Key Recovery from State Information of Sprout: Application to Cryptanalysis and Fault Attack

Subhamoy Maitra, Santanu Sarkar, Anubhab Baksi, Pramit Dey

Indian Statistical Institute, Kolkata
and Indian Institute of Technology Madras,
and National Institute of Technology, Karnataka, Surathkal
subho@isical.ac.in,sarkar.santanu.bir@gmail.com,anubhab91@gmail.com,
pramitdey@yahoo.com

Abstract. Design of secure light-weight stream ciphers is an important area in cryptographic hardware & embedded systems and a very recent design by Armknecht and Mikhalev (FSE 2015) has received serious attention that uses shorter internal state and still claims to resist the time-memory-data-tradeoff (TMDTO) attacks. An instantiation of this design paradigm is the stream cipher named Sprout with 80-bit secret key. In this paper we cryptanalyze the cipher and refute various claims. The designers claim that the secret key of Sprout can not be recovered efficiently from the complete state information using a guess and determine attack. However, in this paper, we show that it is possible with a few hundred bits in practical time. More importantly, from around 850 key-stream bits, complete knowledge of NFSR (40 bits) and a partial knowledge of LFSR (around one third, i.e., 14 bits); we can obtain all the secret key bits. This cryptanalyzes Sprout with 2^{54} attempts (considering constant time complexity required by the SAT solver in each attempt, which is around 1 minute in a laptop). This is less than the exhaustive key search. Further, we show how related ideas can be employed to mount a fault attack against Sprout that requires around 120 faults in random locations (20 faults, if the locations are known), whereas the designers claim that such a fault attack may not be possible. Our cryptanalytic results raise quite a few questions about this design paradigm in general that should be revisited with greater care.

Keywords. Cryptanalysis, Fault Attack, Key-stream, Sprout, Stream Cipher.

1 Introduction

Very recently a new paradigm for light-weight stream cipher design has been explored by Armknecht and Mikhalev [3] that uses shorter internal state. Even with the shorter internal state, this design claims to resist the time-memory-data-tradeoff (TMDTO) attacks. One interesting feature of this design is that, unlike the popular stream ciphers, it uses the secret key bits during the pseudorandom bit generation too (in general the secret key is only used in the initial key scheduling phase). In this direction, a specific stream cipher, called Sprout is presented that uses 80-bit secret key and 70-bit initialization vector. This [3] is a very interesting idea to design stream ciphers with shorter internal states. The main difference in operational part of such design with the traditional ones is to use the secret key during the Pseudo-Random Generation Algorithm (PRGA). In traditional stream cipher designs, the key and IVs are loaded in the cipher state during the Key Loading Algorithm (KLA) and then the cipher is run sufficient number of rounds without generating any key-stream bit during the Key Scheduling Algorithm (KSA). It is believed that after the KSA, the cipher state reaches a considerably random looking configuration. Then the Pseudo-Random Generation Algorithm (PRGA) routine is executed and in that phase the secret key bits are generally not used again (only the state bits are used). The design of Sprout, motivated towards design with relatively shorter state and still resisting the TMDTO attack, considers using the secret key bits (stored in a separate register) during the PRGA too. Sprout considers several building blocks from the previous designs of Grain family [1,2,12,13].

While a new paradigm of design is quite interesting, the success of the design depends on how an actual instantiation of such idea can successfully resist cryptanalytic attempts. This is the reason we look at Sprout in detail. We specifically note the following claims by the designers of Sprout [3] and refute those.

Guess and Determine Attacks. The designers make certain arguments to claim that efficiently recovering the secret key bits from the state may not be possible. Here the secret key size is $\kappa = 80$. They consider that only one fourth of the secret key bits (i.e., 20 out of 80) may be recovered, but the rest of the bits (i.e., $3\kappa/4 = 60$) could be found only by exhaustive key search, i.e., with an effort of 2^{60}.

In Section 2, we show that this is not correct. In fact, once the state is known at some round while PRGA is running, one can find the secret key bits quite efficiently. This raises serious security concern regarding this specific cipher. The use of each key bit independently in updating the state during the PRGA seems to be the main reason for recovering the secret key bits.

Algebraic Attacks. The designers claim that algebraic attack may not be possible on Sprout as the algebraic equations become complicated rapidly. While this statement is indeed true (see Table 4), we manage to keep the degree bounded by adding new variables for the Linear Feedback Shift Register (LFSR) and the Non-Linear Feedback Shift Register (NFSR) updates (Sections 2.1, 2.2).

We show that guessing a portion of the state bits, it is possible to recover the secret key too. That presents a cryptanalysis of the cipher as the state size is equal to the key size in this design. One may always consider design modifications that may be tried to resist such attacks, but the ideas will always attract additional hardware and thus the main motivation towards such designs with shorter states will be lost.

Fault Attacks. The authors of Sprout mention that though the ciphers in the Grain family are vulnerable to fault attacks [4,18], those ideas will not affect Sprout as one cannot recover the secret key even if the state is known by fault attack.

As we have noted earlier, this is not correct indeed. In Section 3, we present how it is possible to recover the state by mounting Differential Fault Attack (DFA) against Sprout. We need around 120 faults to mount such an attack and we respect all the usual assumptions related to the restrictions on injecting the faults. However, we note that Sprout is better resistant than the Grain family against the DFA.

Before proceeding further, let us briefly describe the stream cipher Sprout and provide some introduction to fault attacks.

1.1 Description of Sprout

Sprout is basically an adaptation of Grain 128a [1]. Table 1 gives an overview of the stream cipher. One may note that the designers of Sprout reduce the size of the key (80 bits), IV (70 bits) and state (80 bits) from Grain 128a; while they increase the number of initialization rounds (320).

Cipher	Key size	IV size	State size	Initialization rounds
Sprout	80	70	80 (40 LFSR + 40 NFSR)	320
Grain 128a	128	96	256 (128 LFSR + 128 NFSR)	256

Table 1. Overview of Sprout and Grain 128a.

Like the members of the Grain family [1,2,12,13], the state of Sprout is composed of one LFSR and one NFSR, which we denote by L_t and N_t respectively for a given round t. As mentioned in Table 1, both of them are of 40 bits. For a given round t, we denote LFSR bits as $l_t, l_{t+1}, \ldots, l_{t+39}$ and NFSR bits as $n_t, n_{t+1}, \ldots, n_{t+39}$ respectively. We also denote the secret key by k and its bits by $\mathsf{k}_0, \mathsf{k}_1, \ldots, \mathsf{k}_{79}$, respectively.

Sprout keeps the secret key intact in a separate register other than the 80-bit state. In each round after the initial 80 rounds, one particular key bit will be involved in the state through the bit n_{t+39} (depending on the output of a 6 variable linear function of the state) or will not be involved at all. Initially, to make sure all the key bits are involved in the state, in the first 80 rounds of the KSA routine, each key bit is directly involved in the system (through n_{t+39}). This process is determined by the function 'round key', which is defined as [3, Equation 5]:

$$k_t^* = \begin{cases} \mathsf{k}_t, \text{ if } 0 \le t \le 79 \\ \mathsf{k}_{(t \bmod 80)} \wedge (l_{t+4} \oplus l_{t+21} \oplus l_{t+37} \oplus n_{t+9} \oplus n_{t+20} \oplus n_{t+29}), \text{ otherwise.} \end{cases} \quad (1)$$

Thus, after the first 80 rounds of initialization, one particular key bit is involved in the state if the linear term $(l_{t+4} \oplus l_{t+21} \oplus l_{t+37} \oplus n_{t+9} \oplus n_{t+20} \oplus n_{t+29})$ results in 1. Ideally, this term should be perfectly random over $\{0,1\}$, and hence one particular key bit is involved in the state with a probability of $\frac{1}{2}$.

This cipher uses one counter which is composed of 9 bits; of which the lower 7 bits are a modulo 80 counter, whose bits are denoted by $(c_t^6, c_t^5, c_t^4, c_t^3, c_t^2, c_t^1, c_t^0)$ respectively for a given round t. The 5th LSB (c_t^4) of this counter is only employed in the design. It may be noted that, c_t^4 has a cycle of period 80 - in each cycle it takes the values : $\underbrace{0,0,\ldots,0}_{\text{16 times}},\underbrace{1,1,\ldots,1}_{\text{16 times}},\underbrace{0,0,\ldots,0}_{\text{16 times}},\underbrace{1,1,\ldots,1}_{\text{16 times}},\underbrace{0,0,\ldots,0}_{\text{16 times}}$.

The NFSR is initially loaded with the first 40 bits of IV. During the KSA, the register is shifted one bit 320 times; and the bit n_{t+39} is updated as $n_{t+39} = z_t \oplus g(N_t) \oplus k_t^* \oplus l_t \oplus c_t^4$. Here the non-linear function $g(N_t)$, having the same form of Grain 128a, is defined as:

$$\begin{aligned} g(N_t) = {} & n_t \oplus n_{t+13} \oplus n_{t+19} \oplus n_{t+35} \oplus n_{t+39} \\ & \oplus n_{t+2}n_{t+25} \oplus n_{t+3}n_{t+5} \oplus n_{t+7}n_{t+8} \oplus n_{t+14}n_{t+21} \oplus n_{t+16}n_{t+18} \\ & \oplus n_{t+22}n_{t+24} \oplus n_{t+26}n_{t+32} \oplus n_{t+33}n_{t+36}n_{t+37}n_{t+38} \\ & \oplus n_{t+10}n_{t+11}n_{t+12} \oplus n_{t+27}n_{t+30}n_{t+31} \end{aligned}$$

and z_t denotes the key-stream produced, which is defined as:

$$\begin{aligned} z_t = {} & (n_{t+4}l_{t+6} \oplus l_{t+8}l_{t+10} \oplus l_{t+32}l_{t+17} \oplus l_{t+19}l_{t+23} \oplus n_{t+4}l_{t+32}n_{t+38}) \\ & \oplus (l_{t+30}) \oplus (n_{t+1} \oplus n_{t+6} \oplus n_{t+15} \oplus n_{t+17} \oplus n_{t+23} \oplus n_{t+28} \oplus n_{t+34}). \end{aligned} \tag{2}$$

The LFSR L_t, is initially loaded with the last 30 IV bits appended with a specific 10 bit pattern: $(1,1,1,1,1,1,1,1,1,0)$. In each round, it is shifted one bit, and the bit l_{t+39} is updated from both its contents as well as the key-stream bit during KSA: $l_{t+39} = l_t \oplus l_{t+6} \oplus l_{t+15} \oplus l_{t+20} \oplus l_{t+25} \oplus l_{t+35} \oplus z_t$.

Here the key-stream bit is generated prior to the state update. Like the members of the Grain family, the key-stream produced during the KSA is not outputted, rather it is internally fed to the state (through n_{t+39}). When the KSA routine is over, the PRGA routine is carried out and now the key-stream produced is not fed back. Thus, in PRGA:

– the NFSR update function is changed to: $n_{t+39} = g(N_t) \oplus k_t^* \oplus l_t \oplus c_t^4$,
– the LFSR update function is changed to: $l_{t+39} = l_t \oplus l_{t+6} \oplus l_{t+15} \oplus l_{t+20} \oplus l_{t+25} \oplus l_{t+35}$,
– the key-stream produced z_t is outputted;

and everything remains the same as in KSA. It is to be mentioned that the round t is not reset to zero at the beginning of PRGA.

1.2 Brief Background of Fault Attack

Fault attacks consider that a fault can be injected to toggle one or more locations of the cipher state and then it is possible to note the differences in the key-stream bits to recover certain information about the state. This model of attack could be successfully employed against a number of cryptographic primitives [7]. Initial works in this direction appear in [8,10] and later such attacks on stream ciphers have been explored in [14]. Here, a typical attack scenario generally consists of an adversary who can inject a random fault (using laser shots/clock glitches [19,20]) on a cryptographic device and as a result of that one or more bits of its internal state may get altered. Once a fault is injected, the faulty output from this altered device is utilized to deduce information about its internal state. In this model, the attacker needs a few privileges like the ability to re-key the device, control the timing of the fault etc. One may note that this is in fact a differential attack [9]. In Differential Fault Attack, the attacker is allowed to inject faults in the internal state during the PRGA (rather than putting differences during KLA as done in differential attacks against stream ciphers). Then by analyzing the difference between the faulty and the fault-free key-streams, the attacker should be able to obtain some information about the internal state.

One may be aware that all the ciphers in the eStream [11] hardware portfolio, namely Grain v1, Mickey 2.0 and Trivium, have been cryptanalyzed against Differential Fault Attacks [4,5,6,15,16,17,18]. In all these cases, it was enough to recover the cipher state by DFA as the KSA and PRGA of such ciphers are reversible, providing the secret key once the state is known. However, this is not the case for Sprout [3]. In Sprout, the recovery of state does not immediately imply that the secret key bits will be revealed too. In fact, the designers comment that this may not be possible efficiently. However, in Section 2 we refute that claim and show that one can obtain the secret key bits efficiently once the state is known.

Another difference in the fault attack scenario here is we have the unknown secret key bits constantly involved in the state update. This is unlike that of Grain v1, Mickey 2.0 and Trivium. Thus obtaining clean signatures of the fault locations, to identify the location of a fault injected in a random position, is quite a challenging task here. We use the probability of matching between each corresponding pair of fault-free and faulty key-stream bits to explain the signature. Further we consider correlations between the differential stream and the signatures to obtain the good matches and thereby identifying the fault location (Section 3). After finding the location of the faults, we can use corresponding differential key streams to obtain a system of nonlinear equations and those can be solved using an efficient SAT Solver tool (Section 3.1).

Organization of the Paper. In the rest of the paper, we discuss about the applicability of the three type of attacks we have mentioned. Section 2 deals with the Guess and Determine approach, where we assume that (somehow) we know the state (LFSR + NFSR) of the cipher at some round during the PRGA, and use the information to recover the secret key efficiently. Next, in Sections 2.1 and 2.2, we present how Algebraic Attacks can be mounted on the cipher successfully given the full/partial knowledge of the state. The results we obtain are quite surprising and it raises serious concerns regarding the security of the cipher. Later, in Section 3 we discuss about the fault attack model on Sprout. We first discuss about fault signatures and then, in Section 3.1 we deal with the problem of state recovery by observing the actual key-streams and the faulty key-streams. Thus, the fault attack finds out the state information from the key-streams; and the previous two attacks find out the secret key from the known state information. Finally, Section 4 concludes the paper.

2 Recovering the Secret Key Bits from the State

Let us begin with what has been explained in [3, Section 5] regarding the security against the Guess and Determine Attacks.

> "The newly inserted key bit is not used for the output until it propagates to the position n_{t+38}. There it will become part of the monomial $n_{t+4}n_{t+38}l_{t+32}$ of the output function. Even if the attacker knows the other values of the output function, she can only recover this key bit when n_{t+4} and l_{t+32} are both equal to 1. Hence in average only one bit out of four can be recovered in this straightforward way. Observe in addition that before this particular key bit is involved in the output function for the first time, it has been used as linear terms in the NFSR update function when it was at the position n_{39}. Thus, the key bit influences the state of the NFSR before it could be recovered (which is the case with probability 1/4 only). Guessing the key bits that cannot be recovered would hence induce an additional effort of $O(2^{3\kappa/4}) = O(2^{60})$."

While the above observation is correct, the corollary in the last sentence is indeed an upper bound and that can be reduced drastically by looking at the involvement of a specific secret key bit in several rounds of key-stream bits.

Here we assume that, for a particular round t, we somehow intercept 80 bits of the state during PRGA. Now let us take a closer look at how the state update is done. As described in Equation 1, one particular secret key bit (namely, $\mathsf{k}_{(t \bmod 80)}$) may be involved in the state through n_{t+39}, which is determined by the following round key function:

$$k_t^* = \begin{cases} \mathsf{k}_t, & \text{if } 0 \le t \le 79 \\ \mathsf{k}_{(t \bmod 80)} \wedge (l_{t+4} \oplus l_{t+21} \oplus l_{t+37} \oplus n_{t+9} \oplus n_{t+20} \oplus n_{t+29}), & \text{otherwise.} \end{cases}$$

We consider the case $t \ge 80$ as this case is prevalent during PRGA. As we know the round t, we immediately know which particular secret key bit may be involved to the state in that round.

Definition 1 (Involvement Term). *Given a particular round t, the involvement term, denoted by μ_t, is defined as:*

$$\mu_t = \begin{cases} 1, & \textit{if } 0 \le t \le 79 \\ l_{t+4} \oplus l_{t+21} \oplus l_{t+37} \oplus n_{t+9} \oplus n_{t+20} \oplus n_{t+29}, & \textit{otherwise.} \end{cases}$$

Clearly, $\mathsf{k}_{(t \bmod 80)}$ will be involved to the state in the t-th round if $\mu_t = 1$. At this point we show that, given a state, we can practically get this information for 20 consecutive key bits (on whether or not they will be involved in the corresponding state) in Theorem 1.

Theorem 1. *Consider that at a round t during PRGA, the state (i.e.,* $[n_t, \ldots, n_{t+39}]$ *and* $[l_t, \ldots, l_{t+39}]$*) is available. Then it is possible to obtain* μ_u *for* $u \in [t-8,\ t+11]$.

Proof. Look at the locations used in the NFSR for the involvement of the secret key bits. These are n_{t+9}, n_{t+20} and n_{t+29}. Now looking at the sequence of these three bits for $t = -8, \ldots, 0, \ldots, 11$; one may note that all these triplets are available from the state information $[n_{t+39}, \ldots, n_t]$. The corresponding LFSR bits will always be available as we can move forward and backward with the LFSR. Thus, $\mu_u = l_{u+4} \oplus l_{u+21} \oplus l_{u+37} \oplus n_{u+9} \oplus n_{u+20} \oplus n_{u+29}$ is known for $u \in [t-8, t+11]$. □

One may notice that, in order to generate key-stream z_t for the round t, we need to know the first bit of NFSR (n_t). Since we do not precisely know the corresponding n_t's for $t \leq 0$ (it is lost during NFSR update), we have to exhaustively search for it. This appears to be less practical. Hence, we should consider μ_t's for $t > 0$.

Definition 2 (Involvement Vector). *The vector,* $M_t = \{\mu_{t+1}, \ldots, \mu_{t+11}\}$ *for a given round t is called the involvement vector.*

Based on the involvement vector for a round t, we can easily infer about whether the key bits $\mathsf{k}_{(t+1 \bmod 80)}, \mathsf{k}_{(t+2 \bmod 80)}, \ldots, \mathsf{k}_{(t+11 \bmod 80)}$ will be involved in the rounds $t+1, t+2, \ldots, t+11$, if we are given the state at round t.

M_t does not contain any term involving the secret key bits; and hence, to compute M_t it is sufficient to know the state at round t. Now, if M_t is of Hamming weight w, we can generate 2^w trial keys and use these trial keys one by one to produce key-streams. Noticing whether or not these trial key-streams match with the original key-stream, we can reduce the search complexity for the secret key space greatly. Moreover, if for all the trial keys that produce matching key-streams, some bit(s) in the corresponding position(s) is fixed to a value (either 0 or 1) and the corresponding M_t bit is 1, then we know for sure that the corresponding secret key bit must be equal to that fixed value. Such information, in turn, can be further utilized to infer about more secret key bits. Hence, by a dynamic programming approach, we can find out all the 80 secret key bits. As in each step, trial keys are set separately, the overall computational complexity is not multiplicative, rather additive. Hence, the complexity for this approach remains bounded, and should be much less than the exhaustive search of complexity 2^{80}. This is the central theme of our Guess and Determine Attack.

Let us now describe a few issues that are to be taken care of while generating the matching key-streams:

1. Since, we now have trial keys each of length 11, we can generate at most 13 key-stream bits. This is because, the key-stream generation precedes the NFSR update (we do not have to bother about the LFSR, since once we know it, we can always run it in forward or backward direction as many rounds we want) in each round t. Now, the secret key bit is involved in n_{t+39} (if at all); and the earliest possibility that the key bit may influence the key-stream is through n_{t+38} (from Equation 2), i.e., 2 rounds later. Also notice that, in the meantime, two more key bits may be involved in the state, but they will not affect the key-stream produced before round z_{t+13}.
2. The first two key-stream bits will be identical for all the trial keys (and it will match with the first two of the original key-stream). The reason is already mentioned in the last point: A key bit $\mathsf{k}_{(t \bmod 80)}$ will affect (if at all) the key-stream bit z_{t+2} at the earliest.

Experimental results show that, in the initial involvement vector, on an average 5.5 secret key bits are involved in the state (which is justifiable, since each of the 11 key bits has 0.5 probability to be involved). Hence, we run an exhaustive search over the 5.5 secret key bits that will be involved in the following key-stream bits ($2^{5.5}$ computations). We notice that around 3.5 bits of secret key bits can be recovered from this search.

Example. Assume, without loss of generality, that we manage to get a copy of 80-bit state just after the KSA routine is performed, i.e., the state is known for $t = 320$, $c_t^4 = 0$, so that we know that the secret key bits of interest are $\mathsf{k}_0, \mathsf{k}_1, \ldots$, in that order. Now the PRGA routine is carried out generating the key-stream bits, and also changing the state (but we do not have access to the state now). Also note that, although public, we do not need to use the IV in the course of this algorithm.

Now, for our example, assume that the NFSR and the LFSR cells are:

$$N_t = 0101011000011110110011111011110111100000$$
$$L_t = 0101000100001010001100000101001010010001$$

One may note that the first involvement vector, $M_t = (0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1)$. That means, the key bits $\mathsf{k}_1, \mathsf{k}_3, \mathsf{k}_7, \mathsf{k}_9, \mathsf{k}_{10}$ will be involved in the state. Since the Hamming weight of M_t is 5, we run 2^5 separate trial keys (for the key bits which are not involved, we set 0) to see which of them produce matching key-streams with the original key-stream. Here, in this case, original key-stream (first 13 bits) is $1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1$. As it turns out, there are 4 trial keys of such type:

$$0, \mathbf{1}, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 1, 0, 0$$
$$0, \mathbf{1}, 0, \mathbf{1}, 0, 0, 0, 0, 0, 1, 0, 0, 0$$
$$0, \mathbf{1}, 0, \mathbf{1}, 0, 0, 0, 1, 0, 0, 0, 0, 0$$
$$0, \mathbf{1}, 0, \mathbf{1}, 0, 0, 0, 1, 0, 1, 1, 0, 0$$

Notice that, the 1st bit (2nd MSB) in each trial key is identical (**1**), so as the situation for the 3rd bit; and in both the cases, the corresponding bit of M_t is 1. Thus, now we know that k_1 and k_3 are both 1.

We may continue with this idea for each block of 11 bits consecutively. In that case, we may expect to have much less effort than the exhaustive key search. While this idea is worth exploring and we are working on it, we have noted that directly attempting to solve the system of equations efficiently recovers the secret key too. Next, we explain this.

2.1 Obtaining the Secret Key Bits from the Complete State Information and the Key-stream

This approach is based on algebraic techniques. We treat the secret key bits and unknown state bits as Boolean variables, and then generate several equations. Then, we solve them using a SAT solver tool.

Forming the Equations. Our idea is to introduce new variables to the system of equations such that the degree is bounded, but the number of variables as well as equations grow. Let $z_t, \ldots, z_{t+\ell-1}$ be the key-stream bits from round t to $t+\ell-1$. Suppose at round t, we know both the LFSR $L_t = [l_t, l_{t+1}, \ldots, l_{t+39}]$ and NFSR $N_t = [n_t, n_{t+1}, \ldots, n_{t+39}]$. Our target is to find the secret key from the knowledge of $z_t, \ldots, z_{t+\ell-1}$; $l_t, l_{t+1}, \ldots, l_{t+39}$ and $n_t, n_{t+1}, \ldots, n_{t+39}$. Since L_t and N_t have no unknown bit (all bits are known to be either 0 or 1), key-streams will be some function of the secret key bits only.

However, it may be noted that; it is practically difficult, if not infeasible, to compute the Algebraic Normal Form (ANF) of key-streams after certain rounds as the number of monomials grows exponentially. To overcome this problem, at each PRGA round $t > 0$, we introduce two new variables: $\mathcal{L}_t$ and $\mathcal{N}_t$; equate them with the update functions of L_t and N_t, respectively, and then update the registers. This means, at each round, we are increasing the number of variables as well as the equations by 3 (another equation comes due to the key-stream).

We initially start with 80 variables: $\mathsf{k}_0, \mathsf{k}_1, \ldots, \mathsf{k}_{79}$. Then corresponding to each key-stream bit z_{t+i}, we introduce two new variables $\mathcal{L}_{t+i}, \mathcal{N}_{t+i}$ and obtain three more equations, for all $i = 0, \ldots, \ell-1$. Thus, in total we have 3ℓ equations over $80 + 2\ell$ variables. Note that, this approach allows us to formulate the expression for z_{t+i} via a series of equations, for all $i = 0, \ldots, \ell-1$. If at each round $t > 0$, the variables $\mathcal{L}_t, \mathcal{N}_t$ were replaced by their equivalent algebraic expressions in $\mathsf{k}_0, \mathsf{k}_1, \ldots, \mathsf{k}_{79}$, the expressions will be too large to handle after certain rounds. This is one clear advantage of our approach.

Required Number of Key-stream Bits. One may be interested to know how many key-stream bits, at the least, are required to recover all the secret key bits $\mathsf{k}_0, \mathsf{k}_1, \ldots, \mathsf{k}_{79}$. An estimation of this query can be given as follows. Consider 80 consecutive key-stream bits as a block. Note that, one particular key-stream bit, say z_{t+2}, will contain the key bit $\mathsf{k}_{t \bmod 80}$ with probability $\frac{1}{2}$, as we consider

μ_t to be uniformly random over $\{0, 1\}$ (which is a well accepted assumption for a good cipher). This indicates that, in a block of key-stream bits, around 40 key bits will be involved if we further assume the independence of μ_t's.

Now consider 80 key-stream bits as a block. In such a block, one particular key bit will not be involved with probability $\frac{1}{2}$. Next, we generalize this observation for x such consecutive blocks. That is, in $80x$ consecutive key-stream bits, one particular key bit will not be involved with probability $\frac{1}{2^x}$ (assuming the independence of the key-stream bits). Therefore, the expected number of key bits that will not be involved in the $80x$ consecutive key-stream bits is $\frac{80}{2^x}$. To ensure each key bit is involved in the $80x$ consecutive key-stream bits at least once, we require that $\frac{80}{2^x} < 1$. This gives $x \geq 7$. In other words, we need at least $7 \times 80 = 560$ key-stream bits to expect that each key bit is involved in (at least one) key-stream bit. Practically we observe through our experiments, for unique and quick solvability this required number of key-stream bits is nearly 900. This means, we manage to find out the secret key (80 bits) on the basis of nearly 900 following key-stream bits together with the LFSR and NFSR state information at a specific round.

We formulate the whole problem in a SAT solver, Cryptominisat-2.9.5, under SAGE 5.13 [21] on a laptop running with Linux Mint 17.1. The hardware configuration is: Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz D CPU 1.83 GHz and 4 GB RAM. It is worth mentioning that, the system of equations can be solved within 1 second in our implementation. In Table 2, we present our experimental results which we obtain after averaging 100 random experiments. While column 1 of Table 2 gives the number of key-stream bits we utilise per experiment, column 2 gives the average number of solutions as returned by the SAT solver, column 3 gives the average number of solved key-bits, and the last column gives the average time consumed by our computer. It is to be noted that, column 3 is basically the result of logarithm (base 2) of column 2, subtracted from 80.

# Key-stream bits	# Solutions	Avg. # solved key bits	Time (second)
350	23.2	75.46	0.24
450	3.1	78.37	0.24
550	1.9	79.07	0.28
650	1.2	79.74	0.32
750	1.1	79.86	0.37
850	1.1	79.86	0.41
900	1.0	80	0.45

Table 2. Results observed when complete state information is known.

2.2 Obtaining the Complete Secret Key Bits from Partial State Information and Key-stream

Since there are some really impressive results towards finding out the secret key bits given the complete state information (Section 2.1), one natural question is therefore: Whether it is possible to find the secret key from the partial (< 80 bits) information of the state.

# Key-stream bits	LFSR bits known	NFSR bits known	# Solutions	Avg. # solved key bits	Time (seconds)
300	First 30	First 30	292.8	71.81	21.76
350	First 30	First 30	237.8	72.11	55.33
400	First 30	First 30	5.4	77.57	51.39
300	Every 1/3rd	All	602.6	70.77	50.54
350	Every 1/3rd	All	34.4	74.90	49.33
400	Every 1/3rd	All	6.6	77.28	76.64

Table 3. Results observed when partial state information is known.

Thus, for the experiments, we now assume state is known except for a few bits. Since the state and the key in Sprout are of same length, this kind of analysis gives better attack than the exhaustive key search. We present our experimental results in Table 3. After lot of simulations, we observe that if NFSR is known fully and 1/3rd bits LFSR are known (i.e., $l_t, l_{t+3}, l_{t+6} \ldots, l_{t+39}$ are known), then one can find the remaining unknown state bits and key very efficiently. Thus, it is possible to cryptanalyze Sprout with search complexity $< 2^{80}$.

3 Differential Fault Attack: Fault Signatures, Identifying Fault Locations and Obtaining the State

Consider that we put a difference by injecting a fault at the f-th bit of the state. The state is of 80 bits, consisting of a 40-bit NFSR denoted by $[n_t, \ldots, n_{t+39}]$ and a 40-bit LFSR state $[l_t, \ldots, l_{t+39}]$. Introducing a fault at f-th bit means, if $f < 40$, then the fault is injected at the LFSR location f, and for $f \geq 40$, we consider that the fault is injected at the $(f-40)$-th bit of the NFSR.

Given a fault f, we consider z_i and $z_i^{(f)}$, the key-streams for the fault-free and the faulty cases, for $i = 0$ to $\lambda - 1$. We denote $\zeta_i^{(f)} = z_i \oplus z_i^{(f)}$, and $q_i^{(f)} = \frac{1}{2} - \Pr(\zeta_i^{(f)} = 1)$.

Definition 3. *The vector* $Q^{(f)} = (q_0^{(f)}, q_1^{(f)}, \ldots, q_{\lambda-1}^{(f)})$ *is called the signature of the fault at location* f. *The sharpness of the signature* $Q^{(f)}$ *is defined as* $\sigma(Q^{(f)}) = \frac{1}{\lambda}\sum_{i=0}^{\lambda-1} |q_i^{(f)}|$.

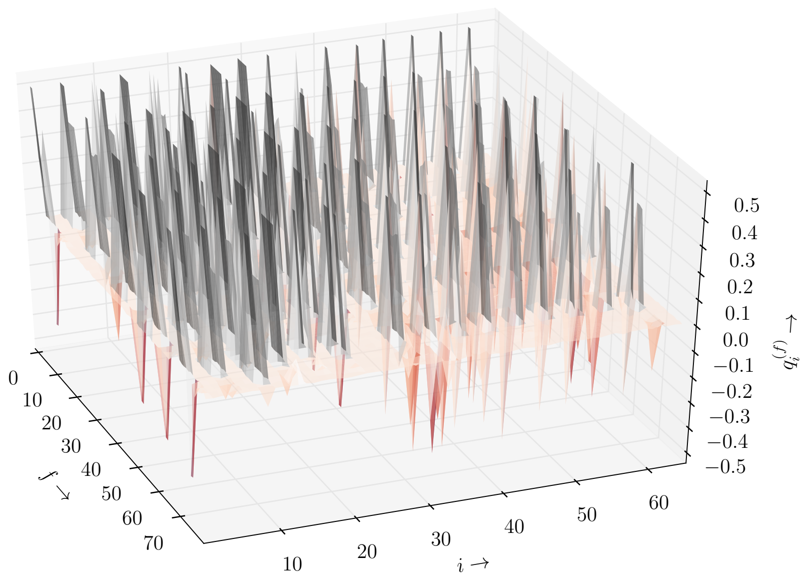

Fig. 1. Plot of $Q^{(f)}$ for all f in $[0, 79]$.

For our experiments, we consider $\lambda = 64$. Further, we make 2^{15} runs with random key-IV pairs to prepare the signatures $Q^{(0)}, Q^{(1)}, \ldots, Q^{(79)}$. In the actual attack model, this is done in the off-line phase by the attacker by knowing the description of the cipher. The signatures are presented in Figure 1. These signatures are stored in a file for comparison during the on-line phase. As we have defined the sharpness, we may note that the faults at certain location may be useful than some other location. For example, one may note the cases for $f = 30$ and $f = 31$ as in Figure 2. It is very clear that identifying the location if the fault is indeed injected at 30 (blue) has much better chance than that of 31 (red). Now consider that a fault is injected at a random (but unknown) location $g, (0 \leq g < 80)$. Corresponding to that, we will obtain the key-streams z_i and $z_i^{(g)}$, the key-streams for the the fault-free and the faulty cases. We consider $\eta_i^{(g)} = z_i \oplus z_i^{(g)}$. Let us consider $\nu_i^{(g)} = \frac{1}{2} - \eta_i^{(g)}$ and $\Gamma^{(g)} = (\nu_0^{(g)}, \nu_1^{(g)}, \ldots, \nu_{\lambda-1}^{(g)})$.

Definition 4. *The vector $\Gamma^{(g)} = (\nu_0^{(g)}, \nu_1^{(g)}, \ldots, \nu_{\lambda-1}^{(g)})$ is called the trail of the fault at the unknown location g.*

Note that, this happens while we actually inject a fault and manage to obtain the data itself. Thus, here we have no question of probability attachment. Now $\Gamma^{(g)}$ is compared with each of the $Q^{(f)}$'s, for $f = 0, \ldots, 79$.

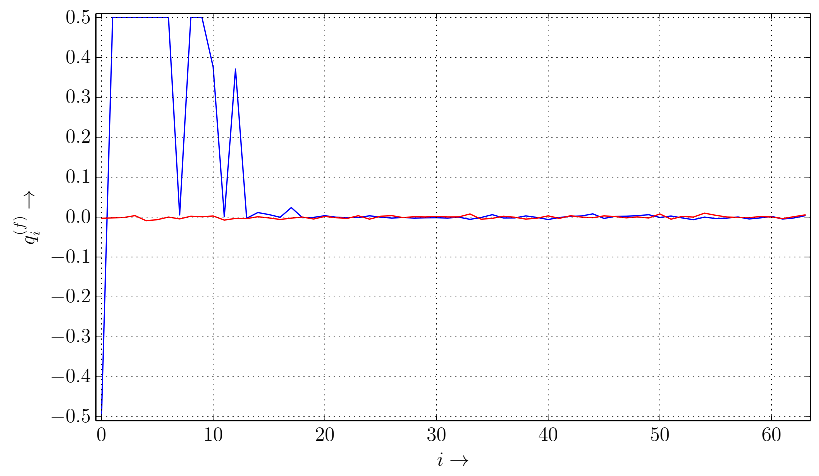

Fig. 2. Plot of $Q^{(30)}$ (blue) and $Q^{(31)}$ (red).

Definition 5. *We tell that a signature* $Q^{(f)} = (q_0^{(f)}, q_1^{(f)}, \ldots, q_{\lambda-1}^{(f)})$ *and a trail* $\Gamma^{(g)} = (\nu_0^{(g)}, \nu_1^{(g)}, \ldots, \nu_{\lambda-1}^{(g)})$ *do not match, if there exists at least one* $i, (0 \leq i < \lambda)$ *such that* $(q_i^{(f)} = \frac{1}{2}$ *and* $\nu_i^{(g)} = -\frac{1}{2})$ *or* $(q_i^{(f)} = -\frac{1}{2}$ *and* $\nu_i^{(g)} = \frac{1}{2})$.

However, it is quite natural that this may not always happen, and thus we need to extend this definition. For this purpose, we incorporate the correlation coefficient between two sets of data.

Definition 6. *We use correlation coefficient* $\mu(Q^{(f)}, \Gamma^{(g)})$ *between the signature* $Q^{(f)} = (q_0^{(f)}, q_1^{(f)}, \ldots, q_{\lambda-1}^{(f)})$ *and a trail* $\Gamma^{(g)} = (\nu_0^{(g)}, \nu_1^{(g)}, \ldots, \nu_{\lambda-1}^{(g)})$ *for evaluating how good they match. Naturally,* $-1 \leq \mu(Q^{(f)}, \Gamma^{(g)}) \leq 1$. *In case they do not match as per the Definition 5, then we assign* $\mu(Q^{(f)}, \Gamma^{(g)}) = -1$.

Then we make the following experiment to consider how one can locate the faults. For each fault g (consider now that g is known), we calculate the trail $\Gamma^{(g)} = (\nu_0^{(g)}, \nu_1^{(g)}, \ldots, \nu_{\lambda-1}^{(g)})$. We now calculate $\mu(Q^{(f)}, \Gamma^{(g)})$ for each of the faults $f, (0 \leq f < 80)$. We note

1. $\max_{f=0}^{79} \mu(Q^{(f)}, \Gamma^{(g)})$,
2. $\mu(Q^{(g)}, \Gamma^{(g)})$, and
3. $\alpha(Q^{(g)})$, the number of f's for which $\mu(Q^{(f)}, \Gamma^{(g)}) > \mu(Q^{(g)}, \Gamma^{(g)})$.

One may now note in Figure 3, when $\mu(Q^{(g)}, \Gamma^{(g)})$ (blue) is close to $\max_{f=0}^{79} \mu(Q^{(f)}, \Gamma^{(g)})$ (red), $\alpha(Q^{(g)})$ is small and it is easier to locate these faults. However, if $\mu(Q^{(g)}, \Gamma^{(g)})$ is much smaller than $\max_{f=0}^{79} \mu(Q^{(f)}, \Gamma^{(g)})$ (red), i.e., $\alpha(Q^{(g)})$ is large, that means identifying this fault location will be harder.

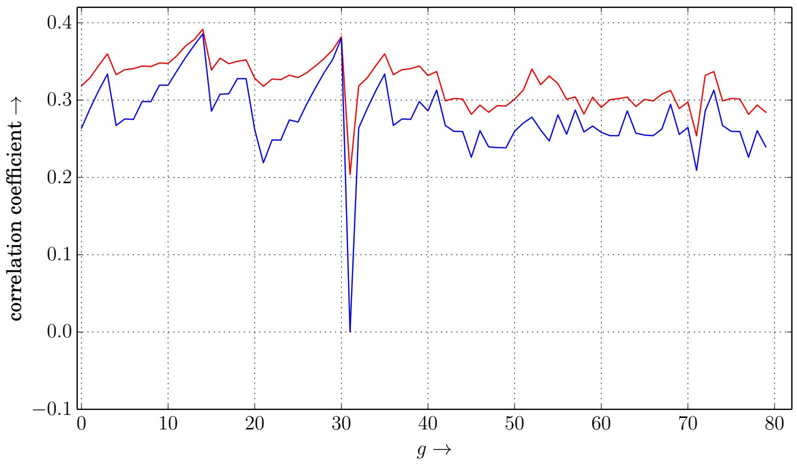

Fig. 3. Plot of $\max_{f=0}^{79} \mu(Q^{(f)}, \Gamma^{(g)})$ (red) and $\mu(Q^{(g)}, \Gamma^{(g)})$ (blue).

Given $\alpha(Q^{(g)})$, for each g, we can actually estimate how many attempts we should require to obtain the actual fault location. Further, we should consider a few parameters to explain our attack. From the experimental data, one may note that obtaining the random fault locations from signatures is more challenging in Sprout than those of the Grain family. This is due to the non-linearity and secret key bit involvement in case of Sprout. As we will describe in Section 3.1, obtaining the exact state is possible from the differential key-stream corresponding to 20 correct fault locations (details are in Table 5). Thus, we need to pinpoint 20 fault locations from a larger number of random faults.

The exact algorithm for mounting the fault attack is as follows, where we consider that all the faults are injected in the same round.

– Inject a fault at some random fault location.
– Obtain the differential trail $\Gamma^{(g)} = (\nu_0^{(g)}, \nu_1^{(g)}, \ldots, \nu_{\lambda-1}^{(g)})$. Note that we need to estimate g.
– For each f in $[0, 79]$, calculate $\mu(Q^{(f)}, \Gamma^{(g)})$.
– Prepare a list of possible fault locations in Loc_i for this i-th experiment when $\mu(Q^{(f)}, \Gamma^{(g)}) \geq \tau$, for some threshold τ.

We need to re-key the cipher again and again to prepare a list of possible fault locations Loc_i, for $i = 1$ to 20 towards solving the systems of equations as described later. Naturally, we will try to obtain faults in the locations where it is easier to identify them. That means, we would like to locate the faults when it is injected at some location g such that $\alpha(Q^{(g)})$ is small.

After several experimentation, we consider the set of following 26 fault locations, $S = \{10, 11, 12, 13,\ 14, 18, 19, 28, 29, 30,\ 41,\ 42,\ 46,\ 55,\ 56,\ 57,\ 58,\ 60,\ 63,$

$64,\ 68,\ 69, 71, 73, 74, 78\}$, (once again note that the values < 40 belong to the LFSR and the values ≥ 40 belong to NFSR) for which $\prod_{g \in S}(1 + \alpha(Q^{(g)})) \approx 2^{39.82} < 2^{40}$. This means that in such a case we need to try out little less than 2^{40} options to find the correct fault locations out of 26 places. As described in Section 3.1, it is enough to have the correct knowledge of 20 fault locations.

Now the question is how one can manage the faults in those locations. Naturally, a random fault can be injected to a specific fault location with probability $\frac{1}{80}$. That is, a location f will not be touched in r attempts is $(1 - \frac{1}{80})^r$. Thus, the expected number of locations in S that will not be touched by a fault in r attempts is $26 \cdot (1 - \frac{1}{80})^r$ and we like this quantity to be less than 6 so that we can have around 20 correct locations. One may note that this can be achieved with $r = 120$. Further $\max_{T \subset S, |T|=20} \prod_{g \in T}(1 + \alpha(Q^{(g)})) < 2^{35.2}$. This means that we will be able to obtain 20 correct fault locations with good expectation given an effort of less than $2^{35.2}$. Once we obtain the correct complete state, we can use that for obtaining the secret key bits as in the previous section. Naturally, when we obtain wrong states and wrong secret key bits for wrong choices of fault locations, then the differential key-stream generated from that solution will not match with the exact output. In this manner, we can identify the correct fault locations and thereby obtain the correct state and secret key bits.

3.1 Obtaining the State from the Differential Key-streams

As pointed out earlier (in Section 2.1), the ANF of the key-stream expression will be huge within a very few rounds. Table 4 quickly encompasses the rapid increase of the ANF expression of the key-stream produced by Sprout. Here, at the beginning of PRGA, we start with a total of $80+40+40 = 160$ variables: $\mathsf{k}_0, \mathsf{k}_1, \ldots, \mathsf{k}_{79}$ and $l_t+0, l_{t+1}, \ldots, l_{t+39}, n_t, n_{t+1}, \ldots, n_{t+39}$. With our current computational capability, it is difficult to go beyond round 9.

Round	1	2	3	4	5	6	7	8	9
Degree	3	3	6	6	8	10	12	14	16
# Monomials	13	13	34	55	95	512	9026	46385	2674135

Table 4. Growth of key-stream expression of Sprout.

With that set-up, we next generate the key-stream bits z_t; then we introduce two new variables $\mathcal{L}_t, \mathcal{N}_t$ and obtain three more equations (it is already explained in Section 2.1). Thus, finally we have 3ℓ equations over $160 + 2\ell$ variables generated from fault-free key-streams.

We use a similar technique to extract equations from faulty key-streams. Let us assume that a fault is injected in the LFSR location ϕ at PRGA round t. The same method will work if the fault is injected in the NFSR. Since we re-key the cipher with the same (key, IV) pair before injecting a fault; after fault injection we get the state $l_t, l_{t+1}, \ldots, l_{t+\phi-1}, 1 \oplus l_{t+\phi}, l_{t+\phi+1} \ldots, l_{t+39}$ and

$n_t, n_{t+1}, \ldots, n_{t+39}$ at the t-th round of PRGA. Then corresponding to each faulty key-stream bit z_t^{ϕ}, we introduce two new variables $\mathcal{L}_t^{(\phi)}, \mathcal{N}_t^{(\phi)}$ and obtain three more equations. Thus we have additional 2ℓ variables and 3ℓ equations for each faulty key-streams.

Thus when we introduce ν faults, the total number of variables is $160+2(\nu+1)\ell$ and the total number of equations is $3(\nu+1)\ell$.

Experimental Results. After the identification of fault location and injection time, a system of equations are formulated, and the equations are then fed into a SAT solver. In our experiments, we want to minimize the number of faults so that it helps in reducing the number of re-keying of the cipher.

# Faults	Solution time (seconds)		
	Minimum	Maximum	Average
22	5.89	42.83	10.23
21	14.18	109.55	25.82
20	7.15	982.14	231.51

Table 5. Results observed while obtaining state from fault attack.

We have considered the case when the faults are introduced in both LFSR and NFSR randomly (here we consider that expected half of the faults are injected in LFSR and the other half in NFSR). The results are as in Table 5. The amount of key-stream is taken as 23. We have presented the time required for the SAT solver part. For each row, we consider a set of ten (10) experiments. From Table 5, it is clear that we can easily find the state of Sprout by injecting faults at 20 locations when the correct locations are known. This can be achieved by injecting around 120 faults at random locations as explained previously.

4 Conclusion

In this paper, we have discussed several crypatanalytic results on the newly proposed stream cipher, Sprout [3], which uses a different paradigm of stream cipher design, where the secret key bits are used during the PRGA. First, we demonstrate that given the state of the cipher, the secret key bits can be recovered efficiently. This refutes the claim of the designers related to the guess and determine attack. More importantly, from the knowledge of the complete NFSR (40 bits) and a partial information on the LFSR bits (around one third, i.e., 14 bits), we can obtain all the secret key bits by studying around 850 key-stream bits. Thus Sprout gets cryptanalyzed in 2^{54} attempts (considering constant time complexity required by the SAT solver in each attempt). Similar ideas are employed to mount a fault attack against Sprout that requires around 120 faults, whereas the designers claim that such a fault attack may not be possible. Since

Sprout uses the basic structure of the Grain family, there may also be a chance to extend such type of analysis on these ciphers, where secret key bits are used during the PRGA. Further, we expect to have some more designs on this very paradigm to appear in near future, where our results may be utilized for better understanding of such designs. In this direction, one may note the following informal point. Instead of treating just one key bit each time in the round key function, some function of multiple key bits could be used. This prevents the straightforward involvement of a particular key bit at one round, and may provide better security at the cost of little hardware.

References

1. M. Ågren, M. Hell, T. Johansson and W. Meier. A New Version of Grain-128 with Authentication. Symmetric Key Encryption Workshop 2011, DTU, Denmark.
2. M. Ågren, M. Hell, T. Johansson and W. Meier. Grain-128a: a new version of Grain-128 with optional authentication. IJWMC, 5(1): 48–59, 2011. This is the journal version of [1].
3. F. Armknecht and V. Mikhalev. On Lightweight Stream Ciphers with Shorter Internal States. To be presented in FSE 2015.
4. S. Banik, S. Maitra and S. Sarkar. A Differential Fault Attack on the Grain Family of Stream Ciphers. In CHES 2012, LNCS, Vol. 7428, pp. 122–139.
5. S. Banik and S. Maitra. A Differential Fault Attack on MICKEY 2.0. CHES 2013, LNCS, Vol. 8086, pp. 215–232, 2013.
6. S. Banik, S. Maitra and S. Sarkar. Improved differential fault attack on MICKEY 2.0. Journal of Cryptographic Engineering. http://link.springer.com/article/10.1007%2Fs13389-014-0083-9, 2014
7. A. Barenghi, L. Breveglieri, I. Koren and D. Naccache. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. Proceedings of the IEEE, Vol. 100, No. 11, November 2012, pp. 3056–3076.
8. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In CRYPTO 1997, LNCS, Vol. 1294, pp. 513–525.
9. E. Biham and O. Dunkelman. Differential Cryptanalysis in Stream Ciphers. Cryptology ePrint Archive, Report 2007/218.
10. D. Boneh, R. A. DeMillo and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In EUROCRYPT 1997, LNCS, Vol. 1233, pp. 37–51.
11. The ECRYPT Stream Cipher Project. eSTREAM Portfolio of Stream Ciphers. http://www.ecrypt.eu.org/stream/
12. M. Hell, T. Johansson and W. Meier. Grain - A Stream Cipher for Constrained Environments. ECRYPT Stream Cipher Project Report 2005/001, 2005. Available at http://www.ecrypt.eu.org/stream.
13. M. Hell, T. Johansson, A.Maximov and W. Meier. A Stream Cipher Proposal: Grain-128. In IEEE International Symposium on Information Theory (ISIT 2006).
14. J. J. Hoch and A. Shamir. Fault Analysis of Stream Ciphers. In CHES 2004, LNCS, Vol. 3156, pp. 1–20.
15. M. Hojsík and B. Rudolf. Differential Fault Analysis of Trivium. In FSE 2008, LNCS, Vol. 5086, pp. 158–172.
16. M. Hojsík and B. Rudolf. Floating Fault Analysis of Trivium. In INDOCRYPT 2008, LNCS, Vol. 5365, pp. 239–250.

17. Y. Hu, J. Gao, Q. Liu and Y. Zhang. Fault analysis of Trivium. Designs, Codes and Cryptography, 62(3): 289–311, 2012.
18. S. Sarkar, S. Banik and S. Maitra. Differential Fault Attack against Grain family with very few faults and minimal assumptions. To appear in IEEE Transactions on Computers, 99(PrePrints):1, 2014. http://www.computer.org/csdl/trans/tc/preprint/06857997-abs.html
19. S. P. Skorobogatov and R. J. Anderson. Optical Fault Induction Attacks. In CHES 2002, LNCS, Vol. 2523, pp. 2–12.
20. S. P. Skorobogatov. Optically Enhanced Position-Locked Power Analysis. In CHES 2006, LNCS, Vol. 4249, pp. 61–75.
21. W. Stein. Sage Mathematics Software. Free Software Foundation, Inc., 2009. Available at http://www.sagemath.org. (Open source project initiated by W. Stein and contributed by many).