

# Fault Analysis of Stream Ciphers

M.Sc. Thesis

Ya'akov Hoch

[yaakov.hoch@weizmann.ac.il](mailto:yaakov.hoch@weizmann.ac.il)

Advisor: Adi Shamir

Weizmann Institute of Science  
Rehovot 76100, Israel

## **Abstract**

A fault attack is a powerful cryptanalytic tool which can be applied to many types of cryptosystems which are not vulnerable to direct attacks. The research literature contains many examples of fault attacks on public key cryptosystems and block ciphers, but surprisingly we could not find any systematic study of the applicability of fault attacks to stream ciphers. Our goal in this work is to develop general techniques which can be used to attack the standard constructions of stream ciphers based on LFSRs, as well as more specialized techniques which can be used against specific stream ciphers such as RC4, Scream and various NESSIE candidates. While most of the schemes have been successfully attacked, we point out several interesting open problems such as attacks on FSM filtered constructions and the analysis of high Hamming weight faults in LFSRs.

# Acknowledgements

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Background . . . . .	6
1.2	Physical Fault Induction . . . . .	7
1.3	Fault Attack Models . . . . .	8
1.4	A Taste of Fault Attacks . . . . .	8
1.5	Overview of the Thesis . . . . .	11
<b>2</b>	<b>Attacks on Synthetic LFSR Based Stream Ciphers</b>	<b>12</b>
2.1	Introduction . . . . .	12
2.2	Classical (Direct) Attacks on LFSR Based Stream Ciphers . . . . .	13
2.2.1	Correlation Attacks . . . . .	13
2.2.2	Algebraic Attacks . . . . .	14
2.2.3	Re-synchronization attacks . . . . .	14
2.3	Attacks on Non-Linearly Filtered LFSR Based Stream Ciphers . . . . .	15
2.3.1	Checking the Guess . . . . .	16
2.3.2	Constructing the Linear Equations . . . . .	17
2.3.3	Unknown Filter Functions . . . . .	18
2.4	Attacks on Clock Controlled LFSR Based Stream Ciphers . . . . .	20
2.4.1	A phase shift in the data register . . . . .	21
2.4.2	Faults in the clock register . . . . .	21
2.4.3	Faults in the data register . . . . .	23
2.5	Attacks on Finite State Machine Filtered LFSR Based Stream Ciphers . . . . .	25
2.5.1	Randomizing the LFSR . . . . .	25
2.5.2	Faults in the FSM . . . . .	25
<b>3</b>	<b>Fault Attacks on Real Life LFSR-Based Stream Ciphers</b>	<b>27</b>
3.1	A Fault Attack on LILI-128 . . . . .	27
3.2	A Fault Attack on SOBER-t32 . . . . .	29
3.2.1	Stripping the Stuttering . . . . .	29

3.2.2	Recovering the LFSR State . . . . .	31
3.3	A Fault Attack on SNOW 2.0 . . . . .	32
<b>4</b>	<b>Attacks on Other Real Life Stream Ciphers</b>	<b>34</b>
4.1	An Attack on Scream . . . . .	34
4.1.1	The Basic Attack . . . . .	34
4.1.2	Detecting in which variable the fault occurred . . . . .	35
4.1.3	Identifying where in the variable the fault occurred . . . . .	35
4.1.4	Recovering the input to the $F$ Function . . . . .	36
4.1.5	The actual attack . . . . .	36
4.1.6	An Attack Against Scream-S . . . . .	37
4.1.7	Fault Identification . . . . .	37
4.2	An Attack on RC4 . . . . .	38
<b>5</b>	<b>Summary</b>	<b>41</b>
5.1	Summary of the Results . . . . .	41
5.2	Further Work . . . . .	41

# List of Figures

2.1	Filtered LFSR . . . . .	15
2.2	Clock Controlled LFSR . . . . .	20
2.3	An example of a Phase Shift Attack . . . . .	21
3.1	LILI-128 . . . . .	27
3.2	SOBER-t32 . . . . .	30
3.3	SNOW 2.0 . . . . .	32
4.1	The main loop of Scream and Scream-0 . . . . .	35
4.2	The G and F functions . . . . .	36
4.3	Pseudo-code for RC4 . . . . .	38
5.1	Result summary . . . . .	41

# List of Algorithms

1	CRT-RSA . . . . .	9
2	Unknown CryptoSystem - Phase I . . . . .	10
3	Unknown CryptoSystem - Phase II . . . . .	10
4	Attack on Non-Linearly Filtered LFSRs . . . . .	16
5	Checking the guess . . . . .	16
6	Attack Utilizing Faults in the Clock Register . . . . .	22
7	Recovering the Clock LFSR from the Data LFSR . . . . .	23
8	Utilizing Faults in the Data LFSR . . . . .	24
9	Faults in the FSM . . . . .	26
10	Attack Against LILI-128 . . . . .	28
11	Stripping the Stuttering . . . . .	30
12	Recovering the LFSR State . . . . .	31
13	Attack on RC4 . . . . .	39
14	Biham et all Attack on RC4 . . . . .	40

# Chapter 1

## Introduction

### 1.1 Background

In modern cryptography it is common practice to divide ciphers into two classes: block ciphers and stream ciphers. A block cipher is a cipher which operates on chunks of plaintext. Block ciphers are usually slower than stream ciphers and are primarily used in applications in which the data rate is relatively low. On the other hand stream ciphers are commonly composed of a PRG (pseudo-random generator) which produces a pseudo-random stream of bits which is then bitwise xored with the data stream to produce the ciphertext. Stream ciphers are usually very fast, requiring only a few CPU cycles per word of encrypted output, and are typically used in applications which require very high data rates.

Attacks against cryptosystems can be divided into two classes, direct attacks and indirect attacks. Direct attacks include attacks against the algorithmic nature of the cryptosystem regardless of its implementation. Indirect attacks make use of the physical implementation of the cryptosystem and include a large variety of techniques which either give the attacker some ‘inside information’ on the encryption process (such as power[20] or timing analysis [19]) or some kind of influence on the cryptosystem’s internal state such as ionizing radiation flipping random bits in the device’s internal memory. Fault analysis is based on a careful study of the effect of such faults (which can affect either the code or the data) on the ciphertext, in order to derive (partial) information about either the key or the internal state of the cryptosystem. Fault analysis was first used in 1996 by Boneh, Demillo, and Lipton in [2] to attack number theoretic public key cryptosystems such as RSA (by using a faulty CRT computation to factor the modulus  $n$ ), and later by Biham and Shamir in [3] to attack product block ciphers such as DES (by using a high-probability differential fault attack on the last few rounds). While these techniques were generalized and applied to other public key and block ciphers in many subsequent papers, there are almost no published results on

the applicability of fault attacks to stream ciphers, which requires different types of attacks and analytic tools. The goal of this thesis is to fill this void by embarking on a systematic study of all the standard techniques used to construct stream ciphers, and by analyzing their vulnerability to various types of fault attacks.

## 1.2 Physical Fault Induction

Fault attacks have been successfully conducted in laboratories mainly embedded implementation of block ciphers and public key cryptosystems. Different physical techniques of applying the faults can result in different fault models. The most common techniques for injecting fault are (for more details see [24]):

- Varying the external voltage to the cryptoprocessor can cause the processor to misinterpret or skip instructions.
- Varying the external clock can cause the cryptoprocessor to misread data. For example, the processor accesses the bus is before the memory had time to latch out the requested value. This type of fault is consistent with the fault model we use in this thesis.
- Shinning the cryptoprocessor with intense burst of visible light can cause memory bits to flip. The internal registers of the cryptoprocessor must be exposed for this technique to work.
- Shinning the cryptoprocessor with laser light has similar effect as visible light, with the advantage that it is easier to localize the fault. Laser can also be used at a higher intensity to cut lines in the cryptoprocessor resulting in changes to a logic gate or overwriting ROM memory cells.
- Using X-Rays or ion beams can also produce faults. The advantage of this type of radiation is that the cryptoprocessor does not have to be exposed.
- In components aboard spacecraft it is common to experience SEU (single event upset) events due to the cosmic radiation. These faults flip a single bit in a specific memory cell.

Of course the various types of hardware are not all equally sensitive to fault attacks. For example, the fact that certain types of non-volatile memory are sensitive to a non-symmetric probability of bit-flipping (i.e., a one bit is more likely to change into a zero bit than vice versa) when electro-magnetic radiation is applied, was used by Biham and Shamir in [3]. While in SRAM due to the symmetric way the memory cell is implemented,

ionizing radiation is more likely to have a symmetric probability of bit-flipping. Recently Anderson in [1] discovered an extremely low-tech, low-cost technique which allows an attacker with physical access to the cryptoprocessor (especially when implemented on a smartcard) to cause faults at very specific locations. Anderson's technique utilizes a tabletop optical microscope to focus the light from a camera flash onto a very small area of the integrated circuit. This extremely simple apparatus was used to affect even single bits in the internal registers of the cryptoprocessor. This discovery transfers the ability to perform fault attacks to one's backyard making this kind of attack a major threat to smartcard issuers and users. To summarize, while at first the cost of conducting an invasive fault attack could be quite high (de-packaging the cryptoprocessor, equipment for producing intense laser beams, etc) we are seeing that the the cost of applying these attacks is now significantly lower and can be conducted even in a rudimentary lab setting.

### 1.3 Fault Attack Models

The basic attack model used in this thesis assumes that the attacker can apply some bit flipping faults to either the RAM or the internal registers of the cryptographic device, but that he has only partial control over (and knowledge of) their number, location and timing. In addition, he can reset the cryptographic device to its original state and then apply another randomly chosen fault to the same device. In general, there is a tradeoff between the amount of control he has and the number of faults needed to recover the key. This model tries to reflect a situation in which the attacker is in possession of the physical device, and the faults are transient rather than permanent. Other fault attack models which we have not considered in this thesis include:

- Permanent bit failures in the data (cause for example by a stuck bit in a memory cell)
- Faults which affect the code of the program instead of the data resulting in different instruction being carried out (either transiently or permanently)

Both types of faults have been used in actual lab implementations of fault attacks.

### 1.4 A Taste of Fault Attacks

The first fault attack we will describe is the original attack by Boneh, Demillo and Lipton[2] against a cryptosystem implementing RSA with a Chinese Remainder Theorem (CRT) computation. In RSA the act of encrypting a message  $M$  involves computing  $C = M^d \bmod n$

where  $n$  is a large number  $n = p \cdot q$  and  $p$  and  $q$  are primes. In order to speed up the exponentiation, most implementations of RSA use algorithm 1.

---

**Algorithm 1** CRT-RSA

---

1. Calculate  $C_p = M^d \bmod p$
  2. Calculate  $C_q = M^d \bmod q$
  3. Use the CRT to compute  $C = CRT(C_p, C_q)$
- 

Suppose we have encrypted our message once and produced the ciphertext  $C$ . We now encrypt the message again, but this time we apply a fault during the execution of the algorithm. Since the most computationally demanding part of the algorithm is the exponentiation modulo  $p$  and  $q$  it is very probable that the fault will occur during the first two steps of the algorithm. We assume without loss of generality that the fault occurred during the computation of  $C_p$ . Let  $C'_p$  be the result of the faulted computation modulo  $p$  and  $C' = CRT(C'_p, C_q)$  the output of the faulted encryption. Notice that  $C' = C \pmod{q}$  but  $C' \neq C \pmod{p}$  this implies that  $GCD(C - C', n) = q$  resulting in a factoring of  $n$ .

The second attack we describe will be that of Biham and Shamir[20] against an unknown cryptosystem. Assume that we have a cryptosystem  $E$  which can encrypt blocks of plaintext. We further assume that the key is stored in non-volatile memory such as EEPROM in which in the presence of ionizing radiation the probability of a one bit flipping to a zero is much higher than the opposite. Let  $k$  be the unknown key used by the encryption, our goal will be to recover  $k$ .  $k'$  will stand for the current content of the key material.

---

**Algorithm 2** Unknown CryptoSystem - Phase I

---

1. Encrypt a message  $M$  and produce the ciphertext  $C_0 = E_k(M)$
  2. Set  $i = 1$
  3. Reset the device
  4. Apply radiation to the device
  5. Encrypt a message  $M$  and produce the ciphertext  $C_i = E_{k'}(M)$
  6. If  $C_i = C_{i-1}$  increase the radiation intensity and goto step 3, if radiation is over a threshold level quit.
  7. Increase  $i = i + 1$  and Goto 3
- 

Notice that algorithm 2 will produce a sequence of ciphertexts  $E_0, E_1, \dots, E_n$  s.t. the Hamming weight of  $E_{i+1}$  is one less than the Hamming weight of  $E_i$ . This is because when we apply the radiation we only flip bits from one to zero, and we keep the radiation level low enough to ensure that we only flip a single bit each time. Also notice that the algorithm terminates with  $E_n = 0$  since this is the only  $k'$  for which there can be no further changes. We now proceed to recover the original key  $k$  one bit at a time.

---

**Algorithm 3** Unknown CryptoSystem - Phase II

---

1. Set  $k = 0$
  2. Set  $i = 1$
  3. Reset the device
  4. Set  $k'$  to be a key reachable from  $k$  by flipping a single bit
  5. Produce  $C = E_{k'}(M)$
  6. If  $C = E_{n-i}$  then set  $k = k'$ ,  $i = i + 1$ . Otherwise return to step 4 and try a different bit
  7. Continue until  $i = n$
- 

During the execution of the algorithm we successively find the keys which produced  $E_i$  for  $i = n \dots 0$ . The algorithm terminates with  $k = k'$  recovering the original key.

## 1.5 Overview of the Thesis

We have succeeded in attacking a wide variety of stream ciphers. We have mainly concentrated on attacking constructions based on LFSRs. While there are other types of constructions which can replace the LFSR's role as a source of a statistically good stream such as T-functions[26] or FCSRs (Feedback with Carry Shift Registers) [25], we have chosen to ignore them in this thesis as the techniques we developed against LFSR do not readily apply to these construction. With the exception of FSM filtered constructions we were able to attack almost any synthetic LFSR based construction which appeared in the literature, and even against FSM filtered constructions we have a number of results. The linearity of the LFSR is at the heart of all of these attacks. These results are covered in chapter 2, where we present a comprehensive attack strategy against non-linearly filtered LFSRs as well as attacks against other synthetic LFSR based constructions. In chapter 3 we present fault attacks against three NESSIE candidates: LILI-128, Sober-t32 and SNOW. All of these ciphers are stream ciphers based on LFSRs. Chapter 4 describes fault attacks against various other stream ciphers and includes attacks against RC4 and Scream. All the attacks were analyzed theoretically and verified by computer simulation, in order to gain better understanding of their actual complexity and success rate. However, they were not tested experimentally by inducing actual faults in a concrete physical implementation. Chapter 5 gives a summary of the results and a discussion of open questions and possible future research in the field.

# Chapter 2

## Attacks on Synthetic LFSR Based Stream Ciphers

### 2.1 Introduction

Linear Feedback Shift Registers (LFSRs) are a very common component in stream ciphers. LFSR's have long cycles and good statistical properties, but due to their inherent linearity LFSRs do not generate good output streams by themselves. Hence, LFSRs are typically used in conjunction with some non-linear component. There are three general constructions for implementing a stream cipher based on LFSRs:

- Filter the output of the LFSR(s) through a non-linear function.
- Have the clocking of one LFSR controlled by the output sequence of another LFSR.
- Filter the output of the LFSR(s) through a finite state machine.

We will now give a formal definition of an LFSR and cite a few important properties which will be used later in this work.

**Definition 2.1.** *A LFSR has two components:*

- *An internal state  $\{X_i\}_{i=1}^n \in \{0, 1\}^n$  called the Register*
- *A linear update function  $L$  specified by the Feedback Taps  $c \in \{0, 1\}^n$ .*

*At each time step the output of the LFSR is  $X_n$  and  $X$  is updated to  $LX$  or specifically  $X_i = X_{i-1}$  for  $i > 1$  and  $X_1 = \langle X, c \rangle$ . where  $\langle, \rangle$  specifies the inner product in  $\{0, 1\}^n$  over  $GF(2)$ .*

The cycle length of the LFSR (for a non-zero starting point) is determined by the feedback taps. In cryptographic applications these are selected to ensure a maximum length cycle of  $2^n - 1$ .

**Proposition 2.2.** *Every output bit of the LFSR can be represented as a linear combination of the initial state bits.*

**Corollary 2.3.** *Given  $n$  output bits from the LFSR, such that the corresponding linear relations in the initial state bits are independent, we can reconstruct the initial state by solving the corresponding system of  $n$  linear equations in  $n$  unknown bits over  $GF(2)$ .*

As the update function is linear we can compute the content of  $X$  at time  $t$  by  $(L^t)X$  thus allowing us to compute future states of the LFSR efficiently through fast matrix exponentiation.

We will use the following notation for the rest of this work:

$\oplus$	bitwise exclusive or over bits or words
$\boxplus$	addition modulo $2^j$ where $j$ will be obvious from the context
$\lll_i$	cyclic rotation left by $i$ bits

**Proposition 2.4.** *Due to the linearity of the update function we have that if  $X = Y \oplus \Delta$  then  $L^n X = L^n(Y \oplus \Delta) = L^n Y \oplus L^n \Delta$*

In other words, knowing an initial difference in the LFSR state allows us to compute all future differences in the LFSR state.

In this chapter we will develop several types of fault attacks against the generic constructions described at the beginning of this chapter. We denote the length of the LFSR by  $n$ , the XOR of the original and faulted value of the LFSR at the time the fault was introduced by  $\Delta$ , and the number of affected bits by  $k$ .

## 2.2 Classical (Direct) Attacks on LFSR Based Stream Ciphers

Besides the novel fault attacks we develop in this thesis there is a considerable number of existing techniques for attacking stream ciphers. Before we start describing fault attacks against various LFSR based stream ciphers, we will give a short description of the leading classical attacks against these constructions.

### 2.2.1 Correlation Attacks

In a correlation attack [22] we assume that the attacker has access to a sequence of bits which is correlated with the raw output stream of one of the LFSR components. The attacker takes that bit sequence as his first approximation of the raw LFSR output stream. He then uses the linear recurrence of the LFSR to successively improve his approximation until he recovers the actual raw output sequence from the LFSR. Now using corollary 2.3 he can recover the initial

state of the LFSR component. The algorithms used for performing a fast correlation attack are highly dependent on the number of feedback taps in the LFSR and are not practical for more than 10 feedback taps. Nowadays the non-linear components of LFSR based stream ciphers are chosen to ensure that no significant correlation exists between any of the LFSR components and the output stream.

### 2.2.2 Algebraic Attacks

An algebraic attack [12] against an LFSR based stream cipher consists of two major steps: finding a system of algebraic equations involving the bits of the key and the output bits  $o_i$  as unknowns. Since the LFSR is linear we have that if an equation  $G$  in the internal state and output bits holds at time  $t$ :

$$G(x_1, x_2, \dots, x_n, o_t, \dots, o_{t+k}) = 0 \quad (2.1)$$

Then due to the linearity of the LFSR we have that at any future time  $t + i$ :

$$G(L_1(x), L_2(x), \dots, L_n(x), o_{t+i}, \dots, o_{t+i+k}) = 0 \quad (2.2)$$

For easily computable linear combinations  $L_1, \dots, L_n$ . So if we have enough output bits we will get an over-defined system of algebraic equations. The simplest method for solving this system is Linearization (others include Groebner base algorithms[28] and XL[27]). In this method we replace any non linear term in the equations be a new variable and solve the resulting system of linear equations. This requires that the new system be over-defined and thus we need about  $O(V^D)$  output bits where  $V$  is the number of variables in the original equations and  $D$  is the maximal degree of the original equations. This means that the algebraic attack is only feasible when we can construct low degree equations with a relatively small number of variables for the given cipher. Nevertheless, algebraic attacks are the best known attacks against many stream ciphers including  $E_0$ ,  $LILI - 128$  and Tokyocrypt.

### 2.2.3 Re-synchronization attacks

It is common practice for stream ciphers to be frequently re-initialized. The reason could be either the need to re-synchronize the sender and receiver or to avoid using long sequences produced by the same key. In order to reduce the amount of secret information required, the cipher is re-initialized with the same key but with different (and publicly known) initialization vectors IVs. In a re-synchronization attack [9] the attacker has access to a number of output streams generated with the same key but with different initialization vectors IVs. The attacker then uses this information to derive information about the key.

## 2.3 Attacks on Non-Linearly Filtered LFSR Based Stream Ciphers

Let  $(x_1, x_2, \dots, x_n)$  be the internal state of the LFSR where  $x_i \in \{0, 1\}$ . A non-linear filter applied to a LFSR is a boolean function  $f(x_{i_1}, x_{i_2}, \dots, x_{i_t})$  whose inputs are a subset of the LFSR's internal state bits (typically,  $n \leq 128$  and  $t \leq 12$ ). More generally the inputs to the function may come from several LFSRs. Each output bit is produced either by evaluating  $f$  on the current state, or by using a lookup table of pre-computed values of  $f$ . The LFSR is then clocked and  $f$  is evaluated again on the resulting state to generate the next output bit.

Figure 2.1: Filtered LFSR

Existing attacks against this construction include the algebraic attack which is generally infeasible when  $t$  is not extremely small and the re-synchronization attack which shares a similar setting with our attack. The main difference between the fault scenario we use in this thesis and the re-synchronization scenario lies in the attacker's knowledge and control over the difference in the initial state. In the re-synchronization scenario, the attacker has no control over the difference in the initial state while he has perfect knowledge of this difference. In our fault model, the attacker assumes some control over the above difference, e.g. a low Hamming weight of the fault, but assumes no further knowledge about the fault.

We now assume that the attacker has the power to cause low Hamming weight faults in the LFSR's internal state bits. The main advantage of using such faults is that the number of possible faults is relatively small, and thus it can be guessed with a non-negligible probability. The attack will proceed as follows:

---

**Algorithm 4** Attack on Non-Linearly Filtered LFSRs

---

1. Cause a fault and produce the resulting output stream
  2. Guess the fault
  3. Check the guess using Algorithm 5, if incorrect guess again
  4. Repeat 1-3 until  $O(t)$  identified guesses are collected
  5. Construct and solve a system of linear equations in the original state bits
- 

---

**Algorithm 5** Checking the guess

---

1. Predict future differences in the input to  $f$  based on the guess of the initial fault
  2. Identify bit locations for which the prediction is for a zero input difference
  3. For these bit locations check if the observed output difference is zero, if not reject the guess.
- 

### 2.3.1 Checking the Guess

To show that algorithm 4 we first need to show the correctness of algorithm 5, i.e., that it can identify incorrect guesses. Notice that due to the linearity of the LFSR clocking operation  $L$ , if we know the initial difference  $\Delta$  due to the fault then at any time  $i$  the difference will be  $L^i(\Delta)$  and we do not have to know the actual state in order to compute it. To verify a guess for  $\Delta$  we predict the future differences in the  $t$  input bits to  $f$ . Whenever this difference is 0 we expect (if our guess was correct) to see an output difference of 0. If our guess was incorrect, then for half of these occasions we will see a non-zero output difference. So on average after  $2^{t+1}$  output bits we expect to reject an incorrect guess. Since we have  $\binom{n}{k}$  possible faults we need on average about  $\log\binom{n}{k} \cdot 2^{t+1}$  output bit to uniquely identify the fault.

Notice that after we have identified the first  $i - 1$  faults, we can use this information to identify the  $i$ -th fault faster. In step 2 of algorithm 5, we identify bit locations where we predict a zero input difference between our current guess and any of the  $i$  available streams. For the second fault this will save  $\frac{1}{2}$  of the data needed, and in general if for the  $j$ -th fault we save a factor of  $C_j$  then for the  $i$ -th fault we will save a factor of  $\frac{1}{\sum_{j < i} C_j}$ . For the parameters of  $n = 128$ ,  $k = 3$  and  $t = 10$  this will save a factor of over 60% in the total amount of data required.

We can sometimes improve the amount of data needed for the attack by analyzing the structure of  $f$ . Define  $A = \{\Delta \mid Pr[f(x) \oplus f(x \oplus \Delta) = 0] > \frac{1}{2} + \epsilon\}$ . After guessing  $\Delta$ ,

the initial difference, we compute as before the differences  $\Delta_n = L^n(\Delta)$  at any future time. When  $\Delta_n \in A$  we know that with probability at least  $\frac{1}{2} + \epsilon$  the difference in the output of  $f$  will be 0. I.e, the average of the difference over the output bits for which  $\Delta_n \in A$  should be  $\frac{1}{2} + \epsilon$ . If our guess of  $\Delta$  was incorrect then we expect to see an average of  $\frac{1}{2}$ . Thus after seeing about  $O(\epsilon^{-2} \frac{|A|}{2^n})$  we should be able to tell with high probability whether our guess of  $\Delta$  was correct. Analysis of  $f$  will show us the optimal  $\epsilon$  and whether we achieve an advantage over the previous strategy.

If the Hamming weight of the faults is very low then we can apply another strategy to reduce the amount of data required by guessing and verifying  $m$  faults simultaneously. This will increase the time complexity by a factor of  $\binom{n}{k}^{m-1}$ , but we can now check our guess by comparing the relative difference in the input of  $f$  for each pair of the  $m + 1$  streams. This gives us a probability of approximately  $2^{-t} \binom{m+1}{2}$  of having a zero relative difference, thus reducing the amount of data required by a factor of  $\binom{m+1}{2}$ . For example, for the parameters  $k = 1, m = 4, t = 10, n = 128$ , we only need  $\frac{1}{12}$  of the data and our running time will increase by a factor of  $2^{28}$ . However, our running time will still be manageable at around  $2^{36}$  basic operations.

### 2.3.2 Constructing the Linear Equations

It remains to show how to construct the system of linear equations. We start by introducing the notion of a linear structure.

**Definition 2.5.** *A 0-order linear structure of  $f$  is an  $n$ -bit vector  $\gamma$  s.t. for all  $X$   $f(X) = f(X \oplus \gamma)$*

Notice that for every  $f$  we always have that  $\gamma = 0$  is a trivial linear structure.

**Proposition 2.6.** *The set  $\Gamma$  of all 0-order linear structures of  $f$  forms a vector space.*

Now let us concentrate on a single output bit. For each faulted stream the attacker observes the difference in the output bit and can compute, based on the known fault, the input difference to  $f$ . After repeating the above a number of times, we collect pairs of input/output differences corresponding to the same output bit location. We will show later how to deal with functions that contain non-trivial linear structures. Under the assumption that  $f$  does not contain non-trivial linear structures we have the following analysis. On average for each input difference about half of the possible actual inputs will be compatible with the observed output difference so each fault eliminates on the average half of the possible inputs. Hence given about  $t$  pairs of input/output differences, we can narrow down by exhaustive search the possible input bits to a single possibility. By determining these bits we get linear equations over  $GF(2)$  in terms of the initial state bits. Using the same faulted

output streams we can also compute the input differences for other output bits collecting more linear equations. Once we collect enough ( $\theta(n)$ ) equations we can solve the set of equations and determine the initial LFSR state.

We will now analyze what happens when  $f$  contains non-trivial linear structures. If  $f$  contains any non-trivial 0-order linear structure, then no matter how many input/output difference pairs we have for a specific output bit, they alone will not uniquely determine the actual input at that bit location. The reason for this is that by definition 2.5 for every input  $Y$  consistent with the observed input/output difference pairs,  $Y \oplus \gamma$  will also be consistent with the observations for every  $\gamma \in \Gamma$ . This means that we need to find another source for our linear equations. However, because of proposition 2.6 we know that the actual input to  $f$  is in the affine space  $Y + \Gamma$  and hence we can write a linear equation on the actual input. Since this input can be described by linear combinations of the initial state bits, we have a linear equation in the original state bits. As before, after collecting enough such equations we can solve for the initial state of the system.

We can pre-compute the 0-order linear structures of  $f$  by computing the autocorrelation function of  $f$  [9], [11].

**Definition 2.7.** *The autocorrelation function of  $f$  is defined as*

$$K_f(\gamma) = \frac{1}{2^t} \sum_{x \in \{0,1\}^t} (-1)^{f(x)+f(x+\gamma)}$$

**Lemma 2.8.** *If  $g = f(x \oplus c) \oplus d$  for some fixed  $c \in \{0,1\}^t$  and  $d \in \{0,1\}$  then*

$$K_f(\gamma) = K_g(\gamma)$$

Notice that  $K_f(\gamma) = 1$  iff  $\forall x \in \{0,1\}^t f(x) = f(x + \gamma)$ . Or in other words  $K_f(\gamma) = 1$  iff  $\gamma$  is a 0-order linear structure of  $f$ . Since  $k_f = \frac{1}{2^t} f * f$  we can use  $\hat{f}$  the Walsh-Hadamard transform [23] of  $f$  to compute the necessary convolution in time  $t2^t$  by noticing that  $\widehat{f * f} = \hat{f} \cdot \hat{f}$ . So we first compute the Walsh-Hadamard transform of  $f$ , which can be done in time  $t2^t$ , then multiply  $\hat{f}$  by itself point-wise (time  $2^t$ ), and finally compute the inverse transform again in time  $t2^t$ .

### 2.3.3 Unknown Filter Functions

So far we assumed that the filter function  $f$  is known, but we can apply a fault attack even if  $f$  is unknown. First notice that in order to verify a guessed fault in algorithm 5 we did not need to know  $f$ . So we can carry out steps 1-4 of algorithm 4 even when the non-linear function  $f$  is unknown or key-dependent.

**Definition 2.9.** *Let  $D(i)$  be the set of input-output difference pairs resulting from the faults at position  $i$  in the output stream.*

*$D_x(i)$  will be the output difference at location  $i$  for an input difference of  $x$ .*

First we claim that if we have for some  $i$   $|D(i) = 2^t$  we can calculate the 0-order linear structures of  $f$ . If we define a function  $g$  s.t.,  $g(x) = D_x(i)$  and let  $c$  be the actual input to  $f$  at time  $i$  then we have:

$$g(x) = f(x \oplus c) \oplus f(c) \quad (2.3)$$

So by lemma 2.8 we have that the autocorrelation function of  $g$  is identical to that of  $f$ . Hence by computing the autocorrelation function of  $g$  we can derive the 0-order linear structures of  $f$ .

Now if for two positions  $i$  and  $j$   $D(i) = D(j)$  and  $|D(i)| = 2^t$  then either the un-faulted inputs  $X, Y$  to  $f$  at positions  $i$  and  $j$  were the same or  $X \oplus Y$  is a 0-order linear structure of  $f$ . As shown in the previous subsection, in either case, we can construct linear equations in the original state variables. After recovering the LFSR state we can easily recover  $f$ . Notice that when choosing a filter function effort will be made to ensure that no linear structures exist because the existence of linear structures enable other direct attacks (correlation, linear analysis, etc). Therefore it is reasonable to expect that  $f$  does not contain linear structures. Similarly we can assume that  $f$  is balanced, since otherwise it would not be very secure for use in a stream cipher. Under the above assumptions we can check with high probability whether  $D(i) = D(j)$  by checking for each input difference that occurs in both sets whether the corresponding output differences are the same. For each input difference which resides in both sets, we have a probability of  $\frac{1}{2}$  that the output differences will be different if the actual inputs were different. So in order to ensure with high probability that  $X = Y$  we need:

$$|D(i) \cap D(j)| \geq \log 2^t = t \quad (2.4)$$

Calculating the expectation of  $|D(i) \cap D(j)|$  we get:

$$E[|D(i) \cap D(j)|] = 2^t \cdot \left(\frac{\#faults}{2^t}\right)^2 \quad (2.5)$$

And since we want:

$$E[|D(i) \cap D(j)|] \geq t \quad (2.6)$$

This implies:

$$\#faults \geq \sqrt{t} \cdot 2^{\frac{t}{2}} \quad (2.7)$$

This means that in practice we do not need  $t \cdot 2^t$  faults (to ensure  $|D(i)| = 2^t$ ) but can with high probability use only  $\sqrt{t} \cdot 2^{t/2}$  faults.

The only property of the LFSR which we used for these attacks is that we can compute future differences based on the initial fault. Thus the attacks generalize directly to a construction composed of several LFSRs connected to the same non-linear filter, providing that the total Hamming weight of the faults in all the registers is low. However, we were unable to find any fault attacks utilizing faults with high (and thus un-guessable) Hamming weight.

## 2.4 Attacks on Clock Controlled LFSR Based Stream Ciphers

The basic clock controlled LFSR construction is composed of two components: the clock LFSR and the data LFSR. The output stream is a subsequence of the output of the data LFSR which is determined by the clock LFSR. For example, when the clock LFSR output bit is 0 clock the data LFSR once and output its bit, and when the clock LFSR bit is 1 clock the data LFSR twice and output its bit. Unless specified otherwise, all attacks in this section will refer to this construction.

Figure 2.2: Clock Controlled LFSR

Other variations include considering more than one bit of the clock LFSR to control the clocking of the data LFSR (E.g., in LILI-128 two bits of the clock LFSR are used to decide whether to clock the data LFSR one to four times). The last variation considered here is the shrinking generator [6] in which the output bits of the clock LFSR decide whether or not the current data LFSR output bit will be sent to the output stream, and thus there is no fixed upper bound on the time difference between consecutive output bits. Existing attacks against clock controlled constructions include correlation attacks [10], algebraic attacks [12] and re-synchronization attacks [10].

Throughout this section we will use the term *data stream* to indicate the sequence produced by the data LFSR  $\{d_i\}_{i=1}^{\infty}$  as opposed to the *output stream* denoted  $S = \{S_i\}_{i=1}^{\infty}$  which is the sequence of output bits produced by the device. The control sequence produced by the clock LFSR will be denoted  $\{c_i\}_{i=1}^{\infty}$ , and we define  $pos_S(i)$  to be the position of the  $i^{th}$  bit of the output stream  $S$  in the data stream.

110101001001010 - clock register  
 001010110101010100101010 - data register  
 110100100101001 - output stream

01010110101010100101010 - data register after phase shift  
 000101101100011 - output stream

110100100101001 - original output stream  
 001011011000110 - faulted output stream

Each bit in the original sequence is compared with the bit to its left in the faulted sequence. When a difference is observed the clock register must have been 1.

\*1\*\*\*1\*\*1\*\*1\*1\* - Partial data recovered by comparing the two sequences.\  
 110101001001010 - The actual clock register.

Figure 2.3: An example of a Phase Shift Attack

### 2.4.1 A phase shift in the data register

A phase shift is a fault in which one of the components is clocked while the other is not. Once the phase shift takes place the device continues operating as usual. In a clock controlled construction a phase shift in the data LFSR can give us information about the clock register. Denote by  $S$  the non-faulted output stream and by  $\hat{S}$  the faulted output stream. Notice that for every bit  $i$  after the fault  $pos_{\hat{S}}(i) = pos_S(i) + 1$  since the data register was clocked one extra time. So the attacker looks for  $i$  s.t.  $\hat{S}_i \neq S_{i+1}$ , this implies that at the  $i^{th}$  location the data register was clocked twice. Thus we can recover a bit of the clock LFSR state (which corresponds to a linear equation in the original state) each time we have such an occurrence.

We need about twice the length of the clock register to recover the whole state since the probability of such an occurrence is  $\frac{1}{2}$ . After recovering the clock LFSR's state we can easily recover the data LFSR's since we now know the position of each output bit in the data stream.

It is left as an easy exercise to show that this attack can be adapted to deal with phase shift faults in the shrinking generator and the stop & go generator.

### 2.4.2 Faults in the clock register

For simplicity of description we assume that the attacker can apply random single bit faults to the clock LFSR at a chosen point in the execution. The same principal used in this simplified description can be carried out even if the timing of the fault is not exactly known and it affects a small number of bits. The first stage of the attack will be to produce the  $n$

---

**Algorithm 6** Attack Utilizing Faults in the Clock Register

---

1. Generate faulted streams until  $n$  distinct streams are produced
  2. Identify bit locations in which we can recover a bit  $c_i$  of the current clock LFSR state
  3. Repeat steps 1&2 at different timings until  $n$  bits of the clock LFSR sequence  $\{c_i\}$  have been identified
  4. Construct and solve a system of linear equations over  $GF(2)$  in the original state bits of the clock LFSR
  5. Utilizing the now known locations of the output bits in the data LFSR sequence construct and solve a system of linear equations over  $GF(2)$  in the original state bits of the data LFSR
- 

possible separate faulted output streams by applying a single bit fault at the same timing (at different unknown locations) to the clock register. We will designate the stream resulting from a fault in the  $i^{th}$  location by  $S^i$ ,  $S_j^i$  being the  $j^{th}$  bit of  $S^i$  (counting from the timing of the fault). Let us observe  $S_j^i$  for a fixed  $j$  s.t.  $j < n$ . This condition assures that the feedback of the clock register has not affected the output stream yet as a result of the fault. I.e., the only changes are a result of the single bit change at the  $i^{th}$  location. If  $i \geq j$  then the fault will not have enough time to affect  $S_j^i$  and  $S_j^i = S_j$ . However, if  $i < j$  then similar to the phase shift example,  $|pos_{S^i}(j) - pos_S(j)| = 1$ . If  $c_i = 1$  then we will get  $pos_{S^i}(j) - pos_S(j) = -1$  (we have clocked the data LFSR one time less) and  $pos_{S^i}(j) - pos_S(j) = 1$  if  $c_i = 0$ . Now assume that for all  $i$   $S_j^i$  is the same. This implies that both neighbors of the original bit in the data stream are identical to the bit itself.

...0 $\hat{0}$ 0... - the original data stream where the  $\hat{*}$  was chosen for the output

... $\hat{0}$ 00... - the original data with faulted clocking

...00 $\hat{0}$ ... - the original data with another faulted clocking

The only other case in which this could happen is if the first  $j$  bits of the clock register were identical, since then we only see one of the neighbors. By choosing  $j$  large enough we can neglect this possibility. If we see  $j - 1$  streams which are identical in the  $j^{th}$  bit but different from the original  $j^{th}$  bit then the data stream must have looked as follows:

...1 $\hat{0}$ 1... - the original data stream where the  $\hat{*}$  was chosen for the output

In this case we know that both neighbors of the bit in the data stream were equal. If the next output bit in the actual stream was different from the neighbors, then the data register must have been clocked twice.

...0 $\hat{0}$ 0 $\hat{1}$ ... - the  $\hat{*}$  bits were chosen for the output

...1 $\hat{0}$ 1 $\hat{0}$ ... - the  $\hat{*}$  bits were chosen for the output

In this case we have recovered a bit of the clock LFSR (since we know the data LFSR has been clocked twice) or more generally a linear equation in the original LFSR state. By analyzing all bit sequences of length up to 5 bits we found that there is a probability of at least  $\frac{6}{32}$  of situation occurring from which we can derive the clocking bit. Hence we can get about  $\frac{3n}{16}$  linear equations. We now repeat the attack and collect another batch of faulted streams with the timing of the faults changed. After repeating this procedure  $\sim 10$  times we will have collected an over-determined set of equations which we can solve for the clocking LFSR's original state. After recovering the clock LFSR we can easily solve for the data LFSR. The attack requires about  $10n$  faults and for each fault a little more than  $n$  bits (for unique identification of the streams). This attack is also applicable to the decimating and stop & go generators since the effect of a single bit fault in the control LFSR is also locally identical to a phase shift in the data LFSR.

### 2.4.3 Faults in the data register

The next attack will focus on the data LFSR, but before we give a description of the attack we will show a general algorithm for recovering the clock register given the data register.

---

**Algorithm 7** Recovering the Clock LFSR from the Data LFSR

---

1. Initialize  $Equations, Locations = \emptyset, i = 1$
  2. Update  $Locations$  according to  $d_i$
  3. If  $|Locations| = 0$  return "Incompatible"
  4. If  $|Locations| = 1$  add the corresponding linear equation to  $Equations$
  5. If  $|Equations| < n$  goto step 2
  6. Solve the system  $Equations$  for the initial state of the clock LFSR
- 

For a clock controlled construction  $pos(i) = \sum_{j=1}^i c_j$  is the position of the  $i^{th}$  bit of the output stream in the data stream. The input to the algorithm will be the sequence  $\{d_i\}$  and we will identify  $pos(i)$  for various  $i$ . Notice that each value of  $pos(i)$  gives us a linear equation over  $GF(2)$  in the original state of the LFSR, since each of the  $c_i$ 's can be represented as a linear combination of the original state bits and  $pos(i)$  is a linear combination of the  $c_i$ 's. Once we have collected enough values we can solve the set of equations for the initial state of the clock LFSR. The algorithm works by keeping a list of all possible values of  $pos(i)$  for each output bit of the device. This is done by simple elimination: check for each existing position in the list whether it is possible to receive the actual output with one of the possible

values of  $c_i$ . Now if we find an  $i$  such that the list of candidates for  $pos(i)$  is a single value we know the corresponding  $pos(i)$ . Experimental results show that given a random initial state for LFSRs of size 128 bits, the algorithm finds the original state after seeing a few hundred bits, finding a linear equation every 5 or 6 bits. If the output sequence was not produced from  $\{d_i\}$  then the algorithm finds an inconsistency in the output stream (the size of the list shrinks to zero) after at most a few tens of bits. This behavior can also be studied analytically. Let  $x_i$  and  $y_i$  be the minimal and maximal candidate values for  $pos(i)$  respectively. Assuming  $y_i$  is not the real value for  $pos(i)$  let us calculate the expectation of  $y_{i+1} - y_i$ . This expectation is bounded from above by  $\frac{5}{4}$ , since there is a probability of  $\frac{1}{2}$  that the maximum grows by 2 and a probability of  $\frac{1}{4}$  that the maximum grows by 1. On the other hand the expectation of  $x_{i+1} - x_i$  is bounded from below by  $\frac{1}{2} + \frac{2}{4} + \frac{3}{8} = \frac{11}{8}$  so the expectation of the change to the size of the list of possibilities for  $pos(i)$  is negative. I.e., the size of the list is expected to shrink unless one of the endpoints is the true position. This implies that the average size of the list is constant and thus the running time is linear. Now our attack will proceed as follows:

---

**Algorithm 8** Utilizing Faults in the Data LFSR

---

1. Generate a non-faulted output stream of length  $10n$
  2. Re-initialize the device, and cause a low Hamming weight fault in the data register
  3. Generate a new (faulted) stream of length  $10n$
  4. Guess the fault and verify it by running algorithm 7 with the calculated difference in the data stream and the output stream difference
  5. Repeat until the guess is consistent with the output stream
  6. Recover the data register state from the actual output and the known clocking register
- 

Since the clocking register was not affected, the difference in the output stream is equivalent to a device with the same clocking and with the data register initialized to the fault difference. Since given a guess of the initial state of the data register, the attacker can calculate the difference at any future point, we can apply the algorithm for recovery of the clock register. For incorrect guesses of the fault, the algorithm will find the inconsistency and for the correct guess the algorithm will find the initial state of the clock register.

We have presented attacks which utilize faults either in the data LFSR **or** in the clock LFSR. It is natural to ask whether we can deal with faults which affect both LFSRs simultaneously. We were not successful in adapting our techniques to deal with simultaneous faults and the main reason for this is that we rely on the local differences between the faulted

and non-faulted streams. When the faults affect only one component, it is relatively easy to analyze the local behavior of the fault while for simultaneous faults things get mixed up in an unstructured way.

## 2.5 Attacks on Finite State Machine Filtered LFSR Based Stream Ciphers

In this section we will show some attacks on a basic FSM filtered LFSR construction. The FSM contains some memory whose initial content is determined by the key. Each time the LFSR is clocked, the LFSR output bit is inserted into a specific address determined by a subset of the LFSR's state, and the bit previously occupying that memory location is sent to the output. The number of memory bits will be denoted by  $M$  and thus there are  $\log M$  address bits. The leading approach against general FSM filtered LFSR constructions are algebraic attacks [12], but since algebraic attacks is feasible only when the attacker can construct a set of low degree algebraic equations, these attacks are only feasible against constructions for which the attacker can construct such a set (e.g., Sober-t32[15]).

### 2.5.1 Randomizing the LFSR

Assume that the attacker has perfect control over the timing of the fault, and that he can cause a fault which uniformly randomizes the LFSR bits used to address the FSM. The first output bit after the fault has been applied will be uniformly distributed over the bits currently stored in the FSM. By repeating the fault at the same point in time we can estimate the ratio of zeros to ones in the memory and thus recover the number of ones currently stored in the FSM. If we do the same at a different point in time we can, by examining the actual output stream, recover the total number of ones entering the FSM. This gives us a linear equation over  $GF(2)$  in the initial LFSR state. By collecting  $\theta(n)$  equations we will get an independent set which we can solve for the initial state.

### 2.5.2 Faults in the FSM

If a random fault is applied to the current contents of the FSM the output stream will have differences at the timings when the LFSR points to the faulted bits' addresses. We start by giving some intuition about the attack. Assume that the LFSR points to the same address at two consecutive clockings. If the fault in the FSM happened at this location before these points in time, only the first occurrence of this location in the output stream will be faulted. When examining the second occurrence no matter what fault occurred in the FSM the bit will not be faulted as long as the timing of the fault was before the first occurrence. When we

notice a case like this we know that the address is the same in the two consecutive timings, this gives us linear relations on the bits of the LFSR. As before, after collecting  $\theta(n)$  relations we can derive the LFSR state. More generally, let  $p$  be the probability of a single bit in the

---

**Algorithm 9** Faults in the FSM

---

1. Reset the device, generate a fault and produce the resulting stream
  2. Repeat step 1 until enough statistics are collected
  3. Analyze the statistics and construct linear equations in the original LFSR state
  4. Repeat steps 1-3 until an over-defined system of linear equations is collected and solve it
- 

FSM being affected by the fault and let us assume that the timing of the fault is uniformly distributed over an interval  $[t_1, t_2]$  of length  $T$ . The probability of a difference in bit  $t$  between the faulted and non-faulted streams is  $\frac{t-t_1}{t_2-t_1}p$  provided that this is the first occurrence of the address. If the most recent occurrence of the same address before time  $t$  is at time  $t_0$  then the probability is  $\frac{t-t_0}{t_2-t_1}$ . So by estimating this probability within  $\frac{1}{2(t_2-t_1)}$  we can tell when the address bits were the same at two different timings  $t_0$  and  $t$ . This gives us  $\log M$  linear equations in the original LFSR bits. We repeat this  $\frac{n}{\log M}$  times and recover the initial state of the LFSR from the resulting set of linear equations.

# Chapter 3

## Fault Attacks on Real Life LFSR-Based Stream Ciphers

### 3.1 A Fault Attack on LILI-128

In this section we will bring some of the techniques presented into action in a fault attack against LILI-128 [4], one of the NESSIE candidates.

Figure 3.1: LILI-128

LILI-128 is composed of two LFSRs:  $LFSR_c$ , which is 39 bits long, and  $LFSR_d$ , which is 89 bits long (with a total of 128 bits of internal state). Both have maximum length cycles. For each keystream bit:

- The keystream bit is produced by applying a nonlinear function  $f_d$  on a fixed set of 10 bits in  $LFSR_d$ .

- $LFSR_c$  is clocked once. Two fixed bits from  $LFSR_c$  determine an integer  $c$  in the range  $\{1, 2, 3, 4\}$ .
- $LFSR_d$  is clocked  $c$  times.

The keystream generator is initialized simply by loading the 128 bits of key into the registers. Keys that cause either register to be initialized with all zeroes are considered invalid. The exact function  $f_d$  used, which bits are taken as inputs and the feedback taps of the LFSRs are irrelevant to the attack. The best known direct attack against LILI-128[31] has data complexity of  $2^{46}$  uses a lookup table of  $2^{45}$  89-bit words and a pre-computational effort which is roughly equivalent to  $2^{48}$  DES operations.

---

**Algorithm 10** Attack Against LILI-128

---

1. Generate the 89 distinct streams resulting from single bit faults in  $LFSR_d$
  2. Evaluate the streams to recover bits of the  $LFSR_c$
  3. Repeat steps 1,2 at different timings until 39 linearly independent equations are collected and solve the set of linear equations for the initial state of  $LFSR_c$
  4. Using the now known position of  $LFSR_d$  for the output bits use algorithm4 to recover the initial state of  $LFSR_d$
- 

The first stage of our attack is to apply a random single bit fault to the data register and produce the resulting stream. Reset the device and repeat this until 89 (the length of  $LFSR_d$ ) distinct streams are observed. Now repeat the same with the construction clocked once before applying the faults. Notice that some of the streams produced will be the same as in the first batch. This is due to the fact that applying the fault and then shifting the LFSR is equivalent to shifting the LFSR and then performing the fault, provided the fault did not affect the feedback. By counting how many streams are repeated one can deduce how many times  $LFSR_d$  was clocked, which provides two bits of  $LFSR_c$ . Thus after repeating the experiment about 20 times we can recover the full  $LFSR_c$  state. Once this state is known we can use the algorithm 4 to recover the state of  $LFSR_d$ . Notice that no further faults are necessary and the data collected in the previous stage can be reused. A tradeoff between the number of faults used and the length of the attack can be achieved by stopping after part of the state has been recovered and guessing the rest.

## 3.2 A Fault Attack on SOBER-t32

SOBER-t32 [7] is another NESSIE candidate with a LFSR based design. SOBER is composed of a LFSR, a non linear filter (NLF) and a form of irregular decimation called *stuttering*. The LFSR works over the field  $GF(2^{32})$  and produces a stream of 32-bit (L-)words  $L_1, L_2, \dots$  called the L-stream. The internal state of the LFSR will be denoted  $\sigma_i = (s_i, s_{i+1}, \dots, s_{i+16})$ , and  $\sigma_0$  will denote the initial state. The L-stream is fed through the NLF to produce 32-bit (N-)words  $N_1, N_2, \dots$  called the N-stream.  $N_i = NLF(\sigma_i)$ . The stuttering decimates the N-stream as follows: the first N-word  $N_1$  is the first stutter control word SCW. The SCW is partitioned into 16 pairs of bits, each pair of bits is read in order and accordingly one of four actions is performed:

1. Clock the LFSR once but do not output anything.
2. Clock the LFSR once, output the current N-word xored with  $0x6996C53A$  and clock the LFSR again (without producing more output).
3. Clock the LFSR twice and then output the current N-word.
4. Clock the LFSR once and then output the current N-word xored with  $0x96693AC5$ .

When all the bits of the SCW have been read, the LFSR is clocked and the output of the NLF becomes the next SCW. The NLF is defined as :

$$NLF(\sigma_i) = ((f(s_i + s_{i+16}) + s_{i+1} + s_{i+6} \oplus Konst) + s_{i+13}) \quad (3.1)$$

where  $f$  is some non linear function whose exact definition is not relevant to the attack.  $\sigma_0$  and  $Konst$  are key dependent.

The best known direct attack is against Sober-t32 without stuttering [15] and has data complexity of  $2^{79}$  and time complexity of  $2^{85}$ .

The attack will proceed in two stages, the aim of the first stage is to strip away the stuttering and recover the full N-stream, i.e., the output after the NLF. The second stage is to recover the original state of the LFSR based on the faults as seen in the N-stream.

### 3.2.1 Stripping the Stuttering

To achieve this stage we assume that we can apply random single bit faults to the output of the N-stream. If we damage a word which is not a stutter control word, then depending on whether the word appeared in the original stuttered output we will see either a single bit difference in the faulted output stream or no change at all. If we fault a stutter control word, then we will see a significant difference in the output stream. However, we know that

Figure 3.2: SOBER-t32

both streams originated from the same N-stream hence we can use them to reconstruct the original N-stream. To check whether two output words originated from the same N-word we simply check if their xor is in the set  $\{0, 0x6996C53A, 0x96693AC5\}$ , and the probability of a wrong identification is negligible since we are matching 32-bit words. We know that in each stream the order of the words is the same so with enough faults we can fully reconstruct the N-stream. Since the probability of a N-word being sent to the output is slightly below  $\frac{2}{3}$  (remember that  $\frac{1}{17}$  of the N-words are used as SCWs) it is enough to cause  $\sim 10$  faults in the SCW to ensure that we reconstruct a significant part of the N-stream. Since the probability

---

**Algorithm 11** Stripping the Stuttering

---

1. Generate a fault in the output of the N-stream and produce the resulting stream
  2. Identify and discard faults in SCW words
  3. Repeat steps 1,2 until we can reconstruct the N-stream
-

of causing a fault in a SCW is  $\frac{1}{17}$ , we can carry out this stage of the attack with less than 200 faults.

### 3.2.2 Recovering the LFSR State

---

**Algorithm 12** Recovering the LFSR State

---

1. Generate a fault in  $\sigma_{13}$  and produce the resulting output stream
  2. Identify the first non-zero difference in the output stream
  3. If the difference is of the form  $\pm 2^j$  construct a linear equation over  $GF(2)$  in the original LFSR state
  4. Repeat 1-3 until we have  $32 \cdot 17 = 544$  independent equations and solve the set for the original LFSR state
- 

Now we will use faults to the LFSR to retrieve it's original state. We first apply a fault to  $\sigma_{13}$  where  $\sigma$  is the current state of the LFSR. Let us denote the timing of the fault by  $i$ , i.e., we faulted  $\sigma_{i+13}$ . Notice that we have not assumed control over the timing of the fault, only over the location of the fault within the LFSR. We observe the first nonzero difference in the output stream which results from our fault. If  $N_t$  was sent to the output then the observed difference with respect to subtraction mod  $2^{32}$  will be  $\sigma_{i+13} - \hat{\sigma}_{i+13} = \pm 2^j$  where  $\hat{\sigma}_{i+13}$  represents the faulted version and  $j$  is the bit faulted. If  $N_i$  was not sent to the output then the first observed difference is very unlikely to be of the above form. The sign of the difference will give us the original bit in the  $j^{th}$  position (we are exploiting the nonlinearity of  $+$  with respect to  $\oplus$ ). Notice that until now we have not used the fact that we know the N-stream. Since we know the position of the current output word in the N-stream we know the exact place of the bit recovered in the L-stream and hence have a linear equation in the original bits of the equivalent  $GF(2)$  LFSR. By repeatedly applying faults we can recover enough linear equations and reconstruct the initial state. After reconstructing the LFSR state we can find *konst* from the equation for the NLF, the observed N-stream and the calculated L-stream.

In the full description of SOBER-t32, there is also a key-loading procedure which mixes the secret key a session key to initialize *konst* and  $\sigma_0$ . A similar fault attack can be applied to recover the secret key from the session key and the initial state.

### 3.3 A Fault Attack on SNOW 2.0

SNOW 2.0[21] is an improved version of SNOW 1.0 which was one of the NESSIE candidates. The current best direct attack on SNOW 2.0 is against a modified version (see [16] has data complexity of  $2^{17}$  and time complexity of  $2^{45}$  with a pre-computation step of  $2^{446}$ . Snow 2.0 is composed of a 16 stage LFSR working over the field  $GF(2^{32})$  and of a FSM. The contents of the LFSR at time  $t$  will be denoted  $(s_{t+15}, s_{t+14}, \dots, s_t)$ . The FSM contains two 32-bit registers denoted  $R1$  and  $R2$ .

Figure 3.3: SNOW 2.0

At each time step the LFSR is clocked and the FSM registers are updated. The output of the FSM denoted  $F_t$  is calculated as :

$$F_t = (s_{t+15} \boxplus R1_t) \oplus R2_t$$

and the output of the cipher  $z_t$  is given by :

$$z_t = F_t \oplus s_t$$

The FSM registers are updated as follows :

$$R1_{t+1} = s_{t+5} \boxplus R2_t$$

$$R2_{t+1} = S(R1_t)$$

Where  $S$  is a fixed nonlinear permutation of  $GF(2^{32})$ .

Assume we have applied a single bit fault to  $R1$ . The immediate effect on the output stream

will be a difference of  $\pm 2^j$  where  $j$  is the faulted bit. The sign of the difference will give us the actual bit of  $R1$ . By resetting the device and applying faults to the different bits of  $R1$  we can recover all of  $R1$ . Once we have recovered  $R1$  at time steps  $t$  and  $t + 2$ , i.e., we know both  $R1_t$  and  $R1_{t+1}$  we can recover a word from the LFSR through the equation:

$$R1_{t+2} = s_{t+6} \boxplus R2_{t+1} \tag{3.2}$$

and since

$$R2_{t+1} = S(R1_t) \tag{3.3}$$

Giving us

$$s_{t+6} = R1_{t+2} \boxplus (2^3 2 - S(R1_t)) \tag{3.4}$$

After recovering 16 different words from the LFSR we can reconstruct the entire internal state.

Notice that we can also use faults of higher Hamming weight. For example if we know that the Hamming weight of the faults is bounded by 2 then the observed output difference will be of the form  $\pm 2^j \pm 2^k$ . If the two faults are far enough from each other to ensure a low probability of interference between their effects, we can recover two bits of  $R_1$  from a single fault. Notice that we can check whether the faults have interfered with each other by checking if there is a unique solution of the above form.

# Chapter 4

## Attacks on Other Real Life Stream Ciphers

### 4.1 An Attack on Scream

Scream is a stream cipher designed by Halevi, Coppersmith & Jutla in [5]. There are three versions of the cipher: Scream-0, Scream-S & Scream-F<sup>1</sup>. The Scream-0 and Scream-F variations are equivalent from the viewpoint of the attack and hence we will refer only to Scream-0. The only difference between Scream-0 and Scream-S is that the S-boxes used are in Scream-0 are fixed while they are key dependent in Scream-0. The best direct attack against Scream-0 [29] has data complexity of  $2^{43}$  space complexity of  $2^{50}$  and time complexity of  $2^{80}$ .

The state variables are initialized during the key and nonce setup. Since  $F$  is invertible, the state transition is also invertible. We will use this fact to run the cipher backwards and recover the key once we have recovered the internal state. The key and nonce setup which are not described here is also invertible if one has knowledge of  $x, z, W[1]$  and the nonce right after the nonce-setup.  $G$  is a scaled down version of the AES round function and has two parameters besides its input; a  $2 \times 2$  invertible matrix (over  $GF(2^8)$ ) and a S-box permuting  $GF(2^8)$ . For the purpose of this attack it suffices to say that the two matrices  $M_1, M_2$  are invertible and not key-dependent. The two S-boxes  $S_1, S_2$  are invertible and for Scream-0 & Scream-F are not key-dependent.

#### 4.1.1 The Basic Attack

Now we will describe an attack against Scream-0 and Scream-F, the attack against Scream-S will be described later in this section. The attack against Scream-0 and Scream-F applies

---

<sup>1</sup>All diagrams in this section were taken from [5]

State:  $x$ ;  $y$ ;  $z$  ( three 16-byte blocks)  
 $W$  ( a table of 16 16-byte blocks)  
 $i_w$  ( an index into  $W$  (initially  $i_w = 0$ )

1. repeat(until you get enough output bytes)
2.     for  $i = 0$  to 15 // generate the next 16 output blocks
3.          $x := F(x \oplus y)$  // modify the evolving state" x
4.          $x := x \oplus z$
5.         output  $x \oplus W[i \bmod 16]$
6.         rotate  $y$  // the exact rotation depends on  $i$
7.     end-for
8.      $y := F(y \oplus z)$  // modify  $y$ ;  $z$ , and  $W[i_w]$
9.      $z := F(z \oplus y)$
10.      $W[i_w] := F(W[i_w])$
11.      $i_w := i_w + 1 \bmod 16$
12. end-repeat

Figure 4.1: The main loop of Scream and Scream-0

single bit faults and recovers the values of the internal variables at specific points. Then using this information and knowledge of the nonce used, the attacker can recover the seed.

### 4.1.2 Detecting in which variable the fault occurred

Since the attack model does not assume control over the location of the fault, in order to exploit the faults we apply we need to know where they occurred. If the fault occurred in the  $W$  table, we will see a single bit difference in the corresponding 16-byte block of the output stream and then 15 blocks with zero difference. If the fault occurred in the  $z$  variable we will see a single bit difference in corresponding block in the output stream followed by significant difference in the following output blocks. If the fault occurred in the  $x$  or  $y$  variables then the corresponding output block will have a significant difference and from the difference pattern we can identify which bit was flipped. All the above observations are a direct result of the structure of  $F$ .

### 4.1.3 Identifying where in the variable the fault occurred

In the case that  $z$  or the  $W$  table were affected by the fault we can identify which bit was flipped by simply looking at the output difference. But how do we identify which bit was flipped if  $x$  or  $y$  where affected ? If we follow the progression of the difference though the  $F$  function we can see that depending on the byte in which the fault occurred there is a different pattern of bytes in the block which have non-zero differences. Thus we can identify

Figure 4.2: The G and F functions

in which byte the fault occurred out of the 16 bytes that compose the variable.

#### 4.1.4 Recovering the input to the $F$ Function

A basic step in the attack is applying faults to recover the input to the  $F$  function at a certain point in the stream. Suppose we apply a single bit fault to the input to the  $F$  function in step 3 of the algorithm. Following through the propagation of the difference we see that we can detect from the output difference in step 5 the input byte that was effected by the fault. Now by applying the inverse of the matrix  $m_1$  or  $m_2$  (depending on which input byte was faulted) to the output difference, we can recover the difference at the output of the S-box. We now know two things:

1. The input difference to the S-box has a Hamming weight of one.
2. The output difference from the S-box.

There are on the average two possible input values that satisfy these constraints. Therefore after collecting the data from two faults to the same input byte we can reconstruct the byte.

#### 4.1.5 The actual attack

Assume that  $F$  is known (as in Scream-0 and Scream-F) and additionally we know the values of  $z,x$  and  $W[1]$  right after the nonce-setup, in this case we can recover the key by inverting the nonce-setup and key-setup. Thus in order to recover the key we only need the values of  $z,x$  and  $W[1]$  right after the nonce-setup. A basic step in our attack will be to apply a fault before the  $F$  function and observe the difference in the output. From this we reconstruct the input to the  $F$  function (and therefore gain knowledge of the output as well) at this stage.

Faults applied before step 9 will give us the value of  $z$  for the next round. Faults applied to  $x, y$  or  $z$  before step 3 in the following round will give us  $x$  and  $y$ . Once we have recovered  $x, y$  &  $z$  we can advance the state of  $x, y$  and  $z$  and xor it with the stream from the cipher to retrieve the  $W$  table. Now we run the cipher backwards to recover the values of  $x, z$  and  $W[1]$  right after the nonce-setup.

#### 4.1.6 An Attack Against Scream-S

The problem with applying the previous attack to Scream-S is that the S-boxes in the  $F$  function are key-dependent and therefore unknown. Therefore, we first conduct a fault attack to recover the S-boxes and then apply the basic attack. First let's examine what information we get by applying a 8 different faults to the 8 bits of one of the inputs to the S-boxes. Using the techniques shown in the previous sections we get the 8 differences in the output of the S-boxes for the specific input. This can be viewed as a 'fingerprint' of the input. Whenever we observe this 'fingerprint' of differences we know that the input byte was the same. Now assume we have collected 256 different 'fingerprints' we can find their relative relations by comparing the differences. In a similar manner we can relate the possible outputs. What is left to reconstruct the S-boxes is the actual input and output of one value. Since there is no further information in the differences we simply guess these 16 bits and for each guess reconstruct the key according to the basic attack and check whether the generated stream is correct. Notice that it is usually not necessary to apply further faults to apply the basic attack since enough data has been gathered during the 'fingerprint' building stage.

#### 4.1.7 Fault Identification

In light of the previous discussion it would be of interest to find a scheme which could detect (single bit) faults inflicted during the operation of Scream. It is rather straightforward to detect faults applied to the internal variables (excluding  $x$ ) by calculating a byte based checksum since during the production of 256 bytes of output this checksum does not change (the changes applied to  $y$  are only permutations and do not effect the checksum. Detecting faults in  $x$  is more difficult but if we assume that the attacker does not have accurate control over the location of the fault, then inadvertently the attacker will effect some variable other than  $x$ . The additional processing required is calculating the checksum twice every 256 bytes. Once to calculate a reference checksum and once to check for faults.

## 4.2 An Attack on RC4

RC4 is a stream cipher designed by Ron Rivest in 1987. Its source code was kept as a trade secret until an alleged version was posted to the Cyberpunks mailing list [8]. RC4 consists of a key scheduling which initializes the permutation  $S$ , initialization and a generation loop. The key schedule will not be of interest for our attack.

Initialization:

$$i = 0$$

$$j = 0$$

Generation Loop:

$$i = i + 1$$

$$j = j + S[i]$$

Swap  $S[i]$  and  $S[j]$

Output  $S[S[i]+S[j]]$

Figure 4.3: Pseudo-code for RC4

The best direct attack on RC4[30] is based of a 'Guess on Demand' approach and has a time complexity of more than  $2^{700}$ .

Our attack will proceed in three stages:

1. Apply a fault to the  $S$  table and generate a long stream (repeat many times)
2. Analyze the resulting streams and generate equations in the original entries of  $S$
3. Solve these equations to reconstruct  $S$ .

We assume that the attacker can fault a single entry of the  $S$  table immediately after the key-scheduling. Our first observation is that the attacker can recognize which value was faulted. I.e., if  $S[x] = a$  and the fault changed its value to  $b$  then we will identify both  $a$  and  $b$  (but not  $x$ ). This can be done by observing the frequency of each symbol in the output stream. If  $a$  was changed to  $b$  then  $a$  will never appear in the output stream, while  $b$  will appear with double frequency. Thus we need a stream of length about 10,000 bytes to reliably identify  $a$  and  $b$ . Our next mission is to identify faults in  $S[1]$ . This is done by looking at the first output byte. If this byte changed as a result of the fault then one of three cases must hold:

1.  $S[1]$  was faulted
2.  $S[S[1]]$  was faulted

3.  $S[S[1] + S[S[1]]]$  was faulted

We know what the original value of  $S[S[1] + S[S[1]]]$  was so we can check if the fault affected this cell (by identifying  $a$  and  $b$ ). If we fault  $S[1]$  and can identify the fault, i.e.  $S[1]$  changed from  $a$  to  $b$ , then we know two things. First the original value of  $S[1]$  was  $a$  and second,  $S[b + S[b]] = c$  where  $c$  is the actual observed output in the faulted stream. So our first issue is how to recognize faults. If case 2 holds then with high probability the second output byte  $S[S[2] + S[S[1] + S[2]]]$  will not be faulted. If the first case holds then the second output byte will always be faulted.

---

**Algorithm 13** Attack on RC4

---

1. Produce a faulted stream and identify the fault
  2. For faults which affected  $S[1]$  write an equation involving elements of  $S$
  3. After collecting enough equations, infer the contents of  $S$  from the known entries, when an unknown entry is necessary guess it.
  4. Check the resulting recovered state for consistency with the actual output, if inconsistent change the guesses
- 

Now that we have identified a fault that affected  $S[1]$  and changed its value from  $a$  to  $b$  we know two things:  $S[1] = a$  and  $S[b + S[b]] = c$  where  $c$  is the first output byte of the faulted stream. For each fault in  $S[1]$  we get an equation, and after collecting many such equations we start utilizing our knowledge of  $S[1]$  to deduce other values in  $S$ . For example, if  $S[1] = 17$  then the equation  $S[1 + S[1]] = 7$  will give us the value of  $S[18] = 7$ . We deduce as many values as possible from the given equations. If at the end we have not recovered  $S[S[1]]$  then we guess its value. From our knowledge (guess) of  $S[S[1]]$  we can carry out an analysis of the second output byte and recover more equations, this time of the form  $S[b + S[b + S[1]]] = d$  (where  $d$  is the second output byte). Empirical results show that at this stage we recover on average 240 entries of  $S$ , and this is more than enough to deduce the rest from the observed non-faulted stream. We can easily reject incorrect guesses of  $S[S[1]]$  by either noticing an inconsistency in the equations we collect or by recovering  $S$  and comparing the output stream to the observed one.

Another interesting fault attack against RC4 was introduced by Biham et al at the CHES 2004 rump session [18]. The attack waits for a fault to cause the cipher to enter an impossible class of states called Finney states. These states, discovered by Finney in 1994, are characterized by  $S[i] = 1, j = i + 1$ . It is easy to see that the class of F-states is closed, that is, F-states always result in F-states and are only reachable from F-states. Since the

initial state of RC4 is not a F-state, this class of states is impossible to reach naturally. Notice that once the cipher is in an F-state, we can easily deduce the internal state from the output stream. Let  $S^t$  be the contents of  $S$  at time  $t$ . By following the effect of the swapping of  $S[i]$  and  $S[j]$  throughout 255 steps we can see that for each  $t$  and  $k$  we have that  $S^t[k] = S^{t+255}[k-1]$ . So by taking every 255<sup>th</sup> output byte we can immediately recover  $S$  (up to a rotation) and since the effect of 255 steps of the cipher on  $S$  is only a rotation, we can (after at most 256 guesses) recover the original state of the cipher.

It now remains to show how we can use a fault attack to induce the cipher into a F-state and how to recognize that we are in such a state. For the sake of simplicity we assume that the fault can affect only the indices  $i$  and  $j$ . The probability of the resulting state being a F-state is  $2^{-16}$  (assuming we applied the fault at a random timing). If we are not in an F-state, then according to the birthday paradox we expect that after less than 30 bytes we will see the same output byte twice. On the other hand if we are in a F-state then each of the resulting 255 streams, produced by sampling the output stream every 255 bytes, contains a repetition of the internal state and thus every 256 consecutive bytes in each such stream are distinct. This increases the number of bytes required on average to see a repetition to about 80 bytes. This observation enables us to distinguish between streams resulting from F-states and normally occurring streams.

---

**Algorithm 14** Biham et al Attack on RC4

---

1. repeat the following until the initial state is recovered
  2. initialize the cipher
  3. produce a fault
  4. produce 30 output bytes and check for repetitions
  5. if no repetitions exist, produce a long stream to recover initial state
-

# Chapter 5

## Summary

### 5.1 Summary of the Results

The complexity of the attacks in the previous chapters are summarized in the table below. For the synthetic constructions an asymptotic analysis was done while for the real life constructions the analysis was done for the recommended parameters of the ciphers. The parameters  $n, t, T, k$  and  $M$  are as defined in the relevant section. For simplicity the results for the clocking constructions assume that the length of the clocking LFSR is the same as the length of the data LFSR. For SOBER, SNOW and SCREAM, the analysis assumed the attacker had full control over the location of the fault.

Attack	#Faults	Data	Time	Space
Filtered LFSRs	$t$	$t2^t$	$\binom{n}{k}2^t + n^3$	$t2^t + n^2$
Clock controlled (faults in clock register)	$n$	$n^2$	$n^3$	$n^2$
Clock controlled (faults in data register)	1	$n$	$\binom{n}{k}n + n^3$	$n^2$
FSM filtered LFSR (totally randomized)	$nM^2$	$nM^2$	$nM^2 + n^3$	$n^2$
FSM filtered LFSR (faults in FSM)	$T^2 \frac{n}{\log M}$	$T^3 \frac{n}{\log M}$	$n^3$	$T^3 \frac{n}{\log M} + n^2$
LILI-128	10K	1M	$2^{25}$	1M
SOBER-t32	1K	100K	$2^{30}$	100K
SNOW 2.0	1K	2K	$2^{27}$	1K
SCREAM-0	100	10K	100K	10K
RC4 (our attack)	$2^{16}$	$2^{26}$	$2^{26}$	$2^{16}$
RC4 (Biham's attack)	$2^{16}$	$2^{21}$	$2^{21}$	$2^8$

Figure 5.1: Result summary

### 5.2 Further Work

In this work we have conducted a systematic study of the applicability of fault attacks to stream ciphers. For actual implementations of stream ciphers, the applicability of our attacks

depends highly on the abilities of the attacker and specifically his control over the timing and location of the induced fault. Anderson's paper shows us that the capabilities of attackers cannot be underestimated and that it is possible to achieve a high degree of control even with simple and cheap equipment. We feel that it is possible to improve upon most of our attacks and reduce the amount of faults required and the assumptions about the hamming weight of the faults. While we have successfully attacked almost every construction we considered, what I feel most lacking is a formal framework in which to analyze the security of cryptosystems against fault attacks. The existence of such a framework could provide us with the much needed tools to prove the security of the cryptosystems we use in the real world.

Along the way there were a number ideas and partial results which were not presented in this thesis. Below is a partial list of these results:

- Attacks on NLF with medium Hamming weight faults - we have made some progress in applying more efficient algorithms to identify the fault in a restricted model in which the LFSR has a small number of feedback taps and the NLF has a small number of input taps. In this setting once we see an output difference in the few first output bytes, we keep a list of the location of the input taps at the time and relate the list to the fault. Since the feedback is of low weight, we can still follow through the progression of our list of possible locations for the fault through the feedback. When two lists for different observed output differences intersect we assume that the faulted output bits were generated by the same faulted bit in two different NLF input taps.

Another idea is to rate the probability for bit to be faulted by checking the correlation between the output difference stream (for a few lengths of the LFSR) with the raw output stream produced by initializing the LFSR with only this bit. After rating each bit, we perform a limited search with the probabilities guiding the possible choices.

We were unsuccessful in conducting attacks involving high Hamming weight faults.

- Attacks on clock controlled LFSR with low Hamming weight faults - we have made some progress with faults of small weigh in the clock LFSR. The idea is to 'stitch' together output streams and try to reconstruct the original data stream by observing the first few output bits from each fault. It would be interesting to try and extend this approach to higher Hamming weight faults.
- We have tried to conduct a fault attack against the stream cipher MUGI[17] but without success. The problem with our initial approach is that the linear layer in MUGI is before the non-linear one. This means that when conducting a differential

fault attack we can only recover the xor of two values which was not enough to complete the attack.

- We have fault attacks against a block cipher (DES) in OFB mode. The idea is to change the IV used in a controlled way and then to fault the first round of the block cipher and try to counter the initial change we have made. This can be done by observing the output stream and checking whether it is identical to the original output stream. By analyzing the statistics of how often we successfully counter the difference in the IV we can recover the key.
- We have also developed a simple power analysis attack against LFSR based constructions. The attack has two versions, in the first we assume the attacks can measure the total Hamming weight of all LFSRs in the system or the difference in the Hamming weight between successive clocks. This is enough to construct linear equations in the initial state of the LFSRs. The second versions assumes we can reliably detect when the Hamming weight is above a certain threshold and below a different one. We then check the time difference between transitions from one state to the other and deduce the Hamming weight of the LFSRs raw output stream during this period. Again this is enough to construct linear equations in the initial state of the LFSRs. In both versions after collecting enough equations we can solve for the initial state of the LFSR.

# Bibliography

- [1] Ross Anderson *Optical Fault Induction*, June 2002.
- [2] Boneh, Demillo, and Lipton *On the Importance of Checking Cryptographic Prtocols for Faults*, September 1996.
- [3] Biham, Shamir *A New Cryptanalytic Attack on DES: Differential Fault Analysis*, October 1996.
- [4] E. Dawson A. Clark J. Golic W. Millan L. Penna L. Simpson *The LILI-128 Keystream Generator*, November 2000.
- [5] Shai Halevi, Don Coppersmith & Charanjit Jutla *Scream an efficient stream cipher*, June 2002.
- [6] Coppersmith, Krawczyk & Y. Mansour *The Shrinking Generator*, Proceedings of Crypto'93, pp.22–39, Springer-Verlag, 1993
- [7] Philip Hawks & Gregory G. Rose *Primitive Specification and Supporting Documentation for SOBER-t32 Submission to NESSIE*, June 2003.
- [8] Itsik Mantin & Adi Shamir *A Practical Attack on Broadcast RC4*, FSE 2001
- [9] Jovan Dj. Golic & Guglielmo Morgari *On the Resynchronization Attack*, FSE 2003
- [10] Jovan Dj. Golic & Guglielmo Morgari *Correlation Analysis of the Alternating Step Generator*, Designs, Codes and Cryptography, 31, 51-74, 2004
- [11] S. Dubuc, *Characterization of linear structures*, Designs, Codes and Cryptography, vol. 22, pp. 33-45, 2001
- [12] Nicolas Courtois and Willi Meier, *Algebraic Attacks on Stream Ciphers with Linear Feedback*, Eurocrypt 2003
- [13] Steve Babbage, *Cryptanalysis of LILI-128*, Proceedings of the 2nd NESSIE Workshop, 2001

- [14] Steve Babbage, Christophe De Canniere, Joseph Lano, Bart Preneel, Joos Vandewalle, *Cryptanalysis of SOBER-t32*, FSE 2003
- [15] Joo Yeon Cho and Josef Pieprzyk, *Algebraic Attacks on SOBER-t32 and SOBER-128*, FSE 2004
- [16] Nicolas Courtois *Algebraic Attacks on Combiners with Memory and Several Outputs*, ICISC 2004
- [17] D. Watanabe, S. Furuya, K. Takaragi and B. Preneel *A New Keystream Generator MUGI*, FSE 2002
- [18] Eli Biham, Louis Granboulan and Phong Nguyen *Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4*, FSE 2005
- [19] Paul Kocher *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, Proceedings of Crypto'96
- [20] Paul Kocher, Joshua Jaffe and Benjamin Jun *Differential Power Analysis*, Proceedings of Crypto'99
- [21] Patrick Ekdahl and Thomas Johansson *A New Version of the Stream Cipher SNOW*, [www.it.lth.se/cryptology/snow/snow2.pdf](http://www.it.lth.se/cryptology/snow/snow2.pdf)
- [22] Willi Meier and Othmar J. Staffelbach *Fast correlation attacks on stream ciphers*, Journal of Cryptology, 1(3):159-176, 1989.
- [23] Klaus Pommerening *Fourier Analysis of Boolean Maps - A Tutorial*, Preprint May 2000
- [24] Hagai Bar-El, et all *The sorcerers' Apprentice Guide to Fault attacks*, Workshop on Fault Detection and Tolerance in Cryptography, June 2004
- [25] A. Klapper and M. Goresky *Feedback Shift Registers, 2-Adic Span, and Combiners with Memory*, J. Crypt. 10 (1997), 111-147
- [26] Alex Klimov and Adi Shamir *A New Class of Invertible mappings*, CHES 2002
- [27] Nicolas Courtois, Alexander Klimov, Jacques Patarin and Adi Shamir *Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations*, Eurocrypt 2000
- [28] J.C. Faugre *A new efficient algorithm for computing grbner bases (f4)*, Journal of Pure and Applied Algebra, June 1999

- [29] Don Coppersmith, Shai Halevi & Charanjit Jutla *Cryptanalysis of stream ciphers with linear masking*,
- [30] Lars R. Knudsen, Willi Meier, Bart Preneel, Vincent Rijmen, and Sven Verdoolaege *Analysis Methods for (Alleged) RC4*, Asiacrypt 1998
- [31] Markku-Juhani and Olavi Saarinen *A Time-Memory Tradeoff Attack Against LILI-128* FSE 2002