Cryptanalysis of Two Fault Countermeasure Schemes

Subhadeep Banik(✉) and Andrey Bogdanov

DTU Compute, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
{subb,anbog}@dtu.dk

Abstract. In this paper, we look at two fault countermeasure schemes proposed very recently in literature. The first proposed in ACISP 2015 constructs a transformation function using a cellular automata based linear diffusion, and a non-linear layer using a series of bent functions. This countermeasure is meant for the protection of block ciphers like AES. The second countermeasure was proposed in IEEE-HOST 2015 and protects the Grain-128 stream cipher. The design divides the output function used in Grain-128 into two components. The first called the masking function, masks the input bits to the output function with some additional randomness and computes the value of the function. The second called the unmasking function, is computed securely using a different register and undoes the effect of the masking with random bits. We will show that there exists a weakness in the way in which both these schemes use the internally generated random bits which make these designs vulnerable. We will outline attacks that cryptanalyze the above schemes using 66 and 512 faults respectively.

Keywords: AES · Fault analysis · Grain-128 · Infective countermeasures

1 Introduction

There has been a lot of effort to design hardware based countermeasures to prevent fault attacks on AES-128 circuits. Most of these countermeasures can be classified into two broad categories: **(a)** Detection based and **(b)** Infection based. As the name suggests, detection based measures aim to detect the injection of fault by performing various intermediate checks during the course of the encryption operation [7,12,15]. The functionality is achieved by comparing two or more data blocks output by the encryption circuit. Since the comparison operation is itself prone to faults, Infection based countermeasures have also become popular [8,14,16]. In this approach, the circuit is designed in such a fashion that even if an attacker is able to inject a fault in the circuit, he can not utilize the corrupted output to find the secret key. Most of these countermeasures work by introducing additional operations in between or after the encryption algorithm that make it difficult for an adversary to deduce simple enough algebraic relations to deduce the secret key.

A. Biryukov and V. Goyal (Eds.): INDOCRYPT 2015, LNCS 9462, pp. 241–252, 2015.
DOI: 10.1007/978-3-319-26617-6_13

As already mentioned, the main philosophy behind infective countermeasures is to ensure that a faulty ciphertext produced by the system can not be exploited by the attacker to obtain any non-trivial information about the secret key. There have been several infective countermeasures proposed in literature but most of them have a hardware overhead of over 100 %. In [8], a countermeasure using redundant and dummy round functions was proposed. A countermeasure using random masks was proposed in [16]. Both these fault protection schemes were cryptanalyzed in [3]. In response, an improved countermeasure was proposed in [19], that replaced the output of the cipher with a random 128 bit string whenever the system detected a fault. This method was again cryptanalyzed in [4]. In this paper, we will look at two different infective countermeasures that have been proposed very recently. The first [9], was designed mainly to protect block ciphers like AES [6]. The design includes two identical AES modules, the outputs of which are xored and fed into a transformation function composed of the following functions

1. A linear diffusion function based on the principles of cellular automata,
2. A non-linear mixing function that additionally utilizes some internally generated randomness.

The output of the transformation function is xored back to the outputs of one of the AES modules, and produced as ciphertext. Assuming that the adversary does not have the capability to reproduce the same fault in both the AES modules, any difference introduced by a fault in one of the modules is transformed by the non-linear function so that any simple algebraic relation can not be derived between the faulty ciphertext and the roundkey. The second countermeasure proposed in [10] has been designed protect the Grain-128 [11] stream cipher. The work is significant because not many architectures have been proposed to protect stream ciphers from fault attacks. The design decomposes the output boolean function of Grain-128 into two component functions. The first called the masking function, masks the inputs to the output function with certain random bits generated internally. The second called the unmasking function, (which is computed securely using a different register) undoes the effect of the masking, so that the GF(2) sum of the masking and unmasking function equals the output function of Grain-128. We will show that method of utilizing the internal randomness in both these schemes has some weakness which can be used to cryptanalyze them.

1.1 Organization of the Paper

In Sect. 2, we will first provide a complete architectural and mathematical description of the infective countermeasure proposed in ACISP 2015 [9]. In Sect. 3, we will begin by outlining a fault attack on an unprotected AES implementation. We will then go on to reveal a weakness in the non-linear mixing function used in this scheme that makes this function easy to invert. Using this observation we will propose a method that allows the attacker to deduce the secret key using around 66 faults on average. In Sect. 4, we will provide a preliminary mathematical description of Grain-128 and the countermeasure as proposed in [10]. Thereafter in Sect. 5 we will then point out two weaknesses in the scheme. First we show, that due to a flaw in the masking function, any fault localized on a specific LFSR location reveals non-trivial information about the internal state of the cipher. This can be used to mount a state recovery attack, that reveals the entire internal state in less than 512 faults. The second weakness comes from the fact that although the design tries to protect the output function of the cipher, the NFSR update function is left completely unprotected. Using this result a fault attack using as less as 4 randomly applied faults can be mounted. Section 6 concludes the paper.

2 Countermeasure Proposed in ACISP 2015 [9]

The scheme proposed in [9] can be described as follows. The design takes the xor of the outputs of two identical AES modules and passes it through a transformation function $\mathcal{T}$. This function is composed of a sequence of two functions. The first is a cellular automata based linear diffusion function. The output of the linear diffusion function is then input to a non-linear mixing function. The mixing function additionally uses some random bits which are generated internally by a cellular automata based random number generator. The output of the mixing function is then xored back with the output of one of the AES modules and produced as ciphertext. The architecture is described pictorially in Fig. 1.

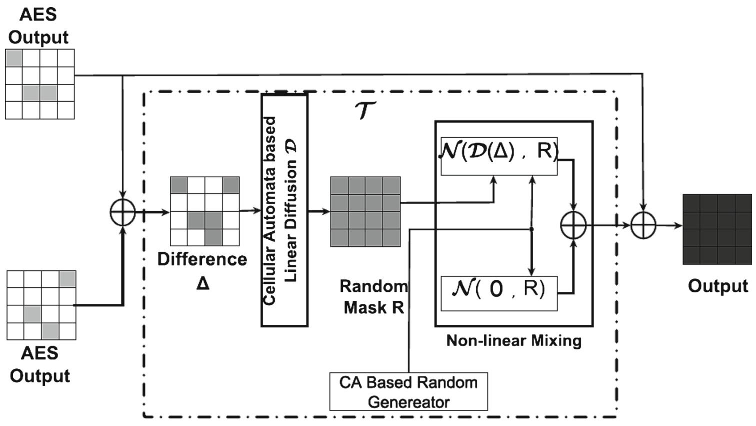
AES Output
$\mathcal{T}$
Cellular Automata based Linear Diffusion $\mathcal{D}$
$\mathcal{N}(\mathcal{D}(\Delta), R)$
Difference Δ
Random Mask R
$\mathcal{N}(0, R)$
Non-linear Mixing
Output
AES Output
CA Based Random Genereator

Fig. 1. Infective countermeasure of ACISP 2015 [9]

The architecture described above makes two assumptions. The first is that the attacker does not have the capability to inject the same fault in both the AES modules. Otherwise the output of both the AES modules is identical and so the input to $\mathcal{T}$ is zero, and since $\mathcal{T}$ maps the zero input to zero, the attacker gets back the original faulty ciphertext. The second assumption is that since the attacker does not know the random bits used to compute the output of the mixing function, it is not possible for him to deduce algebraic relations between the ciphertext and the roundkey.

2.1 Linear Diffusion Function

The linear diffusion function $\mathcal{D} : \{0,1\}^{128} \rightarrow \{0,1\}^{128}$ is based on the principles of a 3-neighborhood cellular automata. The state update operation in such a system can be expressed equivalently as pre-multiplication with a 128×128 binary matrix. In this specific case, the function $\mathcal{D}$ is constructed as follows. One iteration of the automata is first designed using the following primitive polynomial over GF(2):

$$p(x) = x^{128} + x^{29} + x^{27} + x^2 + 1$$

Thus, if X_t and X_{t+1} denote the 128 bit vectors that are input and output respectively of a single iteration of the automata, then these vectors are related as $X_{t+1} = A \cdot X_t$, where A is a 128×128 binary tridiagonal matrix whose ij^{th} element a_{ij} is given as follows:

$$a_{ij} = \begin{cases} m_i, & \text{if } i = j, \\ 1, & \text{if } |i-j| = 1, \\ 0, & \text{otherwise.} \end{cases}$$

The element a_{ii} in the principal diagonal of the matrix A is taken as the i^{th} element m_i of the vector M defined as follows:

$$\begin{aligned} M =[\ & 0,1,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,1,1,1,1,1,0,1,1,1,1,0,1,0, \\ & 1,1,0,0,1,1,1,0,0,0,0,0,0,1,1,0,0,0,1,1,0,1,0,0,1,1,1,1,0,1,0,0,1, \\ & 1,1,1,0,1,0,0,1,1,1,1,0,0,1,1,1,0,0,0,0,0,0,1,1,1,0,0,1,1,0,1,0,1, \\ & 1,1,1,0,1,1,1,1,1,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,1,0\ ] \end{aligned}$$

The vector M is constructed using the polynomial $p(x)$ by following the methods outlined in [5]. The update function described by the single iteration of this automata has a period of $2^{128} - 1$, and achieves full diffusion of any bit difference in 127 iterations. The function $\mathcal{D}$ is constructed using 255 iterations of the automata, i.e., $\mathcal{D}(X) = A^{255} \cdot X$. However, the matrix A is invertible and hence the function $\mathcal{D}$ is efficiently invertible.

2.2 Non-linear Mixing Function

The non-linear mixing function $\mathcal{N} : \{0,1\}^{128} \times \{0,1\}^{128} \rightarrow \{0,1\}^{128}$ is a constructed using a series of bent functions. The function uses a 128 bit random string R which is generated using a cellular automata based random number generator Given $\mathbf{X} = [x_0, x_1, \ldots, x_{127}]$, $\mathbf{R} = [r_0, r_1, \ldots, r_{127}]$, $\mathcal{N}(\mathbf{X}, \mathbf{R}) = \mathbf{Y} = [y_0, y_1, \ldots, y_{127}]$ is defined as:

$$c_i = \bigoplus_{j=0}^{i} x_j r_j \oplus x_{i-1} x_i \oplus r_{i-1} r_i$$

$$y_i = x_i \oplus r_i \oplus c_{i-1}$$

for $0 \leq i \leq 127$, with $x_{-1} = r_{-1} = 0$ with $c_{-1} = c_{127}$. Each y_i is a bent function of algebraic degree 2 and nonlinearity $2^{2i+1} - 2^{i+2}$. Since the fault protection mechanism must output the original ciphertext if no fault is injected, the transformation function $\mathcal{T}$ must map the zero input to zero. For this reason, the output of the nonlinear layer is taken as $\mathcal{S}(\mathbf{X}) = \mathcal{N}(\mathbf{X}, \mathbf{R}) \oplus \mathcal{N}(\mathbf{0}, \mathbf{R})$.

3 Cryptanalysis of the Fault Countermeasure Scheme

3.1 Basic Fault Attack on Unprotected AES

We outline the basic fault attack on AES which finds the secret key by injecting a random byte fault before the 9th round MixColumn (MC) operation [13, Chap. 4.2]. Assuming that a random byte fault has been injected in the first element of a column, the attacker computes a list $\mathcal{L}$ of possible differences at the output column of the MixColumn operation. The list $\mathcal{L}$ thus contains 4×255 four-byte elements. This is a one time operation. As can be seen in Fig. 2, a fault injected in the first byte of the AES state before the 9th round MixColumn will result in a faulty ciphertext that differs with the original ciphertext in byte positions $1, 8, 11, 14$. So given a pair of fault-free and faulty ciphertexts C, C_f, the attacker guesses 4 bytes of the 10th roundkey K_{10} (i.e. the 1st, 8th, 11th and 14th bytes) and computes the four differences Δ_i $(i = 1, 8, 11, 14)$ as follows:

$$\Delta_i = SB^{-1}\left(C[i] \oplus K_{10}[i]\right) \oplus SB^{-1}\left(C_f[i] \oplus K_{10}[i]\right),$$

(note that the i^{th} byte of any block X is represented as $X[i]$). Each tuple $(\Delta_1, \Delta_8, \Delta_{11}, \Delta_{14})$ is then compared with the elements contained in the list $\mathcal{L}$. The candidates $(K_{10}[1], K_{10}[8], K_{10}[11], K_{10}[14])$ for which a match is found are gathered in another list $\mathcal{E}$. With one pair (C, C_f), the list $\mathcal{E}$ contains 1,036 elements on average. By using another pair (C, C_f) with a fault injected into the same column, the corresponding four bytes of the last round key are uniquely

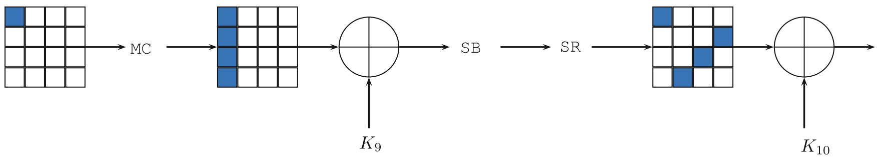

Fig. 2. Basic Attack on unprotected AES

determined with a 98 % probability. Similar analysis holds for faults injected into the 2nd, 3rd and 4th columns. Therefore the last round key can be recovered by using eight faulty ciphertexts with faults induced at chosen locations.

3.2 Cryptanalysis of the Scheme

The fault protection scheme as outlined in [9] can be outlined as follows. The scheme first computes the difference Δ in the two ciphertexts, the first of which is produced by a fault in one of the AES modules, and the second produced by the other AES module which is fault-free. This difference Δ is then passed through the transformation function $\mathcal{T}$ which is basically the composition of the functions $\mathcal{S} \circ \mathcal{D}$. The transformation function is probabilistic since it uses some randomness generated internally by the system. Now instead of $C_f = C \oplus \Delta$, the system outputs $C \oplus \mathcal{T}(\Delta)$. Now intuitively it is clear that $\mathcal{T}$ needs to be a one way function, because if the attacker can obtain the value of Δ from the value of $\mathcal{T}(\Delta)$, he can compute the value of $C_f = C \oplus \Delta$ and perform the attack described in Sect. 3.1. In this section we will prove that $\mathcal{T}$ is not a one way function due to which the security of the scheme collapses. Note that we have already established that $\mathcal{D}$ is efficiently invertible, and hence if we can show that $\mathcal{S}$ is also invertible, we would have proven that $\mathcal{T}$ is not one way.

Lemma 1. *$\mathcal{S}$ is not a one way function.*

Proof. For any fixed input $\mathbf{X} = [x_0, x_1, \ldots, x_{127}]$, $\mathcal{S}(\mathbf{X})$ is queried a few times. Since the function $\mathcal{S}$ uses internally generated randomness, it outputs a different value every time. The task therefore would be to recover $\mathbf{X}$ from $\mathcal{S}(\mathbf{X}) = \mathcal{N}(\mathbf{X}, \mathbf{R}) \oplus \mathcal{N}(\mathbf{0}, \mathbf{R})$ for different values of $\mathbf{R}$. From the description given in Sect. 2.2, the algebraic relation between $\mathcal{S}(\mathbf{X}) = [s_0, s_1, \ldots, s_{127}]$, $\mathbf{X} = [x_0, x_1, \ldots, x_{127}]$, $\mathbf{R} = [r_0, r_1, \ldots, r_{127}]$ is given as:

$$\begin{aligned}
s_0 &= x_0 + \bigoplus_{j=0}^{127} r_j x_j + x_{126} x_{127} \\
s_1 &= x_1 + r_0 x_0 \\
s_2 &= x_2 + r_0 x_0 + r_1 x_1 + x_0 x_1 \\
s_3 &= x_3 + r_0 x_0 + r_1 x_1 + r_2 x_2 + x_1 x_2 \\
s_4 &= x_4 + r_0 x_0 + r_1 x_1 + r_2 x_2 + r_3 x_3 + x_2 x_3 \\
&\vdots \\
s_i &= x_i + \bigoplus_{j=0}^{i-1} r_j x_j + x_{i-2} x_{i-1}
\end{aligned}$$

Define the sequence w_i as follows:

$$w_i = \begin{cases} s_1, & \text{if } i = 0, \\ s_i + s_{i+1}, & \text{if } 0 < i < 127, \\ s_{127} + s_0, & \text{if } i = 127, \end{cases}$$

It can be checked that $w_0 = x_1 + r_0x_0$, $w_1 = x_1 + x_2 + x_0x_1 + r_1x_1$ and $w_i = x_i + x_{i+1} + x_{i-1}x_i + x_{i-2}x_{i-1} + r_ix_i$, for all $i > 1$. Note that

$$w_0 = x_1 \ \textbf{ if } \ x_0 = 0, \ \ \textbf{and} \ \ w_0 = x_1 + r_0 \ \textbf{ if } \ x_0 = 1.$$

Now, if we compute the value of w_0 for different outputs of the function $\mathcal{S}(\mathbf{X})$ (which are generated for different values of the internal random string $\mathbf{R}$), then the value of w_0 will be a constant and equal to x_1 if and only if $x_0 = 0$. If $x_0 = 1$, then $w_0 = x_1 + r_0$, and w_0 will evaluate to a different bit value each time, depending on the value of the random bit r_0. This argument can be extended to any i. If and only if $x_i = 0$, $w_i = x_i + x_{i+1} + x_{i-1}x_i + x_{i-2}x_{i-1}$ and will thus evaluate to a constant for every query. If $x_i = 1$, then the value of w_i has linear dependence on the random bit r_i and is more or less uniformly randomly distributed over the set $\{0, 1\}$. So our algorithm to invert $\mathcal{S}$ is as follows. Query the function $\mathcal{S}$ for a fixed $\mathbf{X}$ around N times. Then compute the vector $W = [w_0, w_1, \ldots, w_{127}]$ for each query $\mathcal{S}(\mathbf{X})$. If w_i evaluates to the same value over all the queries, then we conclude that $x_i = 0$, otherwise we conclude that $x_i = 1$. Computer simulations have confirmed that $N = 8.3$ queries on average are required to fully determine the value of $\mathbf{X}$.

Corollary 1. *$\mathcal{T}$ is not a one way function.*

Proof. Since $\mathcal{T}^{-1} = \mathcal{D}^{-1} \circ \mathcal{S}^{-1}$, and we have established that $\mathcal{S}^{-1}$ and $\mathcal{D}^{-1}$ are both efficiently calculable, we can conclude as such.

Fault Attack on the Scheme: We have just established that the function $\mathcal{T}$ is invertible, if one ensure that the same input is fed to the non-linear function around 8 times. So the attacker proceeds as follows:

1. He first obtains the fault-free ciphertext from the device.
2. He resets the device and applies a fault in the 1st column of the AES state before the 9^{th} round MixColumn and obtains the faulty ciphertext $C + \mathcal{T}(\Delta)$.
3. He repeats the process 8 times, and each time he applies the same fault in the device. This ensures that the input to the nonlinear function $\mathcal{S}$ is the same each time. Note that this can be achieved by using optical fault [18] to flip the logic at a particular register location during each fault injection process.
4. The attacker uses the procedure outlined in Lemma 1 and Corollary 1, to obtain the value of Δ.
5. He then uses the attack outlined Sect. 3.1 to deduce 4 bytes of the 10th roundkey. This requires another fault-free and faulty ciphertext pair with fault in the same column and so the Steps 1–4 need to executed once more.
6. The process is repeated for the 2nd, 3rd and 4th columns of the AES state to obtain the full roundkey.

Fault Complexity: Since finding 4 bytes of the 10th roundkey, requires around $2 \times 8.3 = 16.6$ faults on average, the entire roundkey can be deduced in $4 \times 16.6 \approx 66$ faults on average.

4 Countermeasure Proposed in HOST 2015 [10]

Before we proceed to describe the infective countermeasure proposed in [10], we will give a short mathematical description of the Grain-128 stream cipher. Grain-128 consists of a 128 bit LFSR and a 128 bit NFSR. The state is initialized with a 128 bit Key which is loaded on to the NFSR and a 96 bit IV and a 32 bit pad $P =$ 0x `ffff ffff` which is loaded on to the LFSR. The LFSR state is update according to the rule:

$$y_{t+128} \overset{\Delta}{=} f(Y_t) = y_{t+96} + y_{t+81} + y_{t+70} + y_{t+38} + y_{t+7} + y_t.$$

The NFSR state is updated as follows

$$x_{t+128} = \; y_t + g(X_t), \quad \text{where}$$

$$\begin{aligned} g(X_t) = {} & x_t + x_{t+26} + x_{t+56} + x_{t+91} + x_{t+96} + x_{t+3}x_{t+67} + x_{t+11}x_{t+13} + \\ & x_{t+17}x_{t+18} + x_{t+27}x_{t+59} + x_{t+40}x_{t+48} + x_{t+61}x_{t+65} + x_{t+68}x_{t+84} \end{aligned}$$

The output keystream bit z_t in some round t is produced as

$$\sum_{j \in A} x_{t+j} + y_{t+93} + h(x_{t+12}, y_{t+8}, y_{t+13}, y_{t+20}, x_{t+95}, y_{t+42}, y_{t+60}, y_{t+79}, y_{t+95})$$

where $A = \{2, 15, 36, 45, 64, 73, 89\}$ and $h(s_0, \ldots, s_8) = s_0s_1 + s_2s_3 + s_4s_5 + s_6s_7 + s_0s_4s_8$. The cipher is executed for 256 rounds without producing any output, during which the output bit z_t is fed back to the update functions of the LFSR and NFSR. Thereafter the feedback is discontinued and the cipher starts producing output.

4.1 Fault Protection Scheme in [10]

The fault protection scheme of [10] can be described as follows. The output function h used in Grain-128 is decomposed into two component functions: a masking function h_{masked} and an unmasking function $\mathcal{M}$. The function h_{masked} is computed as follows: a nine bit random string $\epsilon_0, \epsilon_1, \ldots, \epsilon_8$ is generated by an internal mechanism and then the function is computed as follows:

$$\begin{aligned} h_{masked} = {} & (s_0 + \epsilon_0)(s_1 + \epsilon_1) + (s_2 + \epsilon_2)(s_3 + \epsilon_3) + (s_4 + \epsilon_4)(s_5 + \epsilon_5) + \\ & (s_6 + \epsilon_6)(s_7 + \epsilon_7) + s_0s_4(s_8 + \epsilon_8) + (s_0 + s_4)s_8\epsilon_8 \end{aligned}$$

The unmasking function $\mathcal{M}$ is computed so that $h = h_{masked} + \mathcal{M}$. The function $\mathcal{M}$ is computed securely via a different 128 bit register which stores the values of $\mathcal{M}$ for 128 consecutive iterations. The process is described pictorially in Fig. 3.

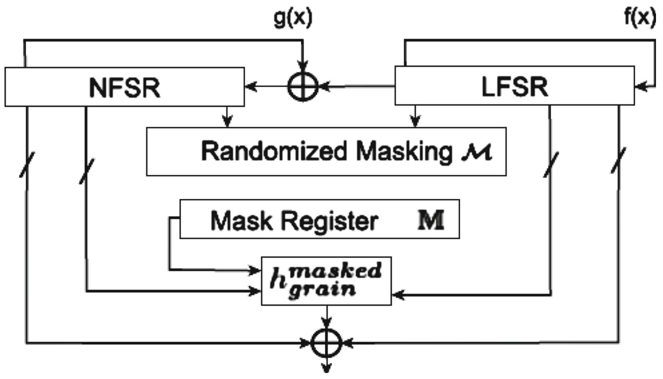

Fig. 3. Fault protection scheme in HOST 2015 [10]

5 Cryptanalysis of the Fault Countermeasure Scheme

5.1 Fault Attack on Unprotected Grain-128

There have been several fault attacks on Grain-128 reported in literature [1,2,17]. The basic philosophy in these attacks is the same. The attacker exploits the low algebraic degree of the output function h. For example if the attacker applies an optical fault which flips the logic at the bit denoted by the input variable s_2, then the difference between the fault-free keystream bit z and the faulty keystream bit z^f is given as

$$z + z^f = h(s_0, s_1, s_2, \ldots, s_8) + h(s_0, s_1, 1 + s_2, \ldots, s_8) = s_3$$

This therefore leaks the value of one state bit of the internal state of Grain-128. By applying several faults an studying the faulty keystream patterns, the attacker can easily determine the entire internal state of Grain-128.

However if one uses the fault protection scheme proposed in [10], then since the value of the unmasking function is computed securely, the difference between the faulty and fault-free keystream bit is given as:

$$z + z^f = h_{masked}(s_0, s_1, s_2, \ldots, s_8) + h_{masked}(s_0, s_1, 1 + s_2, \ldots, s_8) = s_3 + \epsilon_3$$

Since ϵ_3 is an internally generated random bit which the attacker does not know, this prevents the leakage of state information. However we will demonstrate that there exists two weaknesses in this scheme which still allows the attacker to determine the values of the internal state bits.

5.2 First Weakness

If the attacker faults the input bit s_8 (which corresponds to the 95th bit of the LFSR), then the difference between the faulty and fault-free keystream bit is given as

$$\begin{aligned} z + z^f &= h_{masked}(s_0, s_1, s_2, \ldots, s_8) + h_{masked}(s_0, s_1, s_2, \ldots, 1 + s_8) \\ &= s_0 s_4 + (s_4 + s_0)\epsilon_8 \end{aligned}$$

So the attacker proceeds as follows:

- He obtains the fault-free keystream bit z_t in some round t.
- He resets the device and injects a fault in the 95th LFSR bit (s_8) in the round t, and obtains the faulty bit z_t^f.
- He repeats the process N times so that he accumulates N different values of the keystream difference $z_t + z_t^f$.

Now if and only if $s_4 + s_0 = 0$, the value of $z_t + z_t^f = s_0 s_4$, and thus the above process will yield the same value of $z_t + z_t^f$ for each new fault injection. However if $s_4 + s_0 = 1$, then the value of $z_t + z_t^f$ is more or less uniformly randomly distributed over the set $\{0, 1\}$. Thus a fault in the bit s_8 leaks additional information about the internal state. If the attacker is able to execute this process for $0 \le t < \tau$ number of keystream rounds, then this leaks the following information about the state

1. It reveals the value of $s_0 + s_4$ in every round t.
2. If $s_0 + s_4 = 0$ in some round, it additionally leaks the value of $s_0 s_4$ in that round.

Armed with this information the attacker can proceed with the fault attack as follows. He creates an equation bank containing the following equations in the internal state variables for every round $0 \le t \le \tau$:

A. He adds an equation for the fault-free keystream bit z_t:

$$z_t = \sum_{j \in A} x_{t+j} + y_{t+93} + h(x_{t+12}, y_{t+8}, y_{t+13}, \ldots, y_{t+95})$$

B. He adds an equation for the value of the bit $s_0 + s_4$ at each round t (denote this bit value by the term a_t).

$$a_t = x_{t+12} + x_{t+95}$$

C. If $s_0 + s_4 = 0$ at any round t (i.e. if $a_t = 0$), he additionally adds an equation for $s_0 s_4$ (denote this bit value by the term b_t).

$$b_t = x_{t+12} \cdot x_{t+95}$$

The above equation bank is fed to a suitable equation solver which tries to determine the value of the internal state bits. In our experiments, we used the Cryptominisat-2.8 SAT Solver, which determined the solution of the above system in around 0.2 s on average on a system running on a 2.5 GHz processor and 16 GB internal memory for $\tau = 256$.

Fault Complexity: For each round t, we require around $N = 2$ faults on average to determine the value of $s_0 + s_4$. Since a total of $\tau = 256$ keystream rounds are used, the total number of faults required is around $\tau \times N = 512$.

5.3 Second Weakness

The second weakness of the fault protection scheme arises from the fact that the designers make no effort to protect either the NFSR update function g or the seven additional bits from the NFSR that are xored to the function h to produce the output keystream bit. Any random fault applied in the NFSR would therefore propagate along the NFSR through the update function g and the since the seven NFSR bits are unprotected, if the differential introduced by the fault appears on one of these bits they may reveal non-trivial information about the internal state bits. In fact in the work presented in [17], the attacker can apply random faults in the NFSR and by constructing an equation bank for every faulty and fault-free keystream bit, the attacker is able to find the entire internal state of Grain-128 in 5–6 faults by using a SAT based solver within 6 min on average. It is clear that the countermeasure scheme does not counteract the attack presented in [17]. Hence, in order for the scheme in [10] to be secure, it not only must design a better masking function h_{masked}, it must also take steps to protect **(a)** the NFSR update function g and **(b)** the seven NFSR bits that are added to the h function to produce the output keystream bit.

6 Conclusion

In this paper, we looked at the security of two fault countermeasure schemes proposed very recently in literature and proposed attacks on them requiring 66 and 512 faults respectively. We conclude that in both the schemes, the manner in which the designs use the internally generated random bits, make them vulnerable to attack. Additionally, the countermeasure used to protect Grain-128 is simply inadequate since no effort is made to protect the NFSR update function g or the seven additional bits that are xored to the output function h. From the discussion it is evident that the transformation function used in the first scheme needs to be a one way function, failing which the scheme would not provide any security. In the second scheme, not only must a better masking scheme be designed, but some additional effort must be expended to protect the other critical components of the circuit.

References

1. Banik, S., Maitra, S., Sarkar, S.: A differential fault attack on the grain family of stream ciphers. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 122–139. Springer, Heidelberg (2012)
2. Banik, S., Maitra, S., Sarkar, S.: A differential fault attack on the grain family under reasonable assumptions. In: Galbraith, S., Nandi, M. (eds.) INDOCRYPT 2012. LNCS, vol. 7668, pp. 191–208. Springer, Heidelberg (2012)

3. Battistello, A., Giraud, C.: Fault analysis of infective AES computation. In: FDTC 2013, pp. 101–107. IEEE Computer Society (2013)
4. Battistello, A., Giraud, C.: Fault cryptanalysis of CHES 2014 symmetric infective countermeasure. IACR Cryptology ePrint Archive. http://eprint.iacr.org/2015/500.pdf
5. Catell, K., Muzio, J.C.: Synthesis of one-dimensional linear hybrid cellular automata. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **15**(3), 325–335 (1996)
6. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer-Verlag, Berlin (2002)
7. Genelle, L., Giraud, C., Prouff, E.: Securing AES implementation against fault attacks. In: FDTC 2009, pp. 51–62 (2009)
8. Gierlichs, B., Schmidt, J.-M., Tunstall, M.: Infective computation and dummy rounds: fault protection for block ciphers without check-before-output. In: Hevia, A., Neven, G. (eds.) LatinCrypt 2012. LNCS, vol. 7533, pp. 305–321. Springer, Heidelberg (2012)
9. Ghosh, S., Saha, D., Sengupta, A., Roy Chowdhury, D.: Preventing fault attacks using fault randomization with a case study on AES. In: Foo, E., Stebila, D. (eds.) ACISP 2015. LNCS, vol. 9144, pp. 343–355. Springer, Heidelberg (2015)
10. Ghosh, S., Roy Chowdhury, D.: Preventing fault attack on stream cipher using randomization. In: 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 88–91 (2015)
11. Hell, M., Johansson, T., Meier, W.: A stream cipher proposal: Grain-128. In: IEEE International Symposium on Information Theory (ISIT 2006) (2006)
12. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: securing hardware against probing attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Heidelberg (2003)
13. Joye, M., Tunstall, M. (eds.): Fault Analysis in Cryptography. Information Security and Cryptography. Springer, Berlin (2012)
14. Joye, M., Manet, P., Rigaud, J.B.: Strengthening hardware AES implementations against fault attacks. IET Inf. Secur. **1**, 106–110 (2007)
15. Karpovsky, M., Kulikowski, K., Taubin, A.: Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard. In: International Conference on Dependable Systems and Networks (DSN 2004), pp. 93–101. IEEE Computer Society (2004)
16. Lomné, V., Roche, T., Thillard, A.: On the need of randomness in fault attack countermeasures - application to AES. In: FDTC 2012, pp. 85–95. IEEE Computer Society (2012)
17. Sarkar, S., Banik, S., Maitra, S.: Differential fault attack against grain family with very few faults and minimal assumptions. IEEE Trans. Comput. **64**(6), 1647–1657 (2015)
18. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523. Springer, Heidelberg (2003)
19. Tupsamudre, H., Bisht, S., Mukhopadhyay, D.: Destroying fault invariant with randomization. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 93–111. Springer, Heidelberg (2014)