* that decides the encryption details in the specific case [50]. If we compare with ordinary locks, all the locks of a specific type works in the same way, but all the keys are different. The key in modern ciphers is often a sequence of bits (zeros and ones). As the general algorithm is often publicly known, the confidentiality of the message depends on the secrecy of the key (Kerckoff’s law). One of the most famous ciphers in modern time is Enigma, that was used by Nazi Germany in the WWII. The breaking of Enigma by British intelligence affected the outcome of the war. Today when so much information is transmitted over open channels like the Internet and wireless channels (GSM, UMTS, Wi-Fi, WiMAX etc.), cryptography is more important than ever.

As old as the need of secrecy is the urge to read others encrypted messages. *Cryptanalysis* is the science of recovering information without knowledge of the key. The term *cryptology* is sometimes used for the area of cryptography and cryptanalysis together.

The scientific study of cryptology started around WWII with a pioneering paper [47] written by Shannon. Cryptology uses ideas from several other fields such as information theory, computer science, number theory, and abstract algebra. Cryptologic research has historically been done by governments and kept secret. Only the last decades there has been a widespread open research in cryptology. But the number of researchers involved in and the money spent on secret research still far exceed that in open research. Traditionally even publicly used cryptosystems like the GSM cryptosystems A5/1 and A5/2 have been kept secret, but this has changed recently. The international standards DES (Data Encryption Standard) and AES (Advances Encryption Standard) used by governments and financial institutions are publicly known.

1.1 Classification of Ciphers

Cryptosystems can either be *secret key* and *symmetric* (AES, DES, RC4), or *public key* and *asymmetric* (ElGamal, McEliece, RSA). In a symmetric system the sender and receiver have agreed on a secret key, that is used for both encryption and decryption. In an asymmetric cryptosystem the sender uses the receiver's publicly available *public key* to encrypt, and the receiver can then decrypt with his *private key*. The idea of public key cryptography was proposed as recently as 1976 by Diffie and Hellman [18]. The first public key cryptosystem was RSA, which was proposed in 1977 by Rivest, Shamir and Adleman [40]. The letters RSA are the initials of their surnames. Government Communications Headquarters (GCHQ), a part of the British intelligence, later revealed that some of its researchers discovered public key cryptography earlier. They have also released previously secret papers to provide evidence for their claims. But the credit for the discovery must remain with the researchers who first published their work in the open scientific literature. In secret key cryptography a public key cryptosystem is often used to distribute the secret key.

Symmetric cryptosystems is usually divided into block ciphers and stream ciphers. Rueppel [42] describes the difference as:

> *Block ciphers* operate with a fixed transformation on large block of plain-text data; *stream ciphers* operate with a time-varying transformation on individual plain-text digits.

This classification is not absolute, and any block cipher can be used as a stream cipher by using certain modes of operation (see Section 2.4).

The *state* of a stream cipher can informally be defined as the values of the set of variables that describes the current status of the cipher. For each new state, the cipher outputs some bits, and then jumps to the next state where the process is repeated. The output digits z_i of the cipher are called the keystream, and the ciphertext c_i, $i = 0, 1, \ldots$ is a function of the keystream and the plaintext.

Stream ciphers are further classified as being either *synchronous* or *self-synchronizing*. In a synchronous cipher, the keystream depends only on the key and the position i, but is independent of the plaintext and the ciphertext. In a self-synchronous cipher, the

keystream depends on the key and a fixed amount of previous ciphertext, but is independent of the position i. Both designs have their respective advantages and disadvantages.

In a synchronous stream cipher the sender and receiver have to be synchronized. If synchronization is lost because a ciphertext digit is lost in the transmission, the decryption fails and the cipher has to be re-synchronized. A synchronous stream cipher has no error propagation as a modified ciphertext digit does not affect the decryption of any other ciphertext digits.

As the decryption in a self-synchronizing stream cipher is independent of the position and only depends on a fixed amount of ciphertext, the cipher is able to automatically re-establish decryption when a ciphertext digit is lost. Suppose that the state depends on the t preceding ciphertext digits. If a single ciphertext digit is lost, inserted or modified it will affect the decryption of t other ciphertext digits. A self-synchronizing steam cipher is therefore said to have a limited error propagation.

Self-synchronizing stream ciphers seem to be vulnerable to chosen-ciphertext attacks where the attacker has access to a decryption unit (see Chapter 4). The most common way of constructing self-synchronizing stream ciphers is to use a block cipher in some mode (see Section 2.4).

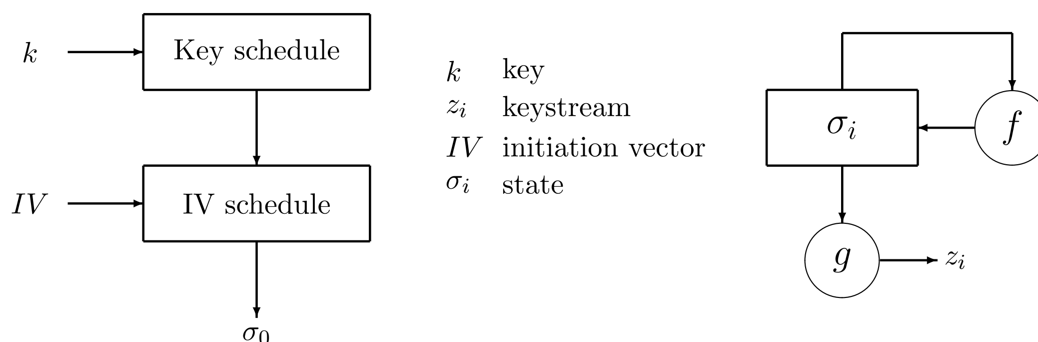

Figure 1.1: *Initiation and output cycle of a synchronous stream cipher*

The output cycle of a synchronous stream cipher can be described by the equations

$$\begin{aligned} \sigma_{i+1} &= f(\sigma_i), \\ z_i &= g(\sigma_i), \\ c_i &= h(z_i, m_i) \end{aligned}$$

where σ_0 is the *initial state* and may be determined from the key k and the initiation vector IV, f is the *next-state function*, and g is the *output function*. The schematic can be seen in Figure 1.1.

A *(binary) additive stream cipher* is a synchronous cipher where the ciphertext is obtained by combining the keystream and the plaintext using the exclusive or (XOR) operation (see Figure 1.2). Almost all currently suggested stream ciphers are additive, and these are the main topic of this thesis. In many common uses of ciphers, the transmission is not error free. In such cases frequent reinitialization is required due to

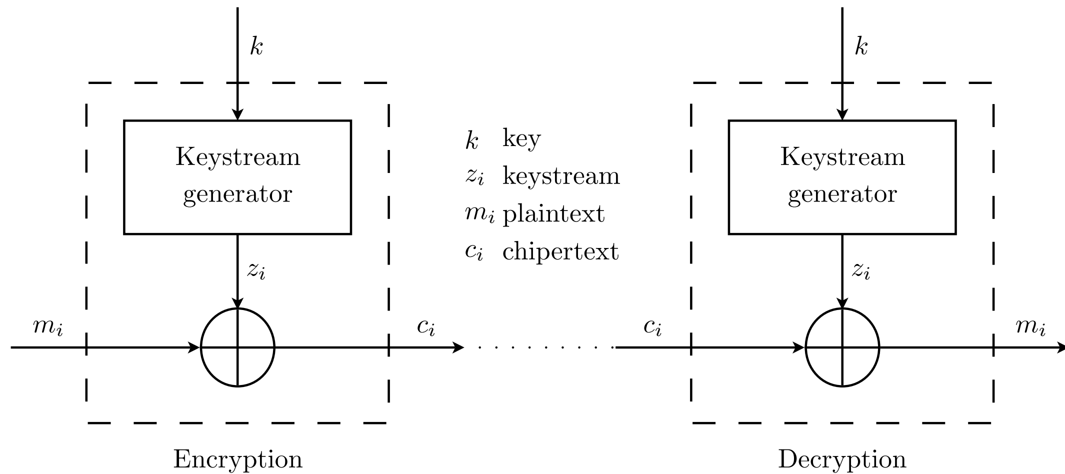

Figure 1.2: *Additive stream cipher*

the need of resynchronization. To share a new secret key is slow, so reinitialization is usually done by sharing a non-secret initialization vector IV. The initialization can be seen in Figure 1.1. A stream cipher intended for such use should describe how such a reinitialization should be done as bad choices can give extremely insecure ciphers. For an example, see Section 4.8.

As stated in [44], there are several benefits of stream ciphers compared to block ciphers:

- Stream ciphers are generally much faster than block ciphers.
- The keystream can be sometimes be generated prior to encryption/decryption. (in the synchronous case)
- No or limited error propagation
- Low hardware complexity

1.2 This Master's Project and eSTREAM

There are a variety of efficient and trusted block ciphers. Unfortunately the same is not true for stream ciphers. It is likely that the though requirements listed above as benefits, too often have had negative consequences on the security. As a response to the lack of efficient ciphers, ECRYPT (a 4-year Network of Excellence funded by the European Union) manages and coordinates a multi-year effort called eSTREAM to identify new stream ciphers suitable for widespread adoption. The timetable for eSTREAM [1] can be found in Table 1.1. This master's project began in September 2005 and ended in May 2006. The initial call for Primitives resulted in 34 candidates. The stream cipher Polar Bear, that is specifically studied in this thesis is one of these candidates. It was created by Johan Håstad and Mats Näslund, and claimed to be suitable for both profile

Table 1.1: *The eSTREAM timetable*

Date	Event
November 2004	Call for Primitives
April 2005	The beginning of the first evaluation phase of eSTREAM
March 2006	The end of the first evaluation phase of eSTREAM
July 2006	The beginning of the second evaluation phase of eSTREAM
End of 2006	Second classification
September 2007	The end of the second evaluation phase of eSTREAM
January 2008	The final report of the eSTREAM

I (software) and profile II (hardware). In this thesis we present the first known attack on Polar Bear. Our attack is a so called guess-and-determine attack with a computational complexity of $O(2^{79})$. This kind of attack is not practical as is would take a billion years on a standard PC. It is however of theoretical interest as is indicates a major weakness in the design. As Polar Bear uses a key size of 128 bit, no attack should have a computational complexity lower than $O(2^{128})$, see Section 4.1. Recently a similar attack with the improved complexity of $O(2^{57.4})$ has been presented by Hasanzadeh et al. [31]. We analyze why these attacks are possible, and suggest how the cipher can be tweaked to avoid this type of attack. We have also optimized the source code of Polar Bear, which enables it to run almost twice as fast. We have not found any other weaknesses in the design.

1.3 Purpose

This master's project aims to evaluate the security and performance of the stream cipher Polar Bear. The four main goals were:

- Evaluation of Polar Bear's security
- Evaluation with respect to statistical tests
- Deliver a highly optimized implementation written i C
- If necessary, suggest enhancements and tweaks

1.4 Outline of the Thesis

The remaining parts of this thesis are organized as follows. In Chapter 2 an introduction to design principles and building blocks of stream ciphers are given. In Chapter 3 certain characteristics such as period, statistical properties, and complexities are discussed. Chapter 4 is a general introduction to attacks on stream ciphers. Chapter 5 describes the Stream Cipher Polar Bear and also briefly RC4 and AES. Chapter 6 presents the main results of this thesis, the attack on Polar Bear, optimization, analysis, and tweaks.

Parts of this chapter have previously been published in [35]. The results are summarized and discussed in chapter 7.

1.5 Definitions and Notation

With a few exceptions, most of the notation is standard. The reader is assumed to be familiar with basic probability and statistics and basic discrete mathematics, such as finite fields, sets and set operations, modular arithmetic, and primes. This may be found in e.g. [7].

The reader is also assumed to know basic complexity theory, such as the notations $O(f(n))$ and $\Theta(f(n))$ for algorithm complexity. This may be found in e.g. [13].

Hexadecimal values are indicated with the subscripted suffix $_{\mathrm{HEX}}$. We use $||$ to denote concatenation, and $\oplus$ denotes bitwise modulo 2 sum. So for instance $70_{\mathrm{HEX}}||\mathrm{A9}_{\mathrm{HEX}} = \mathrm{70A9}_{\mathrm{HEX}}$, and $\mathrm{70A9}_{\mathrm{HEX}} \oplus \mathrm{CA73}_{\mathrm{HEX}} = \mathrm{BADA}_{\mathrm{HEX}}$.

We will write $\mathbb{F}_q$ for the unique (up to isomorphism) finite field with $q = p^n$ elements, where p is prime.

Whenever we say "AES" or "Rijndael", we refer to [17], and not the AES specification [3], since we need the support for 256-bit blocks.

We use the notation $d_{\mathrm{H}}(\mathbf{x}, \mathbf{y})$ to denote the Hamming distance between two strings of equal length. The Hamming distance is the number of positions for which the corresponding symbols are different. We use $\mathrm{wt}(\mathbf{x})$ for the Hamming weight of a string. The weight can be defined as the distance to the string consisting of only zeros. For example $d_{\mathrm{H}}(1011101, 1001001) = 2$ and $\mathrm{wt}(1011101) = d_{\mathrm{H}}(1011101, 0000000) = 5$.

Chapter 2

Stream Cipher Design

The two main goals when designing a practical cipher system are security and performance. Given a fixed level of security the goal is to optimize performance. Performance can be measured as speed, chip area, or power consumption. Other important features is clarity of the design and flexibility in its implementation. It seems like it is difficult to design both a fast and secure stream cipher, as most proposed stream ciphers have documented weaknesses.

When designing a stream cipher there is also a trade-off between the speed in software and hardware, as it is difficult to optimize for both at the same time. The A5/1 and A5/2 stream ciphers that are used in the GSM standard uses linear feedback shift registers (LFSR) over the finite field $\mathbb{F}_2$ (see Section 2.2.1). Consequently they are fast in hardware, but slow in software. Other ciphers like SNOW [19] uses a LFSR over the finite field $\mathbb{F}_{2^{32}}$, and are very fast in software on 32-bit processors. Henceforth, we will write $\mathbb{F}_q$ for the unique finite field with $q = p^n$ elements (p is prime). Another cipher designed for good performance in software is SEAL [41].

2.1 Design Approaches

Rueppel distinguishes four principal approaches to the construction of stream ciphers [42]. The approaches differ in their assumption of the cryptanalyst's capacity and in definition of security and cryptanalytic success. Some are more theoretical than practical, and focus on provable security, which is often not possible for practically used ciphers.

2.1.1 Information-theoretic approach

In this approach the cryptanalyst is assumed to have both unlimited time and computing power. Cryptanalysis is to determine the message or key given only the ciphertext and the distributions of the plaintext and key. A cipher system is *perfectly secure* or *unconditionally secure* if the plaintext and ciphertext are statistically independent. An example of a perfectly secure cipher is the *one-time pad* or *Vernam cipher* [47]. A cipher is *ideally secure* if there is no unique solution for the plaintext given unlimited amount of

ciphertext. This does not mean that the system is secure, as the set of possible plaintext may be very small.

2.1.2 System-theoretic approach

This is the method used for designing most practically used cryptosystem including AES, DES, and RC4. Cryptanalysis of a new cipher should be a difficult problem, and the cipher should have provable properties such as period, linear complexity, etc. These properties are discussed in Chapter 3. It should also be able to withstand all known cryptanalytic attacks like linear and differential cryptanalysis, correlation attacks, and algebraic attacks. None of these attack should be faster than an exhaustive key search where all the possible keys are tried, one after another (see Chapter 4).

It seems like this approach leads to cryptanalysis resistant ciphers with smaller keys than other approaches. This is the approach and methodology that will mainly be discussed in this report.

2.1.3 Complexity-theoretic approach

The goal of the complexity-theoretic approach is not to make the plaintext and ciphertext statistically independent, but to make the plaintext computationally inaccessible. Cryptanalysis is to distinguish the keystream from a random sequence. The attacker is assumed to be limited to polynomial time attacks. A cipher is *perfect* if it is indistinguishable by all polynomial-time attacks. As no such ciphers are known, the proposed generators are often proven to be perfect under the assumption that a "famous" problem, such as discrete log, quadratic residue, or inverting RSA is hard. Rueppel [42] writes that it can be discussed how much the asymptotic analysis proves, since all implementation are finite. But in practice the theoretical results can be used to get estimations and bounds. An example of this is the fixed parameter analysis in the BMGL paper [24].

2.1.4 Randomized stream ciphers

This approach focuses on the size of the cryptanalytic problem instead of the work effort. The main idea is to use a large random string for encryption and decryption. This string is publicly known and accessible by any attacker. The key specifies which part of the string to use, whereas the attacker is forced to search through all of the random data. The security is measured as the average amount of bits a cryptanalyst has to examine.

It is possible to prove lower bounds on the security for this type of cipher systems. But the ciphers are not practically usable as they are far to slow.

2.2 Shift Registers

2.2.1 Linear feedback shift register

A *linear feedback shift register* (LFSR) of length n over the finite field $\mathbb{F}_q$ consists of n stages $[s_{n-1}, s_{n-2}, \ldots, s_0]$ with $s_i \in \mathbb{F}_q$, and a polynomial $p(x) = 1+c_1x+c_2x^2+\ldots+c_nx^n$

over $\mathbb{F}_q$. The contents of $[s_{n-1}, s_{n-2}, \ldots, s_0]$ is called the *state* of the LFSR, and $p(x)$ is called the *feedback* or *connection polynomial.* The register is controlled by a clock, and at each stepping the elements are moved to the right so that $s_i = s_{i+1}$ for $i = 0 \ldots n-2$ and s_0 is outputted. The contents of s_{n-1} is calculated according to

$$s_{n-1} = c_1 s_{n-1} + c_2 s_{n-2} + \ldots c_n s_0.$$

A polynomial $r(x)$ over $\mathbb{F}_q$ is called a primitive polynomial iff it generates all the elements of an extension field of $\mathbb{F}_q$. All primitive polynomials are irreducible (can not be factored into polynomials of lower degree). An irreducible polynomial $r(x)$ of degree n over $\mathbb{F}_p$, p prime, is primitive if the smallest positive integer m such that $r(x)$ divides $x^m - 1$ is $m = p^n - 1$. If the feedback polynomial $p(x)$ is a primitive polynomial of degree n each of the non-zero initial states produces an output sequence with the maximum possible period $q^n - 1$. Such a LFSR is called a *maximum-length* LFSR. The output sequence of a maximum-length LFSR is called a *m-sequence.*

2.2.2 Boolean functions

The elements of the finite field $\mathbb{F}_2$ are denoted 0 and 1. Addition and multiplication in $\mathbb{F}_2$ corresponds to the Boolean operations XOR and AND. Variables that range over $\mathbb{F}_2$ are called *Boolean variables* or bits. A vector whose coordinates are bits is called a *Boolean vector.* A *Boolean function* $f(\mathbf{x})$ is a function from a Boolean vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ to a single bit y.

$$f : \mathbb{F}_2^n \to \mathbb{F}_2 : \mathbf{x} \to y = f(\mathbf{x})$$

Every Boolean function $f(\mathbf{x})$ can be uniquely written as a sum over all distinct products of its variables $x_1, x_2, \ldots, x_n$. As $x_i^2 = x_i$, the number of distinct products equals the number of subsets of $\{x_1, x_2, \ldots, x_n\}$. This is called the *algebraic normal form.* There exists unique constants $a_0, a_1, \ldots, a_n, a_{12}, \ldots, a_{12\ldots n} \in \mathbb{F}_2$ such that

$$f(\mathbf{x}) = a_0 \oplus a_1 x_1 \oplus \ldots \oplus a_n x_n \oplus a_{12} x_1 x_2 \oplus \ldots \oplus a_{12\ldots n} x_1 x_2 \ldots x_n. \qquad (2.1)$$

The *nonlinear order* or *algebraic degree* of a Boolean function is the maximum order of the terms in the algebraic normal form (2.1). A Boolean function $f(\boldsymbol{x})$ is *balanced* if

$$P(f(\mathbf{x}) = 0) = P(f(\mathbf{x}) = 1) = 0.5$$

when $\mathbf{x}$ is chosen uniformly in $\mathbb{F}_2^n$. A Boolean function $f(\mathbf{x})$ is k^{th}-order correlation immune if $y = f(\mathbf{x})$ is statistically independent of any k-subset $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ of the input variables. A Boolean function that is both balanced and k^{th}-order correlation immune is called a *k-resilient function.*

Example Take for example the Boolean function used in the Geffe generator [22]. The algebraic normal form is

$$f(x_1, x_2, x_3) = x_1 x_2 \oplus x_2 x_3 \oplus x_3.$$

The nonlinear order is 2. It is balanced but only 0th-order correlation immune because $P(f(\mathbf{x}) = 0|x_1 = 0) = 0.75$.

Siegenthaler showed that there is a trade-off between the nonlinear order k and the correlation immunity m [48]. A m-th order correlation immune function can have at most nonlinear order $n - m$.

2.2.3 Nonlinear feedback shift register

A *feedback shift register* (FSR) is a generalization of a LFSR where the feedback function is any Boolean function f of n variables. It works in the same way as a LFSR, but the contents of s_{n-1} is calculated according to

$$s_{n-1} = f(s_{n-1}, s_{n-2}, \ldots, s_0).$$

If f is a linear function, the FSR is an LFSR, and otherwise it is called a *nonlinear* FSR. If every output sequence is periodic the FSR is called *non-singular*. If every output sequence has the maximum period 2^n the FSR is called a *de Bruijn FSR* and the sequence a *de Bruijn sequence*.

2.2.4 The 2-adic integers

Just as the real numbers $\mathbb{R}$ are the completion of the rationals $\mathbb{Q}$ with respect to the Euclidean metric $d(x, y) = |x - y|$, the 2-adic numbers $\mathbb{Q}_2$ are the completion of $\mathbb{Q}$ with respect to the 2-adic metric $d(x, y) = |x - y|_2$. The *2-adic integers* $\mathbf{Z}_2 \subset \mathbb{Q}_2$ is the set of all formal power series $\alpha = \sum_{i=0}^{\infty} a_i 2^i$ with $a_i \in \{0, 1\}$. The coefficients α_i is called the *2-adic digits* of α. Addition in $\mathbf{Z}_2$ is defined by "carrying" overflow bits to next term, so that $2^i + 2^i = 2^{i+1}$. Multiplication is defined by shift and addition in the natural way. Under these operations $\mathbf{Z}_2$ is a ring with the additive identity 0 and the multiplicative identity $1 = 1 \cdot 2^0$. The nonnegative integers can be represented in a natural way by finite sequences in $\mathbf{Z}_2$. But a surprising fact is that $\mathbf{Z}_2$ also contains the negative integers. For example is -1 represented by the infinite sequence $1 + 2^1 + 2^2 + 2^3 \ldots$ which can be seen by adding 1 to each of them. Any other negative integer can be represented by $-q = -1 \cdot q$. The 2-adic integers also contains numbers which are not integers, so $\mathbb{Z} \subset \mathbf{Z}_2$.

2.2.5 Feedback shift register with carry

The *feedback shift register with carry* (FCSR) (see Figure 2.1) was introduced by Klapper and Goresky [29]. It can be seen as a LFSR with memory. Every FCSR is associated with an odd positive integer $q \in \mathbb{Z}$ called the connection integer. The feedback is given by the 2-adic digits of $q + 1$.

$$q + 1 = q_1 2 + q_2 2^2 + \ldots + q_r 2^r$$

The shift register uses $r = \lfloor \log(q+1) \rfloor$ stages and a maximum of $\lfloor \log(r) \rfloor$ additional bits of memory. The state of the register is given by the r bits $[a_{r-1}, a_{r-2}, \ldots, a_0]$ and the memory integer m. In each stepping, the following operations are done.

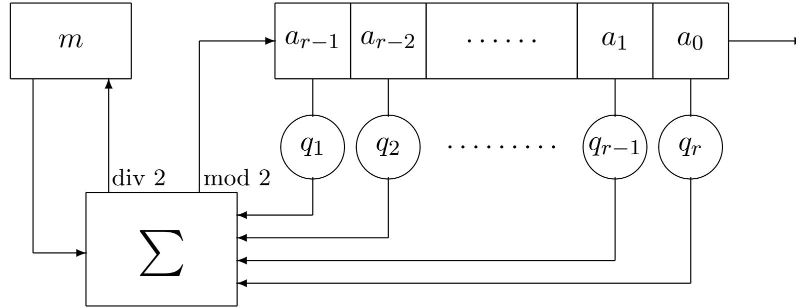

Figure 2.1: *Feedback shift register with carry*

1. Form the integer sum $\sigma = \sum_1^r q_k a_{r-k} + m$
2. Shift the register to the right and output a_0
3. Let $a_{r-1} = \sigma \pmod 2$
4. Let $m = \lfloor \sigma/2 \rfloor$

If 2 is a primitive root to the connection integer q, the output sequence is called an ℓ-sequence and has good statistical properties equal to that of an m-sequence. The period of such a sequence is $q-1$. There are two degenerate initial loadings: $m = a_i = 0$, and $a_i = 1$ and $m = \mathrm{wt}(q-1) - 1$ where wt is the Hamming weight. FCSRs can be analyzed using the algebra over the ring of 2-adic integers.

The LFSR, FSR, FCSR and d-FCSR were generalized by Xu and Klapper [30] to what they call a *Algebraic Shift Register* (AFSR), that is defined over any ring with a principal prime ideal.

2.3 Stream Ciphers Based on LFSRs

The vast majority of modern stream ciphers use one or several linear feedback shift registers as building blocks. The main reasons are that they have large period, good statistical properties, are conveniently analyzed using the algebra over finite fields, and can be implemented fast in both hardware and software. As the output from a LFSR is easy to predict in a known-plaintext attack, the linear properties must be destroyed. There are three main methodologies for doing so. It should be noted that all proposals based only on these methodologies have been broken.

2.3.1 Nonlinear combination generators

A *nonlinear combination generator* uses several maximum-length LFSRs (see Figure 2.2). The keystream is generated as a nonlinear Boolean function f of the outputs of these LFSRs. The function f is called the *combining function*.

If n maximum-length LFSRs with lengths $\ell_1, \ell_2, \ldots, \ell_n$ are used together with the boolean function f, the linear complexity (see Section 3.2.1 for a definition) of the keystream is

$$f(\ell_1, \ell_2, \ldots, \ell_n) = a_0 + a_1\ell_1 + \ldots + a_n\ell_n + a_{12}\ell_1\ell_2 + \ldots + a_{12\ldots n}\ell_1\ell_2 \ldots \ell_n$$

where $a_0, a_1, \ldots$ are the coefficients in the algebraic normal form of f, and the expression is evaluated over the ordinary integers instead of the finite field $\mathbb{F}_2$. Thus, it is desirable to use a combining function with a high nonlinear order. It is also desirable that f has high correlation immunity to withstand correlation attacks. The trade-off between high linear complexity and high correlation immunity can be avoided by permitting f to have *memory* as in the summation generator. Let the memory at time t be m_t. In each step the *integer* sum of the output bits $x_1, x_2, \ldots, x_n$ and the memory m_t is calculated. The least significant bit is outputted and the remaining bit form the new memory m_{t+1}. Unfortunately, this makes the 2-adic span of the sequence low. A fact that was used to attack the summation generator [29]. An examples of a nonlinear combination generators is the broken Geffe generator.

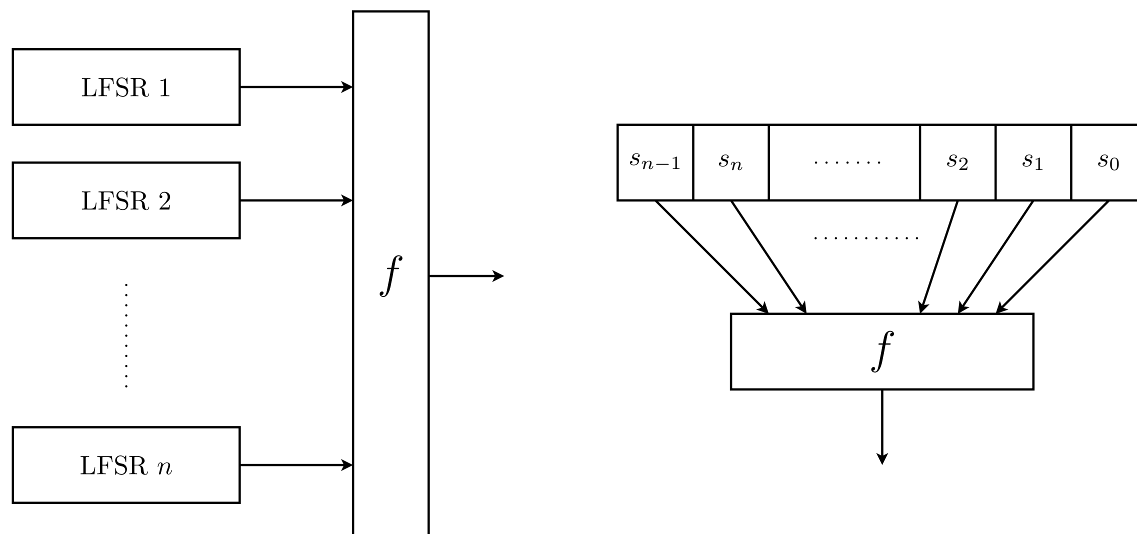

Figure 2.2: *Nonlinear combination generator and nonlinear filter generator*

2.3.2 Nonlinear filter generators

A *nonlinear filter generator* uses a single maximum-length LFSR, and the keystream is generated as a nonlinear function f of the stages of the LFSR (see Figure 2.2). The function f is called the *filtering function.* If the LFSR has length n and f has nonlinear order m, the linear complexity is at most $L_m = \sum_{i=1}^{m} \binom{n}{i}$.

If n is a prime, the probability that the sequence will have linear complexity L_m is

$$P_m \approx e^{-L_m/n \cdot 2^n} > e^{-1/n}.$$

So for large n, the probability is close to one. An example of a nonlinear filter generators is the unsecure knapsack generator.

2.3.3 Clock-controlled generators

A *Clock-controlled generator* consists of at least one LFSR, which is irregularly clocked by some other part of the cipher. In the alternating step generator, the output of one LFSR decide which one of the other two LFSRs that should be clocked. In the shrinking generator, two LFSRs are regularly clocked. If the output of the first LFSR is 1, the output of the second forms part of the keystream.

2.4 Stream Ciphers Based on Block Ciphers

The three following modes of operation for block ciphers are all part of the international standard ISO/IEC 10116:2006. The different modes of operation turns a block cipher into a stream cipher. They can be proven secure under the assumption that the block cipher is secure. In the descriptions, C_i is a ciphertext block, P_i is a plaintext block, and E_k is the encryption function of the block cipher.

2.4.1 Cipher feedback mode

The *cipher feedback mode* (CFB) turns a block cipher into a self-synchronizing stream cipher. The combining function is XOR, and the next block of keystream is determined by encrypting the last block of ciphertext.

$$\begin{aligned} C_i &= P_i \oplus E_k(C_{i-1}) \\ C_{-1} &= \mathrm{IV} \end{aligned}$$

2.4.2 Output feedback mode

The *output feedback mode* (OFB) is structurally similar to the CFB mode, but as the resulting stream cipher is synchronous instead of self-synchronizing, the resulting cipher is vastly different. Instead of encrypting the last block of ciphertext, the last block of keystream is encrypted.

$$\begin{aligned} C_i &= P_i \oplus Z_i \\ Z_i &= E_k(Z_{i-1}) \\ Z_{-1} &= \mathrm{IV} \end{aligned}$$

2.4.3 Counter mode

The *Counter Mode* (CTR) is newer than the other modes. The keystream is obtained by encrypting a block consisting of the IV concatenated with a counter. The counter can be any function $f(i)$ that does not repeat for a long time. The simplest and most popular

choice is an actual counter $f(i) = i$. CTR mode allows a random access property for decryption.

$$\begin{aligned} C_i &= P_i \oplus E_k(B_i) \\ B_i &= IV || f(i) \end{aligned}$$

Chapter 3

Basic Characteristics

A *pseudo-random bit generator* (PRBG) is a deterministic algorithm, that given a random binary string of length k outputs a binary string of length $\ell \ggg k$, that appears to be random. Every good synchronous stream cipher is therefore a PRBG. The output string can not be truly random, as there are only 2^k possible output strings and 2^ℓ strings of length ℓ. For ciphers it is utterly important that it is hard to guess the next bit given all the proceeding bits. If no polynomial-time algorithm exist that is able to guess the next-bit with a probability significantly greater than 0.5, the PRBG is called a *cryptographically secure* PRBG. Some applications of PRBGs, like simulations, do not seem to rely on this property, but it is difficult to find a weaker mathematical definition with useful provable properties.

Bernoulli distribution The *Bernoulli distribution* is a discrete probability distribution with probability function

$$P(n) = \begin{cases} 1-p & \text{for } n = 0 \\ p & \text{for } n = 1 \end{cases}$$

Random sequence A random bit sequence $\boldsymbol{r} = \{r_i\}_{i=0}^{\infty}$ is a sequence such that every bit r_i follows a Bernoulli distribution with $p = 0.5$, and all the bits are independent of each other. The goal of every stream cipher is to be computationally indistinguishable from such a sequence.

3.1 Statistical Properties

A random sequence has many properties with a high probability. For example, the relative frequencies of zeros and ones are likely to be almost equal for long sequences. Such properties can be used to distinguish a pseudo-random sequence from a truly random.

3.1.1 Period

Having a period is clearly a statistical defect, that distinguish a sequence from a random one. A cipher with a too small period is obviously easy to predict. The period must be large enough to ensure that is is never repeated. It is usually done by using a building block that can be proven to have a large period, for instance a maximum-length LFSR.

3.1.2 Golomb's randomness postulates

Golomb's randomness postulates are highly structured conditions that a sequence should fulfill to appear random. These conditions are *far* from being sufficient. Let $\boldsymbol{s} = \{s_i\}_{i=0}^{\infty}$ be a binary sequence with period N. A run of $\boldsymbol{s}$ is a subsequence consisting of only 0's or 1's, and neither preceded nor succeeded by the same symbol. The *autocorrelation* of $\boldsymbol{s}$ is defined as

$$C(\tau) = \sum_{i=0}^{N-1} (-1)^{s_i + s_{i+\tau}}.$$

The sequence $\boldsymbol{s}$ satisfies Golomb's postulates if

- The number of 1's in every period differ from the number of 0's by at most one.
- In every period, at least half of the runs must have length 1, at least one-fourth length 2, etc., as long as the number of runs so indicated exceeds one. Moreover, for each of these lengths, there must be (almost) equally many runs of 0's and 1's.
- $\boldsymbol{s}$ is a 2-level autocorrelation sequence. That is

$$C(\tau) = \begin{cases} N & \text{if} \quad \tau \equiv 0 \pmod N \\ -1 & \text{otherwise} \end{cases}$$

A binary sequence that satisfies Golomb's postulates is called a *pseudo-noise sequence* or a *pn-sequence*. Examples of *pn*-sequences are m-sequences and ℓ-sequences, that both appears as output sequences of shift registers, see Section 2.2.1 and 2.2.5.

3.1.3 Statistical tests

There is a large number of different statistical tests used for evaluating PRBGs. All tests evaluate if a bit sequence has some property, that a random sequence is likely to have. Because random sequences do not have nontrivial statistical properties with a probability equal to one, the test only determine if it is likely that the tested sequence is random or not.

The significance level α, is the probability of rejecting the hypothesis that the sequence is a random bit sequences, even though the hypothesis is true. In practice a significance level between 0.001 and 0.05 is used.

Examples of useful statistical tests are the frequency test, serial test, poker test, runs test, and autocorrelation test.[1] Another interesting test is Maurer's universal statistical

[1] For a detailed explanation of these tests, see [37].

test, that is able to detect a very general class of defects including the ones detectable by the five tests above. The idea behind the test is that it should not be possible to significantly compress a random sequence. FIPS 140-2 is a NIST standard, that specifies a set of tests and significance levels, that a cipher should satisfy to qualify for a specific security rating (1–4, from lowest to highest). All the tests above are *empirical* and not theoretical in the sense that samples of sequences are generated from which certain statistics are evaluated.

Passing a fixed set of statistical tests is a *necessary*, but not *sufficient* condition for a stream cipher to be secure. Passing a test only makes it unlikely that the keystream has a specific defect.

For cryptographic purposes it is important that any sequence obtained by concatenating bits from several keystream sequences also is indistinguishable from a random sequence. Therefore, it is important to test that the nth bit is Bernoulli distributed with $p = 0.5$. It is done by generating a sequence consisting of the nth bit from a large number of sequences (perhaps with related keys).

Test suites

Several test suites with batteries of statistical test aimed at making it easy to test PRNGs exist. All of the following test suites are freely available for download.

- *Marsaglia's Diehard Battery of Tests* is the oldest and most well known test suite. It contains 15 statistical tests. [33]
- *NIST Statistical Test Suite* contains 16 statistical tests and is developed and maintained by the U.S. National Institute of Standards and Technology. They provide guidance in the use and application of the tests. [2]
- *DieHarder* is a test suite by Robert G. Brown, that aims to include test from Diehard, the NIST STS, and several other test in a single package. It fixes several problems with the Diehard package, and uses the GNU Scientific Library interface. The test suite is still under active development. [8]

3.2 Measures of Complexity

The different complexities all try to measure how hard a sequence is to produce. For a complexity measure to be practically useful for cryptographic purposes, two requirements have to be fulfilled.

- An efficient algorithm to calculate the complexity has to be known.
- The distribution of the complexity for random sequences has to be known

If these requirements are fulfilled, the complexity measure can be used as a statistical test. Complexity measures are often more interesting in cryptology than other statistical

tests, as some of them give methods to recreate the sequence using building blocks commonly used in stream ciphers. For the linear complexity, the distribution for a random sequence has been exactly calculated. It has been approximated for the maximum order complexity. Some complexity measures are only of theoretical interest because there is no efficient algorithm to calculate them.

3.2.1 Linear complexity

The *linear complexity* or *linear span* of a binary sequence $\boldsymbol{s}^n = \{s_i\}_{i=0}^{n-1}$ is the smallest nonnegative integer L such that there exists a linear recursion with fixed constants $c_1, c_2, \ldots, c_L \quad c_i \in \{0,1\}$ satisfying

$$s_j + c_1 s_{j-1} + \ldots + c_L s_{j-L} = 0 \quad L \leq j < n.$$

This is also the length of the smallest LFSR that can be used to duplicate the sequence.

Berlekamp-Massey algorithm

The Berlekamp-Massey algorithm [34] is an efficient algorithm to find the linear complexity of a binary sequence of length n. The iterative algorithm finds one or all of the shortest LFSR capable of generating the sequence. The running time of the algorithm is $O(n^2)$.

Let $\boldsymbol{s} = \{s_i\}_{i=0}^{\infty}$ be a binary sequence with linear complexity L. Then the Berlekamp-Massey algorithm finds a LFSR of length L which generates $\boldsymbol{s}$, given a subsequence of length $2L$.

The original Berlekamp-Massey algorithm works for sequences over the binary field $\mathbb{F}_2$, but can be generalized to an arbitrary finite field $\mathbb{F}_q$.

Linear complexity profile

Given a random bit sequence $\boldsymbol{r}^n = \{r_i\}_{i=0}^{n-1}$, with n moderately large, it was proved by [37] that the expectation and variance of the linear complexity $L(\boldsymbol{r}^n)$ is

$$E\big(L(\boldsymbol{r}^n)\big) \approx \frac{n}{2} \text{ and } V\big(L(\boldsymbol{r}^n)\big) \approx \frac{86}{81}.$$

This can be used as a statistical test. The linear complexity profile of a stream cipher should be indistinguishable from the linear complexity profile of a random sequence.

3.2.2 Quadratic span

The *quadratic span* $Q(\boldsymbol{s}^n)$ is the length of the shortest quadratic recurrence that generates the sequence $\boldsymbol{s}^n$. It is stated in [53] that for a moderately large n, the expectation of the quadratic span of a random sequence of length n is

$$E\big(Q(\boldsymbol{r}^n)\big) \approx \sqrt{2n}.$$

This measure is only of theoretical interest as no efficient algorithm to determine the quadratic span of a sequence is known, and the statistics of random sequences is unknown.

3.2.3 Maximum order complexity

As with the quadratic span, no efficient algorithm has been found for complexities with any fixed maximum order larger than one. However for recursions without a fixed maximum order, efficient algorithms exists. The *maximum order complexity* or *maximum order span* of a sequence $\boldsymbol{s}$ is the length M of the shortest feedback shift register that can generate that sequence. This can be efficiently computed in linear time. For a given sequence $\boldsymbol{s}^n = \{s_i\}_{i=0}^{n-1}$ it is obvious that

$$M(\boldsymbol{s}^n) \leq Q(\boldsymbol{s}^n) \leq L(\boldsymbol{s}^n).$$

Approximate values for the mean and variation were found by [20]. For a random bit sequence $\boldsymbol{r}^n = \{r_i\}_{i=0}^{n-1}$, the expected maximum order complexity is

$$E\big(M(\boldsymbol{r}^n)\big) \approx 2\lg n.$$

3.2.4 2-adic span

In analogy with the definition of linear complexity, the *2-adic span* is intended to measure how large FCSR is required to generate a sequence. It was proposed by Klapper et al. in 1997 [29]. Let λ be the number of bits in the register and memory of a FCSR. The *2-adic span* $\lambda_2(\boldsymbol{s})$ of an eventually period binary sequence $\boldsymbol{s} = \{s_i\}_{i=0}^{\infty}$ is the smallest value of λ, which occurs among all FCSRs whose output is the sequence $\boldsymbol{s}$. They also found an efficient algorithm to calculate the 2-adic span of a given sequence. It needs only $2M + 2\lg(M)$ bits, where M is the 2-adic span of the sequence. Like the linear complexity profile, the 2-adic complexity profile for a random binary sequence of length n grows approximately as $n/2$.

The 2-adic span was used to attack [29] the summation cipher, that gives sequences with large linear complexity, but much smaller 2-adic complexity. A typical example is a linear complexity of 2^{79}, but a 2-adic span of 2^{18}.

3.2.5 Ziv-lempel complexity

The *Ziv-lempel complexity* is another measure of complexity. It has it roots in the field of data compression and quantifies the rate at which new patterns appear in a sequence. It measures how much the sequence can be compressed with the Ziv-lempel compression algorithm. The measure is of practical interest, and is used in the NESSIE statistical toolbox.

3.3 Key, State and IV size

It is obvious that a cipher should never have a state smaller than the size of the key. Otherwise it would be faster to search through the internal states than the keys. This is however not enough. By using various *time memory trade-offs* (TMTO) attacks, the attack time can be severely cut at the expense of memory. Babbage [5] suggest the principle:

> If a secret key length of k bits is required, a state size of at least $2k$ bits i desirable.

Hong et al. [27] discovered another generic TMTO attack, and proposes the following design principle:

> The entropy of the key and the IV should always be at least twice the key size.

It implies that the IV size should be at least as big as the key size. For an explanation of the attacks behind these recommendations, see Section 4.2.

Chapter 4

Generic Attacks on Stream Ciphers

The methods for attacking a stream cipher can be classified according to the information available to the cryptanalyst, the aim of the attack, or the way the attack is done. It is often assumed that the attacker has knowledge of the cryptographic algorithm, but not the key. To attack an unknown cipher is a much harder problem, and it is the reason why all military ciphers are kept secret. History has however showed that it is diffucult to keep the cryptographic function secret.

Jönsson [28] lists four different categories according to the information available to the attacker.

1. *Ciphertext-only* The worst case for the cryptanalysist. Given only the ciphertext the attacker tries to recover the key or plaintext. The plaintext must have redundancies for such an attack to be successful.

2. *Known-plaintext* The attacker knows the ciphertext and all or part of the plaintext. For additive stream ciphers this is equivalent of knowing all or part of the keystream. The aim is to deduce the key or more plaintext.

3. *Chosen-plaintext* The attacker has access to an encryption unit and can encrypt any chosen plaintexts. The aim is to deduce the key.

4. *Chosen-ciphertext* The attacker has access a decryption unit and can decrypt any chosen ciphertext. The aim is to deduce the key.

This can also be seen as an ordering of how realistic the different kinds of attacks are. Because ciphertext is transmitted in public, it is easy to access. To get access to a corresponding plaintext is more difficult. It is possible that the attacker guesses some parts of the message, for example names or numbers that are likely to appear. In several implementations, the first encrypted characters are known or partly known. For example, most file types have a fixed header in the beginning of the file. If the attacker can get someone to encrypt the messages, the chosen-plaintext attack can be realistic.

The chosen-ciphertext attack is less realistic, but a good cipher should be secure against even this kind of attack.

For a synchronous stream cipher, category 2–4 above are equivalent. This is not true for a self-synchronizing stream cipher, where chosen-ciphertext attack often are most effective. Other categories are *chosen-IV attacks*, were the attacker can chose initiation vectors, and *related key attacks* were the attacker can chose a specific relation between the keys.

The Nessie report [44] classifies attacks depending on the aim of the attack. The assumption is that the plaintext is known.

1. *Key recovery* A method to recover the key.
2. *Prediction* A method for predicting a bit or sequence of bits of the keystream with a probability better than guessing.
3. *Distinguishing* A statistical method to distinguish the keystream from a random sequence.

To recover the key is obviously the most powerful of the three as it enables both prediction and distinguishing. Prediction enables distinguishing, but a distinguishing can also be thought of as a kind of prediction.

An important result connecting statistical properties and prediction was found by Yao [52]. Yao proved that a pseudo-random generator can be efficiently predicted iff the generator can be efficiently distinguished from a perfectly random source. This is however only true in the black and white world where all polynomial time algorithms are runnable and all non polynomial algorithms are not.

4.1 Exhaustive Key Search

Exhaustive Key Search is a trivial way to recover the key. It can be used against any stream cipher. Given a keystream the attacker tries all different keys until the right one is found. If the key is n bits, the attacker has to try 2^n keys in the worst case and 2^{n-1} keys on average. The computational complexity of an attack is often stated like $O(2^{79})$, which should be read as $c \cdot 2^{79}$ where c is some 'small' constant. An attack with a higher computational complexity than an exhaustive key search is not considered an attack at all.

The time complexity of an exhaustive key search can however be considerably lower with a so called *time-memory trade-off* (TMTO) technique (see Section 4.2).

4.2 Time Memory Trade-offs

A *time memory trade-off* (TMTO) attack is an attack were large amounts of precomputed data is used to lower the computational complexity. The name comes from the fact that there is often a trade-off between the amount of memory used for data storage and the amount of time used for computations.

The first TMTO attack on stream ciphers is due to Babbage [5], and was used by Golić to break the A5 cipher used in GSM standard [23]. The Babbage-Golić attack is a birthday attack. Assume that we have a stream cipher with a key size of k bits and an state size of s bits. Suppose we generate keystream for 2^m different states and store them in a table. We then observe 2^d different keystreams. By the birthday paradox, we will on average be able to break one of these keystreams when

$$m + d = s, \quad \text{with the special case} \quad m = d = \frac{s}{2}.$$

So if the state size is smaller than twice the key size, these complexities will be lower than an exhaustive key search.

Hong et al. [27] was the first to realize that trade-offs can work on key/IV pair instead of the state. The birthday attack below is taken from [11]. We have a stream cipher with a key size of k bits and an IV size of v bits. Suppose we generate keystream for 2^m different key/IV pairs and store them in a table. We then observe 2^d different keystreams. By the birthday paradox, we will be able to break one of these keystreams when

$$m + d = k + v, \quad \text{with the special case} \quad m = d = \frac{k+v}{2}.$$

So if the IV size is smaller than the key size, these complexities will be lower than an exhaustive key search. In this case we can also break all the other keystreams that uses the found key.

To avoid the known TMTO attacks for stream ciphers, the state size should be at least twice the key size, and the IV size should be at least as large as the key size. Otherwise the security level of the cipher can never be as large as the key size.

4.3 Distinguishing Attacks

A *distinguishing attack* is a method for distinguishing the keystream from a truly random sequence. Distinguishing attacks can be generic in the sense that they are likely to work on a category of ciphers, or very specific and targeted at a specific cipher. A typical specific distinguishing attack uses the fact that some part of the keystream, with a high probability, is a function of some other parts of the keystream.

$$Z_i = f(Z_{i-1}, Z_{i-1}, \ldots, Z_{i-n})$$

If a cipher fails any of the ordinary statistical tests, this can be used to distinguish the keystream. But as the generic statistical tests, see Section 3.1.3, were designed to evaluate randomness properties of PRNGs, they seldom find weaknesses in new stream cipher proposals. As a result, they are only used for catching implementation errors. Several new tests have been developed specifically toward stream ciphers [43] [51]. They concentrate on the correlation between key, IV, and keystream. Saarinen describe a chosen-IV distinguishing attack that is able to distinguish 6 of the 35 eSTREAM candidates. The attack can be summarized as

1. Choose n bits $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ in the IV as variables . The rest of the IV and the key are given a fixed value.
2. Find the boolean function f from $\mathbf{x}$ to a single keystream bit (typically, the first).
3. Check if the ANF (Algebraic Normal Form) expression of the Boolean function has the expected number of d-degree monomials. A monomial is a product of positive integer powers of a fixed sets of variables, for example, x_1, x_1x_3, or $x_2x_3x_7$.

Distinguishing attacks often require large amounts of keystream. An easy way to get away from such attacks is to state that the cipher must be rekeyed after a certain amount of keystream. For example the authors of the cipher SNOW 2.0 states that the cipher must be rekeyed after at most 2^{50} words [19]. For block ciphers in OFB or counter mode, there exists generic distinguishing attacks [26]. As the encryption function in a block ciher is a permutation, the ciphertext blocks in these modes will all be different. For a random funtion on n bit blocks, there is likely to be a match when the number of blocks m equals $\sqrt{2^n}$.

4.4 Guess-and-Determine attacks

In a *Guess-and-Determine* attack, the strategy is to guess a few of the unknown variables in the cipher, and from the guessed values deduce the values of other unknown variables. This is often not possible to do faster than exhaustive key search due to nonlinearities and irregularities in the cipher. Because of this, an assumption is made that makes the cipher more linear. If the probability that the assumption holds is p, the expected number of tries until the assumption holds are $1/p$. After the unknown values are deduced, the cipher is iterated, producing output that can be compared with the actual output. The three steps of the attack can be summarized as

1. Guess some parts of the key or state of the cipher.
2. Determine other parts of the key/state under some assumption. The assumption is that the key/IV pair is of some subset of the total set that makes the cipher weak.
3. By calculating keystream from the deduced values and compare with the known keystream we can check if the guess is right and the assumption holds.

The attack is successful if $2^g \cdot (1/p) \cdot w < 2^k$, where g is the number of guessed bits, p is the probability that the assumption holds, w is the work needed to determine if the guess is right and the assumption holds, and k is the key size. The work involved in each step w is often very small compared to the other values and ignored.

In Section 6.2, a guess-and-determine attack on Polar Bear is given.

4.5 Correlation Attacks

Correlation attacks is probably the most important class of general attacks on stream ciphers, and efficient correlation attacks have been found for many stream ciphers. For a correlation attack to be applicable, the keystream $z_1, z_2, \ldots$ must be correlated with the output sequence $a_1, a_2 \ldots$ of a much simpler internal device, such as a LFSR. The two sequences are correlated if the probability $P(z_i = a_i) \neq 0.5$. If this is the case, it might be possible to recover the initial state of the target LFSR.

4.5.1 Basic correlation attack

The first correlation attack was proposed by Siegenthaler [49]. Siegenthaler's attack is aimed at a nonlinear combination generator (see Section 2.3.1), and was ciphertext only, but the description here is a known-plaintext attack as described in [28]. If there is a correlation between the keystream sequence $\boldsymbol{z}$ and the output sequence $\mathbf{a}$ from one of the internal LFSRs, the attacker can mount a *divide-and-conquer* attack and determine the initial states of the LFSRs, one by one until the whole key is known.

Assume that we have observed the keystream $\mathbf{z}$ from a nonlinear combination generator with n LFSRs, and that there is a correlation $P(z_i = a_i) = 1-p > 0.5$ with the output sequence $\mathbf{a}$ of a LFSR of length ℓ. For each possible initial state $\mathbf{u}_0 = (u_1, u_2, \ldots, u_\ell)$, the output sequence $\mathbf{a}$ is generated.

Define $\beta = N - d_H(\mathbf{a}, \mathbf{z})$, where $d_H(\mathbf{a}, \mathbf{z})$ is the Hamming distance between $\mathbf{u}$ and $\mathbf{z}$, and N is the length of $\mathbf{u}$ and $\mathbf{z}$. If we run through all 2^ℓ possible initial states, and if N is large enough, β will with high probability take its largest value when $\mathbf{u}_0$ is the correct initial state.

If a correlation can be found for each of the LFSRs, the computational complexity of recover the key is reduced from $\prod_{i=1}^{n}(2^{\ell_i}-1)$ for exhaustive key search, to $\sum_{i=1}^{n}(2^{\ell_i}-1)$ where ℓ_i is the length of LFSR i. The attack is applicable when the length of the shift registers are small, and when the combining function leaks information about individual input variables.

An standard example of a bad combining function vulnerable to correlation attacks is the one used in the Geffe generator.

$$f(x_1, x_2, x_3) = x_3 \oplus x_1x_2 \oplus x_2x_3$$

It is a poor choice because

$$P(f(X) = X_1) = P(f(X) = X_3) = \frac{3}{4}.$$

The definition of a n$^{\text{th}}$ order correlation immune combining function (see Section 2.2.2) was proposed after discovery of this attack.

4.5.2 Fast correlation attack

Meier and Staffelbach presented a modification called *fast correlation attacks* [36]. The description below is mainly from [28]. These attacks are, when applicable, significantly

faster than exhaustive search over the target LFSR, but requires received sequences of large length. Instead of using exhaustive search, the attacks use certain *parity check equations* that are created from the feedback polynomial. The attacks have two phases. In the first, a set of parity check equations are found. In the second these equations are used in a decoding algorithm to recover the transmitted codeword (the internal output sequence).

Parity check equations can be created in the following way. Suppose that the feedback polynomial $g(x)$ has t non-zero coefficients ($g(x)$ has weight t).

$$g(x) = 1 + c_1 x + c_2 x^2 + \ldots + c_\ell x^\ell$$

From this we get t different parity check equations for the digit a_i. And by noting that for a polynomial over $\mathbb{F}_2$

$$g(x)^{2^k} = 1 + c_1 x^{2^k} + c_2 x^{2^{k+1}} + \ldots + c_\ell x^{\ell 2^k}$$

we get t more equations for each squaring of the polynomial $g(x)$. This can easily be generalized to polynomials over any finite field by taking the polynomial to the power of the characteristic of the field, instead of squaring. The obtained equations are valid for all indexes i. The total number of check equations that can be obtained by squaring the feedback polynomial is

$$m \approx t \log(\frac{N}{2\ell})$$

where N is the length of the sequences $\mathbf{u}, \mathbf{z}$. The m parity check equations can be written as

$$\begin{aligned} u_i + b_1 &= 0 \\ u_i + b_2 &= 0 \\ &\vdots \\ u_i + b_m &= 0 \end{aligned}$$

where b_j is the sum of $t-1$ different digits in the sequence $\mathbf{u}$, not including u_i. If we substitute u_i with z_i we get the following expressions.

$$\begin{aligned} z_i + y_1 &= L_1 \\ z_i + y_2 &= L_2 \\ &\vdots \\ z_i + y_m &= L_m \end{aligned}$$

where y_j is the sum of $t-1$ different digits in the sequence $\mathbf{z}$, not including z_i. By counting the number of equations that hold ($L_i = 0$) we can calculate the probability

$$p^* = P(z_i = u_i | h \text{ equations hold}).$$

Two different algorithms were suggested. One in which p^* is calculated for each observed symbol and the l positions with highest value of p^* are used to find the correct initial state, and one iterative algorithm.

The algorithm works when the weight t is small, but for LFSRs with many taps the algorithm fails, as the number of required equations is to large. The correlation probability p, that the algorithm can handle is much lower for polynomials with many taps. As a consequence, feedback polynomials of large weight should be used.

This algorithm has been improved in several ways. There are other methods of finding parity equations of low weight using matrix representation of the LFSR, polynomial division, and linear codes.

4.6 Algebraic Attacks

Where correlation attacks tries to find a linear approximation of the output function f, an *algebraic attack* uses expressions of higher degree. The first algebraic attacks were called *high order correlation attack* [16], and used low order approximations. Purely algebraic attacks where the equations are not approximations were made possible by efficient ways to lower the degree of the equations [14]. Such attacks are called *fast* algebraic attacks.

Algebraic attacks consists of four steps:

1. Find a system of equations in keystream bits z_i, and the unknown initial state $\mathbf{s}$.
2. Reduce the degree of the equations.
3. Insert the observed keystream bits z_i.
4. Recover the state by solving the system of equations.

Algebraic attacks are more time-consuming than correlation attacks, but requires less keystream. Algebraic attacks are the fastest known attacks on several ciphers and they have been used to break Toyocrypt, E0 (used in Bluetooth), and a modified version of SNOW.

Finding equations

For a pure combiner (one without memory), the keystream is a function of the input variables $z_i = f(x_1^i, x_2^i, \ldots, x_n^i)$. But x_j^i is typically a linear function of the initial state $\mathbf{s}$ so $\mathbf{x^i} = L^i(\mathbf{s})$ where L^i is a linear function in matrix form applied i times. For each observed keystream symbol we get an equation

$$\begin{aligned} z_1 &= f(L^1(\mathbf{s})) \\ z_2 &= f(L^2(\mathbf{s})) \\ &\vdots \end{aligned}$$

This system of equations can be solved with the algorithms described below. For combiners with memory this strategy do not work directly. The equations look like $z_i = f(L^i(\mathbf{s}), \mathbf{m_i})$, where $\mathbf{m_i}$ is the m memory bits at time i. These equations have too many unknowns, and inserting $\mathbf{m_i} = g(\mathbf{m_{i-1}})$ gives equations with a degree that increases exponentially with i. Armknecht et al. [4] solved this by showing that for a combiner with m memory bits, it is always possible to find a boolean function H of degree at most $\lceil s(m+1)/2 \rceil$ such that $H(f(L^i(\mathbf{s}), z_i, \ldots, z_{i+r-1})) = 0$ and $r > m$. They also described a systematic way of finding such a function. This requires larger amounts of keystream, and is only practical when the number of memory bits m is small.

Equation Solving

The problem of solving systems of multivariate polynomial equations over any finite field is NP-complete. When the number of equations m is the same as the number of unknowns n, the best known algorithm for small fields is exhaustive search. For vastly overdetermined systems, somewhat faster algorithms exists. Most of these algorithms is based on linearization. The basic idea is that we linearize by replacing each monomial with a new variable. The system is then solved as a linear system. Take for example the system

$$\begin{aligned} x + y + z &= 0 \\ xyz + xy + z &= 0 \\ y + xyz &= 0 \end{aligned}$$

Substituting $u = xyz$ and $v = xy$ gives

$$\begin{aligned} x + y + z &= 0 \\ u + v + z &= 0 \\ y + u &= 0 \end{aligned}$$

Some of the suggested algorithms include XL (eXtended Linearization) and XSL (eXtended Sparse Linearization). In XL, each equation is multiplied by all monomials of some bounded degree. The expanded system is then linearized. In XSL, the monomials are more carefully selected.

Another option is to use algorithms involving Gröbner bases. However, it is difficult to predict the complexity of these algorithms, and therefore attacks involving them.

4.7 Side channel attacks

Side channel attacks differ greatly from the other attacks, as they use information from the physical implementation instead of theoretic weaknesses. Any information that can be measured, and is dependent on the key, state, or plaintext can potentially be used in a side channel attack.

In a *timing attack* the attacker tries to break a cipher by analyzing the execution time for encryption or decryption. It can be done if the encryption or decryption time depends on the input. This is usually the case for asymmetric algorithms. An example is the common algorithm for modular exponentiation (usually used in RSA implementations), where execution time is proportional to the Hamming weight of the input. In 2003 Boneh and Brumley [9] presented a practical timing attack on SSL-enabled web servers, that recovered the private key in hours. The vulnerability used was an RSA-implementation with an optimized Chinese remainder theorem.

Power analysis is similar to a timing attack, but the attacker studies the power consumption of a cryptographic device (smart card, CPU).

Other information that has been suggested includes leaked electromagnetic radiation and sound. Shamir and Tromer showed that it might be possible to conduct a timing attack based only on the humming noise of the CPU [46].

To circumvent side channel attacks, blinding should be used. Blinding means that the execution time and power consumption are made independent of the inputs. This is often possible, but the downside is that it often leads to a less efficient cipher and higher development costs.

4.8 Implementation Issues

Even though a stream cipher does not have any known theoretical flaws, the implementation of the cipher can make the cipher weak and vulnerable to attacks. Historically cryptosystems have often been broken because of a bad implementation.

The most important factor when using a synchronous stream cipher is to never use the same key/IV twice. By taking the XOR of the two ciphertexts $\mathbf{c}^1$ and $\mathbf{c}^2$, an attacker get the XOR of the plaintexts $\mathbf{m}^1 \oplus \mathbf{m}^2$, which may be enough to recover both of the plaintexts.

To claim that a cipher has a security equal to its key size is wrong if the keys does not come from a uniform distribution. If the distribution is highly non-uniform, the expected time for an exhaustive-search can be much smaller.

One of the most well known examples of bad implementations is the use of RC4 in the WEP standard used in Wi-Fi (see Section 5.2.1).

Chapter 5

Description of Polar Bear

Polar Bear reuses parts of the block cipher Rijndael, and the dynamically changing table from the stream cipher RC4. These ciphers are also included as benchmarks in the eSTREAM testing framework. The two ciphers are here presented briefly.

5.1 Rijndael

The Rijndael block cipher was selected as the Advanced Encryption Standard (AES) in October 2000. It is intended to be used by U.S. Governments, but has like DES become a global standard for software and hardware encryption. The difference between the original Rijndael proposal and the AES standard is that Rijndael allows the block length and the key length to be any multiple of 32 bits in the range 128–256 bits, whereas the AES standard fixes the block length to 128 bits and allows key lengths of 128, 192 and 256 bits only.

Rijndael was designed to be simple, fast, and resist all known attacks, including linear and differential attacks. It is a *key-iterated* block cipher, and consists of the repeated application of a round transformation of the state. The key is enlarged to an expanded key, and a part of the expanded key is added to the state before and after each round transformation.

For a block length of 128 bits, two rounds of Rijndael provides 'full diffusion', in the sense that every state bit depends on all state bits two rounds ago. Change in one state bit is likely to affect half of the state bits after two rounds. For block lengths larger than 128 bits, three round are needed. For a detailed description of the design of Rijndael, see [17].

5.1.1 Attacks

As of 2006, the only successful attacks against AES have been side channel attacks. On AES with reduced rounds there exist several attacks. Examples of such attacks are Truncated differentials attack and Boomerang attacks. Attacks faster than exhaustive key search exists for AES-128 with 7 rounds, AES-192 with 8 rounds, and AES-256 with

9 rounds. This should be compared with the 10,12, and 14 rounds used in full round AES. None of these attacks seems to be able to attack the full round AES, and they require an extremely large number of chosen plaintexts.

In 2002, Courtois et al. claimed that AES could be broken with an algebraic attack using the XSL algorithm [15]. The equation system obtained from Rijndael is over defined, sparse, and very structured. Courtois writes that AES seem over designed in respect to linear and differential cryptanalysis, and that it is an extremely bad cipher from the point of algebraic attacks. There has been much debate over this claim and the effectivity and reliability of the XSL algorithm. Several problems have been found in the underlying mathematics of the XSL algorithm, and the ECRYPT AES Security Report [12] states that “AES cannot currently be considered vulnerable to such attacks”. They also conclude that “There are still no discernible cryptographic weaknesses in the AES.”

Another indication of the security of AES is that in 2003, NSA extended their support for AES. As the first publicly available cipher it was approved for the security levels SECRET and TOP-SECRET (192 or 256 bit keys).

5.2 RC4

RC4 is probably the currently most used stream cipher. It is used in the SSL/TLS standard for secure communication between web browsers and servers, and in the WEP protocol used in 802.11 wireless LAN. It was designed by Ron Rivest in 1987 for RSA Security. The algorithm was long held as a trade secret, but in 1994 the source code was anonymously leaked to the Cypherpunks mailing list.

The inner state of RC4 consists of a dynamically changing table S, and two byte variables. A key of variable length k (8–2048 bits) is used to permute S, which initially contains the values $0 \ldots 255$ in ascending order.

$j = 0$
For $i = 0$ to 255
    $j = (j + S[i] + \text{key}[i \mod k]) \mod 256$
    Swap($S[i]$, $S[j]$)

Output is then generated depending on the state, and in each step some elements in S are permuted. The huge state size of $\log(256!) \approx 1684$ bits seems to rule out linear cryptanalysis. Analysis by Robshaw shows that the period with very high probability exceeds 10^{100}.

5.2.1 Attacks

Somewhat surprisingly for such a widely known and analyzed cipher, Mantin and Shamir found a trivial distinguishing attack as late as 2001 [32]. The first few hundred output bytes are non-random and leak information about the key. Especially the first few bytes are highly biased and the second output byte of RC4 takes on the value 0 with probability 2^{-7} instead of the expected 2^{-8}.

The reason for these weaknesses, is that the table S does not have a uniform distribution after the initial permutation. Mossel et al. [39] showed that for a table of size n, shuffling by semi-random transpositions have a computation complexity of $\Theta(n \log n)$ to get the table uniformly permuted. In the paper [38], Mironov shows that the only pass used in RC4 is clearly insufficient.

This weakness can be used to recover the key in the WEP protocol of the IEEE 802.11b standard. By analyzing only a small amount of keystream from a a few sessions, it is possible to recover the key instantly. The reason is that in WEP, the session key is created by concatenating the secret key and the IV. This turns out to be extremely weak. The SSL standard is not affected as it uses a hash function to combine the secret key with the IV.

If the first thousand bytes are discarded, such practical attacks can be avoided. But even then RC4 are not theoretically secure as there exists a distinguishing attack requiring 2^{33} bytes of keystream [21].

5.3 Polar Bear

The new stream cipher Polar Bear is one of 35 candidates submitted to eSTREAM. It was created by Johan Håstad and Mats Näslund, and claimed to be suitable for both profile I (software) and profile II (hardware). Polar Bear is the main subject of this thesis. For a more detailed description of Polar Bear, including variations, see [25].

5.3.1 Description

The cipher uses one 7-word (112-bit) LFSR R^0 and one 9-word (144-bit) LFSR R^1. These are viewed as acting over $\mathbb{F}_{2^{16}}$, which is represented as

$$\mathbb{F}_2[y]/(y^{16} + y^8 + y^7 + y^5 + 1).$$

Besides these registers, the internal state of the cipher also depends on a word quantity, S, and a dynamic permutation of bytes, D_8.

The cipher is primarily designed for a key length of 128 bits. The IV can be any number of bytes up to a maximum of 31. The key schedule is (in the case of 128-bit keys) identical to the Rijndael key schedule.

On each message to be processed, the cipher is initialized by taking the key (more precisely, the expanded Rijndael key), interpreting the IV as a plaintext block, and applying a (slightly modified) five round Rijndael encryption with block length 256. The resulting cipher text block is loaded into R^0 and R^1. Finally, D_8 is initialized to equal the table T_8, the Rijndael S-box, and S is set to zero (see Figure 5.1).

Output is produced 4 bytes at a time. To this end, the two LFSRs are first irregularly clocked, determined by S. Eight bytes, selected from R^0 and R^1, are run through the permutation D_8 to produce the four output bytes. Selected entries in D_8 are swapped. Finally, S and R^0 are modified in preparation for the next output cycle. Entries in R^1 are not modified apart from the LFSR stepping.

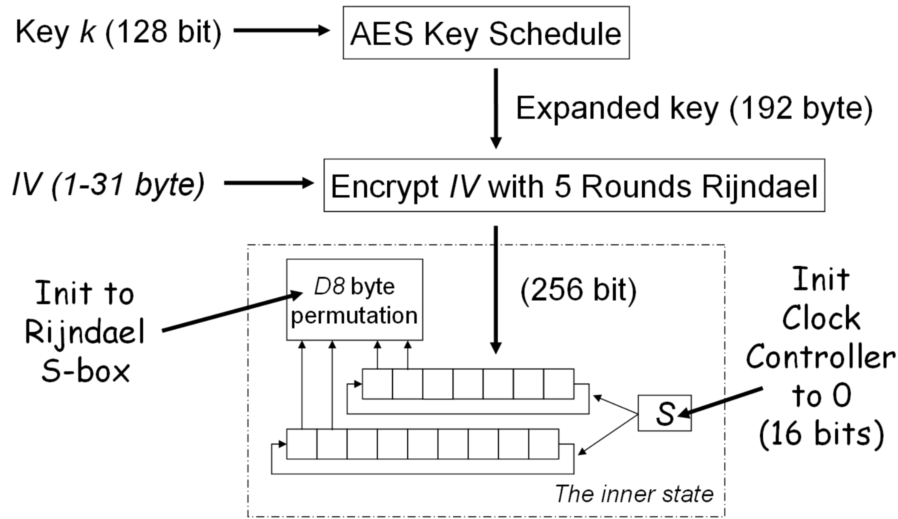

Figure 5.1: *Key and IV schedule*

5.3.2 The output cycle

After each update of the cipher's internal state, four bytes are outputted. Before the first output byte, and between consecutive output pairs of bytes, a state update function is performed as specified below. See also Figure 5.2.

Next state function

Let $\ell_0 = 7$ and $\ell_1 = 9$ be the lengths of the registers. Register R^i is stepped $2 + (\lfloor S/2^{14+i} \rfloor \bmod 2)$ steps with a sparse feedback where each step consists of

- set $f^i \leftarrow \theta^i R^i_{j^i} + \mu^i R^i_0$ for constants[1] θ^i, j^i, and μ^i, where $j^0 = 1$ and $j^1 = 5$.
- set $R^i_j \leftarrow R^i_{j+1}$ for $j = 0, 1, \ldots, \ell_i - 2$
- feedback $R^i_{\ell_i - 1} \leftarrow f^i$.

After stepping both R^0 and R^1 above, do the following steps, first for $i = 0$, then repeat them for $i = 1$:

- Write $(R^i_{\ell_i-1}, R^i_{\ell_i-2})$ as four bytes $\alpha^i_0 || \alpha^i_1 || \alpha^i_2 || \alpha^i_3$.
- Let $\beta^i_j = D_8(\alpha^i_j)$ for $j = 0, 1, 2, 3$.
- Swap elements[2] in D_8 by $D_8(\alpha^i_0) \leftrightarrow D_8(\alpha^i_2)$ and $D_8(\alpha^i_1) \leftrightarrow D_8(\alpha^i_3)$.

Next, update S and R^0

[1] The constants have the values: $\mu^0 = 5CEB_{HEX}$, $\mu^1 = 2C62_{HEX}$, $\theta^0 = 8B5A_{HEX}$, and $\theta^1 = 689A_{HEX}$.
[2] This is not the original swap as explained in Section 6.1.

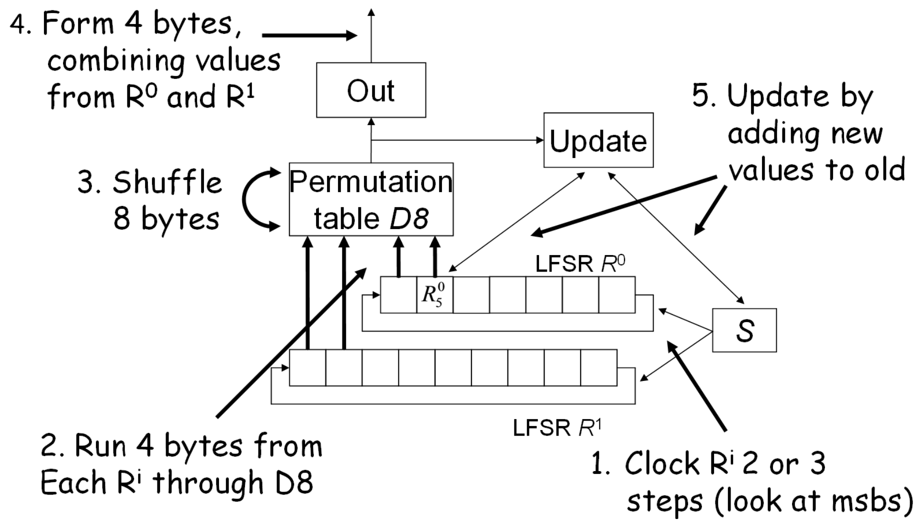

Figure 5.2: *The output cycle*

- Update S according to $S \leftarrow S + \beta_0^1||\beta_1^1 \mod 2^{16}$.
- Update R^0 according to $R_5^0 \leftarrow R_5^0 + \beta_2^1||\beta_3^1 \mod 2^{16}$.

At this point, the internal state is updated, and the output is formed from the above (β_j^0, β_j^1)-pairs as described next.

Output generation

Form four output bytes $b_0||b_1||b_2||b_3$ where

$$b_j = \beta_j^0 \oplus \beta_j^1.$$

If more output bytes are required, the output cycle above is repeated.

Chapter 6

Results

6.1 Permutation Error

The first weakness that we found in the original Polar Bear specification comes from the proposed permutation of $D8$.

$$\begin{array}{rcl} \text{1. new } D_8(\alpha_0) & = & \text{old } D_8(\alpha_2) \\ \text{2. new } D_8(\alpha_1) & = & \text{old } D_8(\alpha_0) \\ \text{3. new } D_8(\alpha_2) & = & \text{old } D_8(\alpha_3) \\ \text{4. new } D_8(\alpha_3) & = & \text{old } D_8(\alpha_1) \end{array}$$

The rows should be executed in the above order. If all four α-values are different, this is a valid permutation consisting of one cycle $(D_8(\alpha_0), D_8(\alpha_1), D_8(\alpha_3), D_8(\alpha_2))$ with three transpositions. But when two or three of the α-values are equal, it might not be a permutation at all. If for example $\alpha_0 = \alpha_1 \neq \alpha_2 \neq \alpha_3 \neq \alpha_0$, the permutation will only affect three elements in $D8$, and the above 'permutation' will result in

$$\begin{array}{rcl} \text{new } D_8(\alpha_0) & = & \text{old } D_8(\alpha_0) \\ \text{new } D_8(\alpha_2) & = & \text{old } D_8(\alpha_3) \\ \text{new } D_8(\alpha_3) & = & \text{old } D_8(\alpha_0) \end{array}$$

The value $D_8(\alpha_2)$ disappears and and the value $D_8(\alpha_0)$ multiplies. The result is that the number of different values in $D8$ are decreasing until only one value remains. As the keystream is generated by XORing elements from $D8$, this has severe consequences for the security of the cipher. The relative frequency of 0's in the keystream will increase until it after a few million bytes consists of zeros only. After this, the cipher will output the unencrypted plaintext as ciphertext.

As the Polar Bear documentation clearly states that it should be a permutation, this is to be seen as a typo. The error was corrected and the correct permutation was stated to be

1. $\text{Swap}(D_8(\alpha_0), D_8(\alpha_2))$
2. $\text{Swap}(D_8(\alpha_1), D_8(\alpha_3))$

which consists of two transpositions. All further references to Polar Bear will be Polar Bear with this corrected permutation.

6.2 A Guess-and-Determine Attack

In this section we present an effective guess-and-determine attack on Polar Bear requiring only a very small amount of known plaintext. Under the assumption of a certain stepping of the registers, a certain sequence of α-values, and a known plaintext, the initial state can be recovered in $O(2^{79})$ time. The key are not recovered, but as the state is recovered at time $t = 0$, the whole message can be determined. Only knowledge of the first 24 bytes of plaintext is needed.

Let the state of LFSR R^i after t steppings of the register be

$$(R^i_{t+\ell_i-1}, R^i_{t+\ell_i-2}, \ldots, R^i_t)$$

where ℓ_i is the length of register R^i. The notation $^*R^0_i$ will be used for stages in R^0 after their update.

Let the first 24 bytes of plaintext be known, and let the corresponding first twelve 16-bit block of keystream be $Z_0, Z_1, \ldots, Z_{11}$.

For the attack to be successful, three assumptions have to be made.

1. During the first six updates of the state, let the steppings for both LFSR R^0 and R^1 be 2-steppings, where the register is stepped two steps. This happens if the fourteenth and fifteenth bit of the word quantity S are 0. Because the word quantity S is initialized to zero, the first stepping for both registers is always a 2-stepping. The probability that the six first steppings is 2-steppings can therefore be assumed to be $(1/2)^{10} = 2^{-10}$.

2. Let no pair of the first eight α be equal. The probability for this is

$$\frac{256!}{(256-8)! \cdot 256^8} \approx 0.90.$$

3. Let no pair of the following forty α be equal. The probability for this is

$$\frac{256!}{(256-40)! \cdot 256^{40}} \approx 0.04.$$

Because all the steppings for both the registers are 2-steppings, all the stages in both LFSRs are used to generate keystream. The probability that all three of the above assumptions holds is greater than $2^{-14.8}$.

Under these assumptions, it suffices to guess the four stages $R^1_9, R^1_{10}, R^1_{11}$ and R^1_{13}, a total of 64 bits, to recover the state. The state can now be recovered with the four equations obtained from the feedback polynomials, the output function, and the nonlinear

update of R^0.

$$
\begin{aligned}
R_i^0 &= \theta^0 \cdot^{(*)} R_{i-6}^0 + \mu^0 \cdot^{(*)} R_{i-7}^0 && (6.1)\\
R_i^1 &= \theta^1 \cdot R_{i-4}^1 + \mu^1 \cdot R_{i-9}^1 && (6.2)\\
Z_i &= \Delta(R_{i+7}^0) + \Delta(R_{i+9}^1) && (6.3)\\
{}^*R_i^0 &= R_i^0 + R_{i+2}^1 \mod 2^{16} && (6.4)
\end{aligned}
$$

The operations in (6.1)–(6.3) are in the finite field $\mathbb{F}_{2^{16}}$, whereas the + in (6.4) is addition modulo 2^{16}. The constants are the ones from the feedback polynomials, and the function $\Delta(x)$ is obtained by looking up the two bytes of x in $D8$, and then concatenate them.

From R_9^1, R_{10}^1, R_{11}^1, R_{13}^1, and (6.3), we get R_7^0, R_8^0, R_9^0, and R_{11}^0. With a knowledge of the four stages R_7^0, R_8^0, R_9^1, and R_{10}^1, we can calculate how $D8$ will be permuted after the first update of the inner state. Let the result of this permutation be $D8'$. As the next 32 α-values are all different, we can treat $D8$ as a constant equal to $D8'$ during the next five updates of the state. The rest of the stages in the registers can now be determined in the following order.
(Where R_i, (6.3) $\rightarrow R_j$ should be read as R_i and (6.3) gives R_j)

$$
\begin{array}{ccc}
R_7^0,\ R_9^0,\ R_{11}^0,\ R_9^1,\ R_{11}^1,\ R_{13}^1,\ (6.4) & \rightarrow & {}^*R_7^0,\ {}^*R_9^0,\ {}^*R_{11}^0\\
{}^*R_7^0,\ R_8^0,\ {}^*R_9^0,\ (6.1) & \rightarrow & R_{14}^0,\ R_{15}^0\\
R_{14}^0,\ R_{15}^0,\ (6.3) & \rightarrow & R_{16}^1,\ R_{17}^1\\
R_{15}^0,\ R_{17}^1,\ (6.4) & \rightarrow & {}^*R_{15}^0\\
R_{11}^1,\ R_{16}^1,\ (6.2) & \rightarrow & R_{20}^1\\
R_{20}^1,\ (6.3) & \rightarrow & R_{18}^0\\
{}^*R_{11}^0,\ R_{18}^0,\ (6.1) & \rightarrow & R_{12}^0\\
R_{12}^0,\ (6.3) & \rightarrow & R_{14}^1\\
R_9^1,\ R_{14}^1,\ (6.2) & \rightarrow & R_{18}^1\\
R_{18}^1,\ (6.3) & \rightarrow & R_{16}^0\\
{}^*R_9^0,\ R_{16}^0,\ (6.1) & \rightarrow & R_{10}^0\\
R_{10}^0,\ (6.3) & \rightarrow & R_{12}^1\\
R_{10}^0,\ {}^*R_{11}^0,\ (6.3) & \rightarrow & R_{17}^0\\
R_{17}^0,\ (6.3) & \rightarrow & R_{19}^1\\
R_{17}^0,\ R_{19}^1,\ (6.4) & \rightarrow & {}^*R_{17}^0\\
R_{10}^1,\ R_{19}^1,\ (6.2) & \rightarrow & R_{15}^1\\
R_{15}^1,\ (6.3),\ (6.4) & \rightarrow & R_{13}^0,\ {}^*R_{13}^0
\end{array}
$$

From $R_9^0, R_{10}^0, \ldots, R_{17}^0$, and starred and unstarred $R_7^0, R_7^0, \ldots, R_{13}^0$, we can determine $D8$ and S, which is the whole state. From this, all future keystream can be calculated.

To try all possible values for the 4 register stages takes $O(2^{64})$ time, and the probability that such an attack is successful is $2^{-14.8}$. The time complexity for the above attack is therefore $O(2^{78.8})$.

6.2.1 An improved attack

Hasanzadeh et al. have lowered the time complexity in a recently presented paper [31]. By using the same attack principle, but with a more careful analysis and selection of

steppings and 'guessed' values, they reach an overall attack complexity of $O(2^{57.4})$. The attack assumes that R^0 is clocked two steps in the first eight steppings, and that the stepping sequence for R^1 is {2, 3, 3, 3, 3, 2, 3, 2}. In this way, they only need to guess the values of two stages. They also notice that, because of the known stepping, only 2^{31} of the 2^{32} values are possible.

6.2.2 Analysis

There are several unfortunate coincidences that make these attacks possible. It is relatively straightforward to see that the attack resistance of Polar Bear does not meet it's key size. For instance, by guessing one (the shorter) LFSR value, it is possible to deduce the value of the other, by observing output. Hence, we have an attack with complexity about 2^{112}. A solution to this would be to make one, or both, of the LFSRs longer. But the LFSRs would have to be almost double the length to withstand attacks like the two describes above. The IV schedule would have to be expanded in some way, that probably would decrease performance.

A second observation is that the dynamic permutation of bytes $D8$ is not permuted before encryption starts, and therefore initially known. If the table $D8$ were mixed during the key/IV schedule, like in RC4, guess-and-determine attacks of this kind would be avoided. But to do a good mixing is slow (The mixing in RC4 is both slow and insufficient). A mixing during each IV schedule would severely reduce the performance on short packages. A better solution, if possible, is to mix the D8 table only during the key schedule.

A third observation is that is to easy to determine more values as soon as some of the stages are known. The reason is that the relations between the register stages and the keystream involves few terms, and are easy to invert, see Section 6.2. These relations come from the feedback trinomials, the choice of the nonlinear updating, and the output generation. It should be mentioned, that an invertible next-state function is desirable to ensure that the state does not loose entropy and converge to a fixpoint. We have tested many different combinations of trinomials, and all seems to be equally vulnerable. If polynomials with more taps than three were used, attacks like the one above would be more difficult, but the cycles/byte performance would be further decreased.

Our last observation is that, even though our attack depends on the possibility of a regular stepping, it is possible that the irregular stepping of Polar Bear actually lowered the attack resistance. Our analysis shows that with a constant 2-stepping, it is impossible to guess fewer than four stages. This implies that no guess-and-determine attack similar to the two described above can have a computational complexity lower than $O(2^{61})$. So as far as we know, the irregular stepping of Polar Bear actually had a negative impact on security. It is however possible that a constant 2-stepping would enable other attacks with an even lower complexity.

6.3 Evaluation with Respect to Other Attacks

6.3.1 Time-memory trade-offs

The huge state of 1956 bits seems to rule out any trade-off involving the state. A birthday attack on the set of key/IV pairs is however possible when the IV is smaller than the key. Polar Bear allows IV sizes as small as 8 bits. This makes a birthday attack with 2^{68} bits memory and computational complexity 2^{68} possible. This is not practical due to the huge amount of memory used, and the attack is not specific to Polar Bear. Almost all eSTREAM candidates allows IVs smaller than the key, and are therefore vulnerable to this kind of generic attack. To avoid such attacks, an IV size v of at least the key size k should be used, or the security of the cipher should be stated to be $(k+v)/2$.

6.3.2 Algebraic attacks

An algebraic attack could potentially be used to recover the initial states of the LFSRs. But as the combiner in Polar Bear consists of the table $D8$, with 1684 memory bits, such attacks seems impossible. Typically, an algebraic attack can only be used when there are very few memory bits.

6.3.3 Correlation attacks

The huge size of the combiner seems to rule out all correlation attacks. The correlation between the key, IV, internal state, and keystream of Polar Bear have been rigorously tested by Turan et al. [51]. They could not find any significant correlations for Polar Bear. They did however find significant correlations for six of the other eSTREAM candidates.

6.4 A new version of Polar Bear - Pig Bear

As the original version of Polar Bear apparently has some weaknesses, a tweak is needed. We have found tweaks that as far as we can tell makes Polar Bear not only secure, but also faster. The main difference is that the table $D8$ is permuted during the key schedule. We have decided to give the tweaked version of Polar Bear the nickname Pig Bear. The name comes from a Swedish rhyme for children [45].

En grisbjörn, en grisbjörn
Min mamma är en isbjörn
Min pappa är en gris
Då ser man ut på detta vis!

Our translation to english (without the last rhyme)

A pig bear, a pig bear
My mother is a polar bear
My father is a pig
Then you look like this!

As small IV sizes enables birthday attacks on the set of key/IV pairs, we recommend that the allowed minimum size is raised. We also think that the consequences of such small IVs should be clearly stated.

6.4.1 Key schedule

We propose that the security is enhanced by adding a key-dependent premixing of the $D8$ table in conjunction with the key schedule. We propose that three full rounds of mixing of D8 is used to this end:

1. Expand the key to 768 bytes of expanded key
2. For $i = 0$ to 767
   Swap($D8[i \pmod{256}]$, $D8[\text{key}[i]]$)

This will only affect the performance of the key schedule. As far as we have been able to tell, no other change is needed.

The reason that the table is mixed three full rounds is that the complexity to get a uniform distribution with the above algorithm is $\Theta(n \log n)$, where n is the size of the table [39]. And this is when the expanded key is a truly random sequence. The RC4 key schedule only mixes the table one full round, and this makes the cipher vulnerable to attacks. According to Mironov [38], this weakness do not disappear unless the table are shuffled two or three full rounds. The AES key expansion is far from random, it is in fact quite weak, especially when related keys are considered. Our analysis shows that if two 128 bits keys with one bit different are used, the resulting expanded keys will on average differ in as few as 49 bits of the first 64 bytes, depending on where the the different bit is. The AES key schedule is therefore insufficient for the above purpose. To fix this another key expansion can be used. We suggest than the key expansion from the Hash function Whirlpool [6] is used instead. We also suggest that this expanded key is used in AES during the IV schedule. Very little of AES security depends on properties of the key schedule. For that reason we do not think that the change makes Polar Bear's IV schedule weaker from a security perspective.

6.4.2 The output cycle

Polar Bear can be made more secure by simply changing a few indexes. The idea behind these two tweaks, is to use different combinations of β-values in different parts of the output cycle. The effect is that it is more difficult to determine LFSR stages from knowledge of other LFSR stages. These two tweaks raises the computation complexity of our attack but are not alone enough to make Polar Bear secure. In our implementations they have no effect on performance.

Update

Update R_3^0 instead of R_5^0 and change the indexes of the β-values.

- Update S according to $S \leftarrow S + \beta_0^1 || \beta_3^1 \mod 2^{16}$.
- Update R^0 according to $R_3^0 \leftarrow R_3^0 + \beta_1^1 || \beta_2^1 \mod 2^{16}$.

Output generation

Change the indexes of the β-values in the output generation. Form four output bytes $b_0||b_1||b_2||b_3$ where

$$
\begin{aligned}
b_0 &= \beta_0^0 \oplus \beta_1^1 \\
b_1 &= \beta_1^0 \oplus \beta_2^1 \\
b_2 &= \beta_2^0 \oplus \beta_3^1 \\
b_3 &= \beta_3^0 \oplus \beta_0^1
\end{aligned}
$$

Permutation tweak

During the permutation of $D8$, $D8(\alpha_1)$ or $D8(\alpha_3)$ might have been changed in the first swap. Because of this, they have to be read from memory a second time, which decreases performance. By reading the β-values in two steps instead of one, the performance of Polar Bear can be increased without affecting security. We propose the following permutation.

1. Let $\beta_0 = D_8(\alpha_0)$ and $\beta_2 = D_8(\alpha_2)$.
2. Swap elements in D_8 by $D_8(\alpha_0) \leftrightarrow D_8(\alpha_2)$.
3. Let $\beta_1 = D_8(\alpha_1)$ and $\beta_3 = D_8(\alpha_3)$.
4. Swap elements in D_8 by $D_8(\alpha_1) \leftrightarrow D_8(\alpha_3)$.

On a Pentium M, this makes Pig Bear approximately 1.7 cycles/byte faster on long streams than Polar Bear.

6.4.3 Security analysis

It is natural to discuss the security of Pig Bear based on existing attacks on Polar Bear, RC4, and AES.

RC4 attacks

The main weaknesses of RC4 are that the first output bytes are not uniformly distributed, and that there exist weak keys that reveals information about specific key bits. No similar weaknesses have been found for Polar Bear, and Pig Bear should be even stronger in this aspect. As the table in Pig Bear are far better shuffled than in RC4 and the output from the LFSRs have nice statistical properties, we do not think that such weak keys should exist.

Related keys

It is generally bad practice to use related keys, but the topic should be discussed anyhow. If the AES key expansion had been used, related key attacks could have been an issue. But as Pig Bear uses the strong key expansion from Whirlpool, a changed bit in the key will result in that half of the bits in the expanded key changes. Hence, as far as we can see, related keys will not be a security issue.

Chosen-IV attacks

If an attacker is able to determine the $D8$ permutation, a guess-and-determine attack can probably be used to attack Pig Bear. From a security perspective, the worst case is when one key is used a large number of times, and the attacker can choose the corresponding IVs. We now have a constant $D8$ unknown for the attacker.

But such an attack would be harder than to break reduced round AES, as the attacker do not have knowledge of the AES ciphertext. The attacker has only knowledge of a complex function of the AES ciphertext.

If patterns in the output from the LFSRs are detectable in the keystream, this could be used to determine the permutation. A simplified version with only one LFSR can be broken in this way, but we believe that by outputting the XOR of the β-values, such attacks are impossible.

6.5 Statistical Testing

Polar Bear and the Pig Bear where both evaluated with respect to the NIST Statistical Test Suite. Both versions passed all the tests since no significant deviations from a random sequence could be found. For the tests, a sample size of 100 keystream sequences of length 10^6 bits were used. Each sequence was created with a randomly chosen key and IV.

6.6 Optimization and Performance

We have been able to optimize the reference code submitted with Polar Bear from 39.1 cycles/byte on a Pentium M to 22.7 cycles/byte. The performance was mainly improved by changing 8 and 16 bit data types to 32 bit only, outputting 32 bit at a time, and arranging operations involving a specific variable consecutively to improve locality. The functions that stepped the registers one step, and that were called two or three times, were changed to functions that step the registers two and three steps at once. A number of other approaches of how to optimize the register stepping were tested, but discarded as they did not improve performance.

Because of the small tweak that changes how the permutation of the table $D8$ is done, Pig Bear is faster than Polar Bear on long sequences as can be seen in the performance figures for the Pentium M 1.6 GHz. This makes Polar Bear faster than AES-CTR (AES in counter mode). All the performance tests in Table 6.1 were done with the eSTREAM

Table 6.1: *Performance figures*

CPU	Name	Stream	40 bytes	Agility	Key Setup	IV Setup
AMD Athlon 64 1.8 GHz	AES-CTR	18.96	23.78	20.57	187.95	12.09
	Polar Bear*	27.63	43.66	30.07	297.81	606.64
HP 9000/785 975 MHz	AES-CTR	17.56	25.92	19.64	215.98	79.57
	Polar Bear*	36.57	57.91	41.12	354.60	819.02
Intel Pentium M 1.7 GHz	SNOW-2.0	4.61	29.82	5.98	63.81	801.73
	RC4**	7.52	335.37	19.52	112.41	13005.38
	AES-CTR	21.78	28.79	24.59	217.74	43.01
	Polar Bear*	39.31	59.29	42.95	273.67	783.70
Intel Pentium M 1.6 GHz	Pig Bear	20.96	.	.	.	.
	Optimized PB	22.69	45.37	26.06	281.81	906.70
	Polar Bear*	39.11	60.74	42.66	269.29	851.63
PowerPC G4 1.67 GHz	AES-CTR	27.06	35.55	31.67	242.69	36.10
	Polar Bear*	44.45	74.52	50.86	276.64	1099.51
UltraSPARC-III 750 MHz	AES-CTR	25.05	34.62	28.50	547.06	121.50
	Polar Bear*	46.50	87.46	49.94	344.22	1646.77
Intel Pentium 4 2.4 GHz	AES-CTR	22.77	31.81	26.69	259.43	68.11
	Polar Bear*	53.40	80.06	59.27	322.85	785.01
Intel Pentium 4 3.0 GHz	SNOW-2.0	5.19	39.04	7.78	90.26	1209.19
	RC4**	10.98	581.88	15.33	193.34	22663.12
	AES-CTR	24.13	33.91	28.01	286.04	93.16
	Optimized PB	30.91	58.22	34.71	343.57	859.00

Stream – Asymptotic encryption rate (cycles/byte). 40 bytes – Packet encryption rate (cycles/byte). Agility – Parallel encryption rate (cycles/byte). Key and IV setup – The efficiency of the key setup and IV setup (cycles)
* These performance tests are made with first reference code that included the permutation error. The corrected code takes approximately 2 cycles/byte more on a Pentium M.
** The key size and IV size was 128 bit for AES-CTR, Polar Bear, and SNOW 2.0. As RC4 do not include support for initialization vectors, the benchmarks are for RC4 with a 256 bit key.

testing framework, and all data except the ones for Intel Pentium M 1.6 GHz are taken from the framework's homepage [10]. The performance figures for the Pentium M 1.6 were created by us with the testing framework Live CD. The Live CD includes Ubuntu Linux, Intel C++ Compiler 8.1, Microsoft Visual C++ Toolkit 2003, and several different version of GCC. The source code is compiled with all three compilers with a large number of compiler options, and the one with the fastest stream performance is chosen.

The stream performance is probably the most important criteria as it is here that stream ciphers has the biggest potential advantage over block ciphers. The key setup is probably the least critical as the time for key setup is typically negligible to the work needed to generate and exchange the key [10].

It should be noted that the data from the two Pentium 4 processors cannot be directly compared as the 3.0 GHz version actually perform less per clock cycle (probably because of memory bottlenecks).

As all optimization was done on a Pentium M, the difference in performance can be expected to be largest on this architecture. As the values in the table above were created with the compiler and options that gave the best performance on long streams, the other benchmarks, such as the numbers of cycles for IV schedule, are not comparable. The IV schedule is not slower in the optimized code, than in the original code. The optimized code is actually a little bit faster, but the value changes with different compiler options.

All variants of the Polar Bear code are fastest with Intel's compiler, SNOW 2.0 and RC4 are fastest with GCC, whereas Microsoft Visual C++ creates the fastest code for AES-CTR.

6.6.1 Further performance tweaks

These tweaks are listed here, and not under Section 6.4, as they are intended to improve the performance, and although we do no think so, might have negative impact on security.

Reduced round IV schedule

With the premixing of the $D8$ table, we believe that the key diffusion during the IV schedule can be reduced. We therefore propose that the number of AES rounds is reduced from five to four. This increases the performance of the IV schedule with up to 20 percent.

Table 6.2: *Performance figures for the speed tweaks*

Name	Stream
Pig Bear RF*	12.69
Pig Bear R	15.68
Pig Bear F*	18.62
Pig Bear	20.96
Optimized PB	22.69

Asymptotic encryption rate in cycles/byte on Pentium M 1.6 GHz.

* These figures are from a modified Pig Bear source, and can likely be optimized further.

Regular stepping

As it is not sure that the irregular stepping improved the security, one might argue that it should be removed. Such a tweaked version (Pig Bear R) with only 2-steppings, improves the stream performance drastically (see table 6.2). The reason is probably that the branch prediction of the CPU works better.

Fewer swaps

Another tweak that improves the performance, is fewer swaps in the table $D8$. The fact is that Polar Bear does twice as many swaps per outputted byte as RC4. If the α-values from the LFSR R^0 are not filtered through $D8$ (Pig Bear F), the encryption speed for long streams is improved (see Table 6.2). The four output bytes $b_0||b_1||b_2||b_3$

now becomes

$$
\begin{aligned}
b_0 &= \alpha_0^0 \oplus \beta_1^1 \\
b_1 &= \alpha_1^0 \oplus \beta_2^1 \\
b_2 &= \alpha_2^0 \oplus \beta_3^1 \\
b_3 &= \alpha_3^0 \oplus \beta_0^1
\end{aligned}
$$

If the tweaks are combined, the output measured in Mbit/s is improved with 65 percent over Pig Bear. The tweaked versions of Pig Bear, R and F, are not as optimized as the Pig Bear implementation and can probably be optimized further. The above tweaks are not recommended on short streams, unless further analysis of their security implications is made. We do however recommend that the tweaks are used on long streams. The cipher could switch to this simpler form when the table are thoroughly shuffled, perhaps after 768 or 1024 outputted bytes. We believe that the $D8$ table makes the cipher secure asymptoticly.

Chapter 7

Conclusions

The original Polar Bear specification had a major weakness as it could be attacked using a guess-and-determine attack with a computational complexity of only $O(2^{57.4})$ (Hasanzadeh et al.). We believe that Polar Bear can be made secure by adding a key-dependent pre-mixing of the D8 table in conjunction with the key schedule. Further tweaks strengthen the security and improves the performance on long sequences.

We have not found any other weaknesses in Polar Bear, and it seems resistant to all known generic attacks. Polar Bear and Pig Bear passes all of the statistical tests in the NIST statistical test suite. Polar Bear also passes new statistical tests that focuses on correlation and are tailored for stream ciphers.

The first evaluation phase of eSTREAM ended in late march 2006. The 35 candidates are initially classified into three categories.

- Focus Phase 2 – Ciphers of particular interest. Mainly unbroken ciphers with very good performance.
- Phase 2 – Ciphers moved to the second phase. Mostly broken ciphers with good performance.
- Archived – Ciphers no longer considered for the final portfolio.

The main criteria for the classification are cryptanalysis and performance, but instead of exact limits, a committee is evaluating each cipher. Further, patented ciphers have not been placed in the focus category. The authors are permitted to submit tweaks to their algorithms before June 30, 2006.

Polar Bear has advanced to the second phase of eSTREAM in both the software and hardware profile. Polar Bear is in both cases placed in the Phase 2 category. The tweak will consist of the one suggested in this thesis. Unfortunately, Polar Bear with its current implementation do not have any "significant performance advantage compared to AES". The performance tests show that Polar Bear perform worst of the remaining ciphers in the software profile. It is a reasonable guess that Polar Bear will not survive the second classification in the end of 2006 unless this is changed.

It might be argued that it is unfair to compare the performance of new ciphers with the best known C implementation of AES, a cipher that since five years ago is an

international standard. The number of hours devoted by a large number of people to optimize the performance of AES can probably be counted in the tenths of thousands.

7.1 Future Work

Stream ciphers are a relatively new field of scientific study and much remain to be done. There is no real understanding of how to create a both secure and fast stream cipher.

In the case of Polar Bear, the new tweaked version that we intend to make available to the eSTREAM community soon should and will be extesivly evaluated. The hardware performance of Polar Bear is currently unknown, hopefully this will change when the eSTREAM hardware testing framework is done.

Bibliography

[1] eSTREAM - The ECRYPT Stream Cipher Project.
http://www.ecrypt.eu.org/stream/

[2] NIST Statistical Test Suite.
http://csrc.nist.gov/rng/

[3] NIST FIPS-PUB 197. Advanced Encryption Standard (AES). Technical report, November 2001.
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[4] Frederik Armknecht and Matthias Krause. Algebraic attacks on combiners with memory. In *Advances in Cryptology - CRYPTO 2003*, pages 162–175. Springer-Verlag, 2003

[5] S. H. Babbage. Improved exhaustive search attacks on stream ciphers. In *ECOS 95 (European Convention on Security and Detection)*, pages 161–166. IEEE Conference publication, May 1995

[6] Paulo S.L.M. Barreto and Vincent Rijmen. The Whirlpool Hashing Function, 2003.
http://planeta.terra.com.br/informatica/paulobarreto/whirlpool.zip

[7] Norman L. Biggs. *Discrete Mathematics*. Oxford University Press, 2003. ISBN 0198507178

[8] Robert G. Brown. DieHarder: A Random Number Test Suite.
http://www.phy.duke.edu/~rgb/General/dieharder.php

[9] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.
http://www.cs.cmu.edu/~dbrumley/pubs/openssltiming.pdf

[10] Christophe De Cannire. eSTREAM testing framework.
http://www.ecrypt.eu.org/stream/perf/

[11] Christophe De Cannire, Joseph Lano, and Bart Preneel. Comments on the Rediscovery of Time Memory Tradeoffs. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/040, 2005.
http://www.ecrypt.eu.org/stream/papersdir/040.pdf

[12] Carlos Cid and Henri Gilbert. AES Security Report. Technical Report ST-2002-507932, ECRYPT, January 2006.
`http://www.ecrypt.eu.org/documents/D.STVL.2-1.0.pdf`

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press, 2001. ISBN 0-262-53196-8

[14] Nicolas Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology - CRYPTO 2003*, pages 176–194. Springer-Verlag, 2003

[15] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. Cryptology ePrint Archive, Report 2002/044, 2002.
`http://eprint.iacr.org/2002/044.pdf`

[16] Nicolas T. Courtois. Higher Order Correlation Attacks, XL algorithm and Cryptanalysis of Toyocrypt. Cryptology ePrint Archive, Report 2002/087, 2002.
`http://eprint.iacr.org/2002/087.pdf`

[17] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Information Security and Cryptography. Springer-Verlag, Berlin, 2002. ISBN 3-540-42580-2

[18] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.
`http://www.cs.jhu.edu/~rubin/courses/sp03/papers/diffie.hellman.pdf`

[19] Patrik Ekdahl and Thomas Johansson. A New Version of the Stream Cipher SNOW. In *SAC '02: Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*, pages 47–61, London, UK, 2003. Springer-Verlag. ISBN 3-540-00622-2.
`http://www.it.lth.se/cryptology/snow/snow20.pdf`

[20] Diane Erdmann and Sean Murphy. An Approximate Distribution for the Maximum Order Complexity. *Des. Codes Cryptography*, 10(3):325–339, 1997.
`http://www.isg.rhul.ac.uk/~sean/moc.ps`

[21] Scott R. Fluhrer and David A. McGrew. Statistical Analysis of the Alleged RC4 Keystream Generator. In *FSE '00: Proceedings of the 7th International Workshop on Fast Software Encryption*, pages 19–30, London, UK, 2001. Springer-Verlag. ISBN 3-540-41728-1

[22] P. R. Geffe. How to Protect Data with Ciphers That Are Really Hard to Break. *Electronics*, pages 99–101, Jan 1973

[23] Jovan Dj. Golić. Cryptanalysis of alleged A5 stream cipher. In *Eurocrypt'97 LNCS 1233*, pages 239–255. Springer-Verlag, 1997.
`http://www.gsm-security.net/papers/Cryptanalysis_of_Alleged_A5_Stream_Cipher.pdf`

[24] Johan Håstad and Mats Näslund. BMGL: Synchronous Key-stream Generator with Provable Security, 2000.
https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/bmgl4.pdf

[25] Johan Håstad and Mats Näslund. The Stream Cipher Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/021, 2005.
http://www.ecrypt.eu.org/stream/ciphers/polarbear/polarbear.pdf

[26] P. Hawkes and G. Rose. The applicability of distinguishing attacks against stream ciphers. in Proceedings of the Third NESSIE Workshop, 2002, 2002.
http://eprint.iacr.org/2002/142.pdf

[27] Jin Hong and Palash Sarkar. Rediscovery of Time Memory Tradeoffs. Cryptology ePrint Archive, Report 2005/090, 2005.
http://eprint.iacr.org/2005/090.pdf

[28] Fredrik Jönsson. *Some results on fast correlation attacks.* PhD thesis, Lund University, May 2002.
http://homes.esat.kuleuven.be/~jlano/stream/papers/jon02.ps

[29] Andrew Klapper and Mark Goresky. Feedback Shift Registers, 2-Adic Span, and Combiners With Memory. *Journal of Cryptology*, 10(2):111–147, March 1997.
http://www.math.ias.edu/~goresky/pdf/2adic.jour.pdf

[30] Andrew Klapper and Jinzhong Xu. Algebraic feedback shift registers. *Theor. Comput. Sci.*, 226(1-2):61–92, 1999

[31] Elham Shakour Mahdi Hasanzadeh and Shahram Khazaei. Improved Cryptanalysis of Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/084, 2005.
http://www.ecrypt.eu.org/stream/papersdir/084.pdf

[32] Itsik Mantin and Adi Shamir. A Practical Attack on Broadcast RC4. In *FSE '01: Revised Papers from the 8th International Workshop on Fast Software Encryption*, pages 152–164, London, UK, 2002. Springer-Verlag. ISBN 3-540-43869-6.
http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/bc_rc4.ps

[33] George Marsaglia. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness

[34] James L. Massey. Shift-Register Synthesis and BCH Decoding. *IEEE Transactions on Information Theory*, IT-15(1):122–127, January 1969

[35] John Mattsson. A Guess-and-Determine Attack on the Stream Cipher Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/017, 2006.
http://www.ecrypt.eu.org/stream/papersdir/2006/017.pdf

[36] Willi Meier and Othmar Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1(3):159–176, 1989

[37] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN 0-8493-8523-7.
http://www.cacr.math.uwaterloo.ca/hac/

[38] Ilya Mironov. (Not So) Random Shuffles of RC4. In *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, pages 304–319, London, UK, 2002. Springer-Verlag. ISBN 3-540-44050-X.
http://eprint.iacr.org/2002/067.pdf

[39] Elchanan Mossel, Yuval Peres, and Alistair Sinclair. Shuffling by semi-random transpositions, 2004.
http://arxiv.org/PS_cache/math/pdf/0404/0404438.pdf

[40] Ron Rivest, Adi Shamir, and Len Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
http://theory.lcs.mit.edu/~rivest/rsapaper.pdf

[41] Phillip Rogaway and Don Coppersmith. A Software-Optimized Encryption Algotithm. *Journal of Cryptology*, 11(4):273–287, November 1998.
http://www.cs.ucdavis.edu/~rogaway/papers/seal.ps

[42] Rainer A. Rueppel. Stream Ciphers. In Gustavus J. Simmons, editor, *Contemporary Cryptology: The Science of Information Integrity*, chapter 2, pages 65–134. IEEE Press, New York, 1992. ISBN 0-87942-277-7

[43] Markku-Juhani 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/papersdir/2006/013.pdf

[44] Marcus Schafheutle and Stefan Pyka. Stream Ciphers. Technical report, NESSIE consortium, February 2003. 103–122 pp.
https://www.cosic.esat.kuleuven.be/nessie/deliverables/D20-v2.pdf

[45] Bea Uusma Schyffert and Kari Modn. En grisbj"orn. In Birgitta Westin, editor, *Bom Bom*, page 31. Raben & Sj"'ogren, Stockholm, 2005. ISBN 91-29-65987-6

[46] Adi Shamir and Eran Tromer. Acoustic cryptanalysis: on nosy people and noisy machines.
http://www.wisdom.weizmann.ac.il/~tromer/acoustic/

[47] Claude Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28(4):656–715, 1949.
http://www.cs.ucla.edu/~jkong/research/security/shannon1949.pdf

[48] Thomas Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, 30(5):776–780, 1984

[49] Thomas Siegenthaler. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Transactions on Computers*, 34(1):81–85, January 1985

[50] Simon Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography.* Anchor Books, New York, 2000. ISBN 0-385-49532-3

[51] Meltem Sönmez Turan, Ali Doğanaksoy, and Çağdas Çalik. Statistical Analysis of Synchronous Stream Ciphers. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/012, 2006.
`http://www.ecrypt.eu.org/stream/papersdir/2006/012.pdf`

[52] A.C. Yao. Theory and applications of trapdoor functions. In *Proceeings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982

[53] Arm M. Youssef and Guang Gong. On the Quadratic Span of Binary Sequences. Technical Report CORR 2000-20, University of Waterloo, March 2000.
`http://www.cacr.math.uwaterloo.ca/techreports/2000/corr2000-20.ps`

The URLs were accessed on June 16, 2006.

TRITA-CSC-E 2006:111
ISRN-KTH/CSC/E--06/111--SE
ISSN-1653-5715

www.kth.se