Differential Fault Analysis of Trivium *

Michal Hojsík[1,2] and Bohuslav Rudolf[2,3]

[1] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
[2] Department of Algebra, Charles University in Prague,
Sokolovská 83, 186 75 Prague 8, Czech Republic
[3] National Security Authority, Na Popelce 2/16, 150 06 Prague 5, Czech Republic
michal.hojsik@ii.uib.no and b.rudolf@nbu.cz

Abstract. Trivium is a hardware-oriented stream cipher designed in 2005 by de Cannière and Preneel for the European project eStream, and it has successfully passed the first and the second phase of this project. Its design has a simple and elegant structure. Although Trivium has attached a lot of interest, it remains unbroken.
In this paper we present differential fault analysis of Trivium and propose two attacks on Trivium using fault injection. We suppose that an attacker can corrupt exactly one random bit of the inner state and that he can do this many times for the same inner state. This can be achieved e.g. in the CCA scenario. During experimental simulations, having inserted 43 faults at random positions, we were able to disclose the trivium inner state and afterwards the private key.
As far as we know, this is the first time differential fault analysis is applied to a stream cipher based on shift register with non-linear feedback.

Keywords : differential fault analysis, Trivium stream cipher, fault injection

1 Introduction

In 2004 eSTREAM project has started as part of the European project ECRYPT. At the beginning there was a call for stream ciphers and 34 proposals were received. Each proposal had to be (according to the call) marked as hardware or software oriented cipher. At the time of writing this paper, the project was in phase 3, and there were just some ciphers left. One of the requirements of the call for stream ciphers was the high throughput, so the winners can compete with AES. In this respect, one of the best proposals is the stream cipher Trivium, which is a hardware oriented stream cipher based on 3 nonlinear shift registers. Though the cipher has a hardware oriented design it is also very fast in software, which makes it one of the most attractive candidates of the eSTREAM project.

In this paper differential fault analysis of Trivium is described. We suppose that an attacker can corrupt a random bit of the inner state of Trivium. Consequently some bits of keystream difference (proper keystream XOR faulty

* The work was partly supported by the grant from GAUK n. 7302/2007.

keystream) depend linearly on the inner state bits, while other equations given by the keystream difference are quadratic or of higher order in the bits of the fixed inner state.

Since we suppose that an attacker can inject a fault only to a random position, we also describe a simple method for fault position determination. Afterwards knowing the corresponding faulty keystream, we can directly recover few inner state bits and obtain several linear equations in inner state bits. Just by repeating this procedure for the same inner state but for different (randomly chosen) fault positions we can recover the whole cipher inner state, and clocking it backwards we are able to determine the secret key. The drawback of this simple approach is that we need many fault injections to be done in order to have enough equations.

To decrease number of faulty keystreams needed (i.e. to decrease the number of fault injections needed), we also use quadratic equations given by a keystream difference. But as we will see later on, we do not use all quadratic equations, but just those which contains only quadratic monomials of a special type, where the type follows directly from the cipher description. In this way we are able to recover the whole trivium inner state using approx. 43 fault injections.

As mentioned above, presented attacks require many fault injections to the same Trivium inner state. This can be achieved in the chosen-ciphertext scenario, assuming that the initialisation vector is the part of the cipher input. In this case, an attacker will always use the same cipher input (initialisation vector and ciphertext) and perform the fault injection during the deciphering process. Hence, proposed attacks could be described as chosen-ciphertext fault injection attacks.

We have to stress out, that in this paper we do not consider usage of any sophisticated methods for solving systems of polynomial equations (e.g. Gröbner basis algorithms). We work with simple techniques which naturally raised from the analysis of the keystream difference equations. Hence the described attacks are easy to implement. This also shows how simple is to attack Trivium by differential fault injection. We believe that usage of more sophisticated methods can further improve presented attack, in sense of the number of the fault injections needed for the key recovery.

The rest of this paper is organised as follows. In Sect. 2 we review related work and Sect. 3 describes used notation. Trivium description in Sect. 4 follows. Attacks description can be found in Sect. 5, which also contains attack outline and differential fault analysis description. We conclude by Sect. 6.

2 Related Work

Let us briefly mention some of the previous results in Trivium cryptanalysis. Raddum introduces a new method of solving systems of sparse quadratic equations and applies it in [2] to Trivium. The complexity arising from this attack on Trivium is $O(2^{162})$. A. Maximov and A. Biryukov [3] use a different approach to solve the system of equations produced by Trivium by guessing the value of some bits. In some cases this reduces the system of quadratic equations to a system of linear equations that can be solved. The complexity of this attack is $O(c \cdot 2^{83,5})$, where c is the time taken to solve a sparse system of linear equations. Different approaches to Trivium potential cryptanalysis - some ways of construction and solution of equations system for Trivium mainly - are discussed in [4]. M. S. Thuran and O. Kara model the initialisation part of Trivium as an 8-round function in [5]. They study linear cryptanalysis of this part of Trivium and give a linear approximation of 2-round Trivium with bias 2^{-31}. In [6] the differential cryptanalysis is applied mainly to initialisation part of Trivium.

Our attack deals with fault analysis of trivium. Side-channel attacks are amongst the strongest types of implementation attacks. Short overview on passive attacks on stream ciphers is given in [7]. Differential power analysis of Trivium is described in [8].

Fault attacks on stream ciphers are studied in [9]. The authors are mainly focused on attacking constructions of stream ciphers based on LFSRs. The corresponding attacks are based on the linearity of the LFSR. More specialised techniques were used against specific stream ciphers such as RC4, LILI-128 and SOBER-t32 [10].

3 Notation

In this paper the inner state of Trivium is denoted as IS and the bits of the inner state (there are 288 of these) as $(s_1, \ldots, s_{288})$. The inner state at time t_0 is denoted as IS_{t_0} and the following keystream (starting at the time t_0) as $\{z_i\}_{i=1}^{\infty}$. We refer to this keystream as the proper keystream. After a fault injection into the state IS_{t_0}, the resulting inner state is denoted as IS'_{t_0} and the following faulty keystream (starting at the time t_0) as $\{z'_i\}_{i=1}^{\infty}$.

The keystream difference, i.e. the difference between the proper keystream and the faulty keystream is denoted as $\{d_i\}$, i.e. $d_i = z'_i \oplus z_i,\ i \geq 1$. The fault position is denoted as e, $1 \leq e \leq 288$. The righ-hand-side of an equation is (as usual) abbreviated to RHS.

4 Trivium Description

The stream cipher trivium is an additive synchronous stream cipher with 80-bit secret key and 80-bit initialisation vector (IV). The cipher itself produces the keystream, which is then XOR-ed to a plaintext to produce the ciphertext. Trivium (as other stream ciphers) can be divided into two parts: the *initialisation algorithm* described by Alg. 1, which turns the secret key and the initialisation vector into the inner state of Trivium, and the *Keystream generation algorithm* described by Alg. 2, which produces the keystream (one bit per step).

The cipher itself consists of 3 shift registers with non-linear feedback functions. These registers are of length 93, 84 and 111 respectively. Keystream production function is a bit sum (i.e. XOR) of 6 bits in total, 2 bits from each register. Feedback function for register i $(i = 0, 1, 2)$ depends on bits of register i quadratically and on one bit of register $(i + 1) \bmod 3$ linearly. If we look closer at any of these feedback functions we see, that it contains only one quadratic term and the rest is linear. Furthermore, this quadratic term is of a special type, namely $s_j \cdot s_{j+1}$.

In the rest of the paper, by the term *pair quadratic equation* we denote a quadratic equation, which contains linear terms and quadratic terms only of the type $s_j \cdot s_{j+1}$. As we will see, these pair quadratic equations are typical for Trivium and in our attack we take an advantage of this.

Algorithm 1 The Initialisation Algorithm of Trivium

```
Input:  Secret key $K = (K_1, \dots, K_{80})$, initialisation vector $IV = (IV_1, \dots, IV_{80})$
Output: Trivium inner state $(s_1, \dots, s_{288})$

1: $(s_1, \dots, s_{93}) \leftarrow (K_1, \dots, K_{80}, 0, \dots, 0)$
2: $(s_{94}, \dots, s_{177}) \leftarrow (IV_1, \dots, IV_{80}, 0, \dots, 0)$
3: $(s_{178}, \dots, s_{288}) \leftarrow (0, \dots, 0, 1, 1, 1)$
4: for $i = 0$ to $4 \cdot 288$ do
5:     $t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$
6:     $t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$
7:     $t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$
8:     $(s_1, \dots, s_{93}) \leftarrow (t_3, s_1, \dots, s_{92})$
9:     $(s_{94}, \dots, s_{177}) \leftarrow (t_1, s_{94}, \dots, s_{176})$
10:    $(s_{178}, \dots, s_{288}) \leftarrow (t_2, s_{178}, \dots, s_{287})$
11: end for
```

Algorithm 2 The Keystream generation algorithm

```
Input:  Trivium inner state $(s_1, \dots, s_{288})$, number of output bits $N \leq 2^{64}$
Output: Keystream $\{z_i\}_{i=1}^{N}$

1: for $i = 0$ to $N$ do
2:     $z_i \leftarrow s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$
3:     $t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$
4:     $t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$
5:     $t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$
6:     $(s_1, \dots, s_{93}) \leftarrow (t_3, s_1, \dots, s_{92})$
7:     $(s_{94}, \dots, s_{177}) \leftarrow (t_1, s_{94}, \dots, s_{176})$
8:     $(s_{178}, \dots, s_{288}) \leftarrow (t_2, s_{178}, \dots, s_{287})$
9: end for
```

5 Differential Fault Analysis of Trivium

In this section we will describe our contribution to the cryptanalysis of Trivium. To show how our attack evolved, we describe more versions of differential fault analysis of Trivium, from the simplest one, to the more sophisticated ones.

Before describing our contribution, let us briefly recall the basic ideas of Differential Fault Analysis (DFA).

Differential fault analysis is an active side channel attack technique, in which an attacker is able to insert a fault into the enciphering or deciphering process or he is able to insert a fault into a cipher inner state. The later is the case of our attack, we suppose that an attacker is able to change exactly one bit of the Trivium inner state. Another assumption of DFA is that an attacker is able to obtain not only the cipher output after the fault injection, but he is also able to obtain the standard output, i.e. the output produced by the cipher without the fault injection.

In this paper, according to the DFA description, we assume that an attacker is able to obtain a part of a Trivium keystream $\{z_i\}_{i=1}^{\infty}$ produced from the arbitrary but fixed inner state IS_{t_0} and that he is also able to obtain a part of the faulty keystream $\{z'_i\}_{i=1}^{\infty}$ produced by the fault inner state IS'_{t_0}. We will discuss the amount of keystream bits needed for any of presented attacks later on. Just for illustration, the attack we have implemented needs about 280 bits of the proper keystream and the same amount of the faulty keystream bits.

Furthermore, in our attacks, an attacker has to be able to do the fault injection many times, but for the same inner state IS_{t_0}, where the value of t_0 is fixed, but arbitrary and unknown. It follows, that in our scenario the stream cipher Trivium has to be run many times with the same key and IV, so an attacker is able to inject a fault to the same inner state IS_{t_0}. This can be achieved e.g. by attacking the cipher in the deciphering mode, assuming the initialisation vector is the part of the cipher input. In this case, we will always use the same pair of IV and cipher text as the cipher input, and we will perform fault injections to the cipher inner state during the deciphering process. Hence, proposed attacks can be performed in the chosen-ciphertext attack scenario.

All our prerequisites are gathered in Sect. 5.1.

The result of our attack is the determination of the inner state IS_{t_0}, which can be used afterwards to obtain the secret key K and initialisation vector IV. This can be done due to the reversibility of the Trivium initialisation algorithm and due to the fact, that the initialisation part is the same as the keystream generation part. Thus, after we determine IS_{t_0}, we run trivium backwards (which also allows us to decipher previous communication) until we obtain an inner state of the form

$$\begin{aligned}(s_1, \ldots, s_{93}) &= (a_1, \ldots, a_{80}, 0, \ldots, 0),\\ (s_{94}, \ldots, s_{177}) &= (b_1, \ldots, b_{80}, 0, \ldots, 0),\\ (s_{178}, \ldots, s_{288}) &= (0, \ldots, 0, 1, 1, 1)\end{aligned} \tag{1}$$

Afterwards if the values $(b_1, \dots, b_{80})$ are equal to the known IV, we can (with very high probability) claim, that $(a_1, \dots, a_{80})$ is the secret key used for encryption.

5.1 Attack prerequisites

Let t_0 be arbitrary but fixed positive integer and let IS_{t_0} be arbitrary but fixed Trivium inner state at time t_0. These are the prerequisites of our attack:

- an attacker is able to obtain first n consecutive bits of the keystream $\{z_i\}$ produced out of the inner state IS_{t_0},
- an attacker is able to inject exactly one fault at random position of the inner state IS_{t_0},
- an attacker is able to repeat the fault injection at random position of IS_{t_0} m times
- an attacker is able to obtain first n consecutive bits of the keystream $\{z'_i\}$ produced out of the inner sate IS'_{t_0} for all fault injections.

The number of consecutive bits of the proper and of the faulty keystream needed, n, differs for presented attacks. For the most simple one $n = 160$ and for the second one $n = 280$. The number of fault injections needed for any of the presented attacks, m, is discussed after the attack descriptions.

5.2 Attack outline

The core of the presented attack is to solve the system of equations in the inner state bits of a fixed inner state $IS_{t_0} = (s_1, \dots, s_{288})$. Because the output function of the Trivium is linear in the inner state bits, some equations can be obtained directly from the proper keystream. Specifically, the first 66 keystream bits are linear combinations of bits of IS_{t_0},

$$z_i = s_{67-i} + s_{94-i} + s_{163-i} + s_{178-i} + s_{244-i} + s_{289-i},\ 1 \le i \le 66, \tag{2}$$

and the following 82 keystream bits depends quadratically on $s_1, \dots, s_{288}$. These are followed by polynomials of degree 3 and higher.

Since we have 288 variables (bits of inner state) and the degree of keystream equations grows very fast, we are not able to efficiently solve this polynomial system. The question is how to obtain more equations. By the analysis of Trivium, we have noticed that a change of a single bit of the inner state directly reveals some inner state bits and also gives us some more equations. Hence, we decided to use DFA as a method for obtaining more equations. For illustration, Tab. 1 contains the first equations given by a keystream difference after a fault injection on position 3, i.e. $s'_3 = s_3 + 1$.

Table 1. Non-zero elements of $\{d_i\}_{i=1}^{230}$ for $s'_3 = s_3 + 1$.

i	d_i
64,91,148,175,199,211,214,217	1
158,175,224	s_4
159,174,225	s_3
212	$s_4 + s_{31} + s_{29}s_{30} + s_{109}$
213	$s_2 + s_{29} + s_{27}s_{28} + s_{107}$
227	$s_{178} + s_{223} + s_{221}s_{222} + s_4$
228	$s_{161} + s_{176} + s_{174}s_{175} + s_{263} + s_{221} + s_2 + s_{219}s_{220}$

5.3 Fault position determination

In our attack we suppose that an attacker is not able to influence the position of a fault injection, i.e. he can inject a fault to the Trivium inner state only at a random position. But as we will see further on, he also needs to determine the fault position, since the equations for the keystream difference depend on this position.

The core of the fault position determination is that the distance between the output bits differs for each register. According to the line 2 of Alg. 2, s_{66} and s_{93} are the output bits of the first register and their distance is $93 - 66 = 27$. In the second register, s_{162} and s_{177} are used as the output bits and their distance is 15. In the third register s_{243} and s_{288} are used and their distance equals 45.

For example suppose that we have injected a fault into one of the registers at an unknown position e, so $s'_e = s_e + 1$, and (only for this example) assume that we know that $e \in \{1, \ldots, 66\} \cup \{94, \ldots, 162\} \cup \{178, \ldots, 243\}$. Denote the index of the first non-zero bit in the keystream difference by a, i.e. $d_a = 1$ and $d_j = 0$ for all $1 \leq j < a$. If the fault was injected into the first register, then according to the output function we have also $d_{a+27} = 1$, since the distance between the output bits of the first register is 27. In the same manner, if the fault was injected to the second register, we have $d_{a+15} = 1$ while $d_{a+27} = 0$, and $d_{a+45} = 1$ while $d_{a+15} = d_{a+27} = 0$ for the third register.

A non-zero bit can occur in the keystream difference at many positions, depending on the values of inner state bits. But since some occurrences of the non-zero bit are certain, with a little bit more work that in our example, we can easily determine the exact fault position. Tables 6, 7 and 8 in Appendix show the positions and the values of the some first (potentially) nonzero bits of the keystream difference in the correspondence to the fault position. In these tables, symbol X denotes a value which is neither 1 nor s_{i+1} or s_{i-1}. By the help of these tables, it is easy to determine the fault position assuming that exactly one fault was injected (our assumption from 5.1). E.g. assume that the first non-zero keystream difference bit has index a, the second non-zero bit has index b and that $b - a = 42$. According to the tables 6, 7 and 8, we see that this can happen only in the case described by the third row of Tab. 6 ($136 - 94 = 42$) and in the case described by the third row of Tab. 8 ($331 - 289 = 42$). In the first case we will also have $d_{b+42} = 1$ ($178 - 136 = 42$) and $d_{b+24} = 0$, while in the second case $d_{b+42} = 0$ and $d_{b+24} = 1$ $(355 - 331 = 24)$. In this way we can distinguish between the two cases and we can claim that the fault position is $94 - a$ in the first case and $289 - a$ in the second case.

5.4 First attack, using linear equations

Let us start with the description of a simple attack on Trivium using fault injection technique.

In this attack an attacker uses only linear equations in the inner state bits $(s_1, \ldots, s_{288})$ given by the proper keystream and by the keystream difference.

Before the attack itself, the attacker does the following precomputation: for each fault position $1 \leq e \leq 288$, the attacker expresses potentially non-zero bits of the keystream difference as polynomials in $(s_1, \ldots, s_{288})$ over GF(2), using Alg. 2 and the fact that $d_i = z_i' \oplus z_i$. Since he only needs linear equations, the attacker has to express only the first n bits of keystream difference and store these equations in a table. The value of n is discussed below. During the attack, the attacker will just make table look-up for the right equation for the actual fault position.

The average number of linear equations given by the keystream difference for a single fault injection can be found in Tab. 2. It follows from this table, that in the precomputation phase of this attack, it is enough to make 180 steps of Trivium (in a symbolic computation) for each fault position, so $n = 180$.

Table 2. The average number (among all fault positions) of equations obtained from a random fault.

number of steps	The average number of equations of degree d obtained from one fault.						
	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$
160	1.86	0.24	0.08	0	0	0	0
180	1.99	1.17	0.39	0	0	0	0
200	1.99	2.52	0.89	0	0	0	0
220	1.99	4.14	1.53	0	0	0	0
240	1.99	5.99	2.82	0.03	0	0	0
260	1.99	7.76	4.15	1.13	0.45	0.37	0.28
280	1.99	9.22	5.22	3.42	1.47	1.23	0.96
300	1.99	9.77	5.86	7.10	3.55	2.66	2.09

Let us have a closer look on the relation between the number of fault injections and the number of inner state bits obtained. At the beginning of the attack, almost every fault injected gives us directly two new variables. But as the attack progresses, it becomes much harder to hit the positions which will bring us two new inner state bits and there will be many fault injections, which bring only one or even no new variable. If we would like to obtain all of the 288 bits of the inner state just by the fault injection, at the end of the attack we will waste many fault injections until we hit the right positions. Hence, it will significantly reduce the number of fault injections needed, if we stop the attack at the point when T bits of inner state are known and we will guess the remaining $288 - T$ bits afterwards.

Table 3 shows the number of fault injections needed, m, to obtain T bits of inner state for different values of T. This is also illustrated on the left of Fig. 1.

Table 3. Number of fault injections needed (m) to obtain a certain number of inner state bits (T) (average over 1000 experiments) in the linear attack

T	20	40	60	80	100	120	140	160	180	200	220	240	260	280	288
m	10	21	33	46	58	71	84	98	113	127	145	165	189	270	382

During the attack, the attacker stores linear equations obtained in a binary matrix M with 289 columns (288 bits of IS_{t_0} plus RHS). The attack itself is described by Alg. 3.

Algorithm 3 Linear attack

```
Input:  Trivium stream cipher with possibility of fault injections (see Sect. 5.1)
Output: Secret key $K$
 1: obtain $n$ consecutive bits of $\{z_i\}$
 2: insert linear equations (2) (see Sect. 5.2) to $M$, using bits of $\{z_i\}$ as RHS
 3: while rank($M$) $< T$ do
 4:    insert a fault into $IS_{t_0}$
 5:    obtain $n$ consecutive bits of the faulty keystream $\{z'_i\}$
 6:    compute keystream difference $d_i = z'_i + z_i,\ 1 \leq i \leq n$
 7:    determine the fault position $e$
 8:    insert equations for the keystream difference (according to value of $e$) into $M$
 9:    do Gauss elimination of $M$
10: end while
11: repeat
12:    guess the remaining $288 - T$ inner state bits
13:    solve the linear system given by $M$ and guessed bits
14:    store the solution in $IS_S$
15:    produce the keystream $\{\tilde{z}_i\}_{i=1}^{n}$ from the inner state $IS_S$
16: until $\exists\ i \in \{1, \ldots, n\} : \tilde{z}_i \neq z_i$
17: run Trivium backwards starting with $IS_S$ until an inner state $IS_0 = (s_1^0, \ldots, s_{288}^0)$
    of type (1) (see page 5) is reached
18: output $K = (s_1^0, \ldots, s_{80}^0)$.
```

As already mentioned, according to Tab. 2, in Alg. 3 we can set $n = 180$, since we would not get any more (previously unknown) linear equations from the bits of the keystream difference $\{d_i\}$ for $i > 180$.

The complexity of this simple attack is given by the complexity of solving a system of linear equations (suppose this is $O(n^3)$) multiplied by the complexity of guessing $288 - T$ variables. For $T = 258$, we obtain the complexity of $258^3 \cdot 2^{30} = 2^{54}$ operations and we need to do (according to Tab. 3) approximately 189 fault injections. For $T = 268$, the attack has the complexity of $2^{44.2}$ operations and we need to do approximately 200 fault injections.

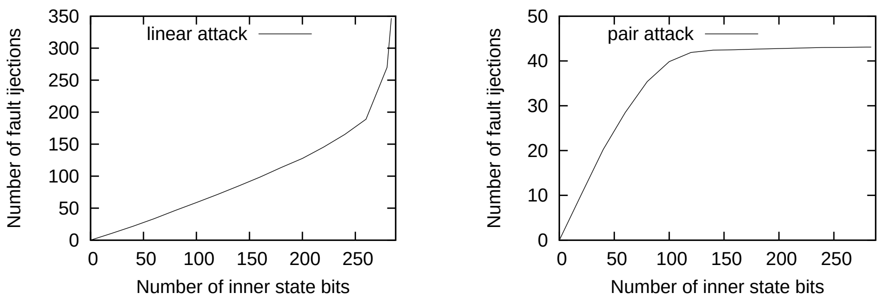

Fig. 1. Number of fault injections needed to obtain a certain number of inner state bits. Left: linear attack. Right: pair quadratic attack.

5.5 Second attack, using pair quadratic equations

The attack presented in this section is a natural successor of the previous one, in the terms of reduction of the number of fault injections needed.

The main difference is that in this case, we do not use only linear equations but also *pair* quadratic equations (see Sect. 4). The reason why we have decided to use only pair quadratic and not all quadratic equations is that most of quadratic equations that appear in Trivium analysis are pair equations. For example all 82 quadratic equations for keystream bits are pair and also most (approx. 80%) of the quadratic equations for the keystream difference bits are also pair (see Tab. 4). Furthermore, the number of all possible quadratic terms in 288 inner state bits is too large (approx. $2^{16.3}$) and hence the complexity of solving a linear system in all quadratic variables would be too high.

Lets have a look at the precomputation part of this attack. Here again, for all possible fault positions $1 \leq e \leq 288$, we need to express bits of the keystream difference as polynomials in the bits of $IS_{t_0} = (s_1, \ldots, s_{288})$. However, now we will store not only all linear but also all pair quadratic equations for each value of e. Hence for each e we need to do more Trivium steps (in symbolical computing) for both the proper inner state IS_{t_0} and for the fault inner state IS'_{t_0}.

As common in cryptology, to simplify the computations we work in the factor ring $\mathrm{GF}(2)[s_1, \ldots, s_{288}]/(s_1^2 - s_1, \ldots, s_{288}^2 - s_{288})$ instead of the whole polynomial ring $\mathrm{GF}(2)[s_1, \ldots, s_{288}]$. This can be done, since for any $x \in \mathrm{GF}(2)$ we have $x^2 = x$ and we would like to make these computations as simple as possible. In this way we also obtain more equations of small degrees, since this factorisation reduces any term of type s_i^n to the term s_i.

In our implementation, we have used the value $n = 280$, so we need to obtain 280 bits of a keystream and we need to do 280 steps of Trivium (in symbolical computing) during the precomputation. We will not theoretically discuss the complexity of these precomputations, but in our implementation the precomputation phase with 280 Trivium step took couple of hours on a standard desktop computer.

The average number of equations of a degree up to 7 given by the keystream difference for a single fault injection and for a certain number of Trivium steps can be found at Tab. 2. Table 4 describes number of pair quadratic equations given by the keystream difference for a single fault injection and certain number of Trivium steps and it compares this number to the amount of all quadratic equations obtained. As we can see, the loss is around 20%.

Table 4. The average number (among all fault positions) of pair quadratic equations obtained from a random fault compared to the average number of all quadratic equations.

number of steps	avg. num. of all quad. eq.	avg. num. of pair quad. eq.	loss (percentage)
160	0.24	0.19	18.84%
180	1.17	0.94	19.64%
200	2.52	1.98	21.63%
220	4.14	3.19	22.75%
240	5.99	4.75	20.75%
260	7.76	6.08	21.66%
280	9.22	7.14	22.52%

During this attack, we store the equations obtained in a matrix M over GF(2) which has $288 + 287 + 1$ columns. The first 288 columns will represent the variables $s_1, \ldots, s_{288}$ and the following 287 columns will represent all pair quadratic terms. We denote the variable of a column j by y_j and define

$$y_j = \begin{cases} s_j, & 1 \leq j \leq 288 \\ s_{j-288} \cdot s_{j-287}, & 289 \leq j \leq 575. \end{cases}$$

The last column contains the right-hand-side value for each equation. At the beginning of the attack, we put all linear and pair quadratic equations obtained from the proper keystream into M. Afterwards for each fault injection, we make fault position determination and according to the fault position, we insert the precomputed equations for the actual fault position.

In addition to the previous attack, we also hold a list of already known variables. This list will help us employ quadratic connections between variables y_i. Strictly speaking, anytime we determine the value of some previously unknown variable y_i, for some $1 \leq i \leq 288$ (so $y_i = s_i$), we go through the whole matrix M and we eliminate variables y_{i+287} and y_{i+288} in each row (only y_{289} for $i = 1$ and only y_{575} for $i = 288$). If we for example determine that for some $1 \leq i \leq 288$, $y_i = s_i = 1$, then we go through all rows of M with non-zero value in column y_{i+287} or y_{i+288}, set this variable to zero and add 1 to y_{i-1} or y_{i+1} respectively. In the case of $y_i = 0$ for some $1 \leq i \leq 288$, we just set y_{i+287} and y_{i+288} to zero. In this way, we can possibly obtain some more linear equations in $s_1, \ldots, s_{288}$ or even determine some new variables. If this is the case, we repeat this procedure again. For the rest of the paper, we denote this procedure as *quadratic_to_linear()*.

In the description of the attack, we suppose that the list of known variables is updated automatically, so we do not mention this explicitly. E.g. 2 new variables will be added automatically to the list of known variables after almost each fault injection.

During the attack, we also use classical linear algebra techniques for solving a system of linear equations in y_i, $1 \leq i \leq 575$, represented by M. Let us denote this by *elimination()*. It is clear, that this procedure can also reveal some new variables. If this happens, we use these new variables for the *quadratic_to_linear()* procedure and if this changes at least one equation in M, we do the *elimination()* again.

The attack is described by Alg. 4.

Algorithm 4 Attack using pair equations

```
Input:  Trivium stream cipher with possibility of fault injections (see Sect. 5.1)
Output: Secret key $K$
 1: obtain $n$ consecutive bits of $\{z_i\}$
 2: insert equations for $\{z_i\}_{i=1}^{148}$ to $M$, using bits of $\{z_i\}$ as RHS
 3: while not (all $s_1, \ldots, s_{288}$ are known) do
 4:    insert a fault into $IS_{t_0}$
 5:    obtain $n$ consecutive bits of the faulty keystream $\{z'_i\}$
 6:    compute keystream difference $d_i = z'_i + z_i,\ 1 \leq i \leq n$
 7:    determine the fault position, $e$
 8:    insert equations for the keystream difference (according to value of $e$) into $M$
 9:    repeat
10:       do quadratic_to_linear()
11:    until it keeps changing $M$
12:    do elimination()
13:    if new variables obtained by elimination() then
14:       goto 9
15:    end if
16: end while
17: run Trivium backwards starting with $IS = (s_1, \ldots, s_{288})$ until an inner state $IS_0 =$
    $(s_1^0, \ldots, s_{288}^0)$ of type (1) (see page 5) is reached
18: output $K = (s_1^0, \ldots, s_{80}^0)$.
```

We have not theoretically analysed the complexity of this algorithm, but its running time on a standard desktop computer was always only a couple of seconds.

The average number of fault injections needed to obtain a certain number of inner states bits by the described attack is shown on Tab. 5 and illustrated on the right of the Fig. 1. These experimental results show, that the behaviour of Alg. 4 is opposite to the behaviour of Alg. 3. In this case, if we would like to obtain only 100 inner state bits, we need to inject approx. 40 faults and for 288 inner state bits we need only approx. 43 faults. It follows that stopping Alg.4 earlier and guessing the remaining variables would be of no significance.

Table 5. Number of fault injections needed (m) to obtain a certain number of inner state bits (T) (average over 1000 experiments) by Alg. 4

T	20	40	60	80	100	120	140	160	180	200	220	240	260	280	288
m	10.1	20.3	28.4	35.4	39.8	41.9	42.4	42.5	42.7	42.8	42.9	43.0	43.1	43.1	43.1

5.6 Possible extensions, future work

In this section we will shortly describe some possible extensions of the previous attack.

The first extension could be an algorithm, which would use all quadratic equations and not only pair equations. The size of the matrix M would be much higher in this case. However, since the matrix would be sparse, it could be represented and handled efficiently. According to Tab. 4, we would get approx. 2 more equations from each fault, so this would reduce the number of faults needed to less than 40.

Next possible extension would be an attack, which would also use equations of higher order. This doesn't necessarily mean that we would try to solve systems of polynomial equations. We could only store these equations, and then use a function similar to the *quadratic_to_linear()* to eliminate terms of higher order. E.g. if we will decide to use equations up to degree 3, we could possible eliminate cubic terms and get some new equations of degree 2. E.g. for fault position 95 we have

$$\begin{aligned} d_{256} &= s_{96}s_{81}s_{82} + s_{96}s_{56} + s_{96}s_{83} + s_{96}s_{161} + s_{96}s_{98} + s_{96}s_{97} + s_{96}s_{185} = \\ &= s_{96} \cdot (s_{81}s_{82} + s_{56} + s_{83} + s_{161} + s_{98} + s_{97} + s_{185}) \end{aligned}$$

so if $s_{96} = 1$ we get new a pair equation. In this way we could obtain more equations, which could be used in the previous attack. This would further reduce the number of fault injections needed.

By adjusting the prerequisites, we can obtain other improvements. E.g. if an attacker can choose the fault position, the number of fault injections needed for the proposed attacks would significantly reduce. Yet another option could be injection of more faults at once. It is clear, that in this case an attacker would obtain much more information from each fault injection (e. g. if two faults are injected at once, 4 inner state bits are obtained directly). Hence, an attack could be carried out using only few fault injection. On the other hand, the fault position determination could be problematic.

6 Conclusion

In this paper, differential fault analysis of Trivium was described. As far as we know, this was the first time differential fault analysis was applied to a stream cipher based on non-linear shift registers.

We have shown, that an attacker is able to obtain the secret key after approximately 43 fault injections using one of the described algorithms, assuming the chosen-ciphertext attack scenario. All the methods proposed in this article arise directly from the Trivium analysis, they are of low complexity and are easy to implement.

Acknowledgement

The authors would like to thank Martin Hlaváč for his helpful comments and the anonymous reviewers of the article for their careful readings and valuable suggestions.

References

1. De Cannière, C., Preneel, B.: Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/30, http://www.ecrypt.eu.org/stream (2005)
2. Raddum, H.: Cryptanalytic Results on Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/039, http://www.ecrypt.eu.org/stream (2006)
3. Maximov, A., Biryukov, A.: Two Trivial Attacks on Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/006, http://www.ecrypt.eu.org/stream (2007)
4. Babbage, S.: Some Thoughts on Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/007, http://www.ecrypt.eu.org/stream (2007)
5. Turan, M.S., Kara, O.: Linear Approximations for 2-round Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/008, http://www.ecrypt.eu.org/stream (2007)
6. Biham, E., Dunkelman, O.: Differential Cryptanalysis in Stream Ciphers. COSIC internal report (2007)
7. Rechberger, Ch., Oswald, E.: Stream Ciphers and Side-Channel Analysis. SASC 2004 - The State of the Art of Stream Ciphers, Workshop Record, pp. 320-326. http://www.ecrypt.eu.org/stream (2004)
8. Fischer, W., Gammel, B. M., Kniffler, O., Velten, J.: Differential Power Analysis of Stream Ciphers. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/014, http://www.ecrypt.eu.org/stream (2007)

9. Hoch, J. J., Shamir, A.: Fault Analysis of Stream Ciphers. In: Joye, M., Quisquater, J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 240-253. Springer (2004)
10. Biham, E., Granboulan, L., Nguyen, Ph.: Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. SASC 2004 - The State of the Art of Stream Ciphers, Workshop Record, pp. 147-155. http://www.ecrypt.eu.org/stream (2004)
11. Courtois, N., Klimov, A., Patarin, J., Shamir, A.: Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In: Preneel, B. (eds.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 392-407. Springer (2004)

Appendix

Table 6. Non-zero elements of $\{d_j\}$, with $s'_i = s_i + 1$, for $1 \leq i \leq 93$.

fault pos.	keystream difference d_j for $j =$												
$s'_i = s_i + 1$	67-i	94-i	136-i	151-i	161-i	162-i	163-i	176-i	177-i	178-i	202-i	220-i	242-i
$i = 1$	1	1		1	s_{i+1}	X		s_{i+1}	X	1	1		
$i = 2, \ldots, 66$	1	1		1	s_{i+1}	s_{i-1}		s_{i+1}	s_{i-1}	1	1		
$i = 67, \ldots, 69$		1	1		s_{i+1}	s_{i-1}		s_{i+1}	s_{i-1}	1		1	
$i = 70, \ldots, 90$		1			s_{i+1}	s_{i-1}	1	s_{i+1}	s_{i-1}	1			1
$i = 91$		1			s_{i+1}	s_{i-1}	1	s_{i+1}	s_{i-1}	1			
$i = 92$		1				s_{i-1}	1		s_{i-1}	1			
$i = 93$		1					1			1			

Table 7. Non-zero elements of $\{d_j\}$, with $s'_i = s_i + 1$, for $94 \leq i \leq 177$.

fault pos.	keystream difference d_j for $j =$											
$s'_i = s_i + 1$	163-i	178-i	229-i	241-i	242-i	243-i	244-i	256-i	274-i	287-i	288-i	289-i
$i = 94$	1	1	1	1	s_{i+1}	X	1	1	1	s_{i+1}	X	1
$i = 95, \ldots, 162$	1	1	1	1	s_{i+1}	s_{i-1}	1	1	1	s_{i+1}	s_{i-1}	1
$i = 163, \ldots, 171$		1		1	s_{i+1}	s_{i-1}	1	1		s_{i+1}	s_{i-1}	1
$i = 172, \ldots, 174$		1			s_{i+1}	s_{i-1}	1			s_{i+1}	s_{i-1}	1
$i = 175$		1			s_{i+1}	s_{i-1}	1			s_{i+1}	s_{i-1}	1
$i = 176$		1				s_{i-1}	1				s_{i-1}	1
$i = 177$		1					1					1

Table 8. Non-zero elements of $\{d_j\}$, with $s'_i = s_i + 1$, for $178 \leq i \leq 288$.

fault pos.	keystream difference d_j for $j =$											
$s'_i = s_i + 1$	289-i	310-i	331-i	371-i	353-i	354-i	355-i	376-i	380-i	381-i	382-i	394-i
$i = 178$	1	1	1	1	s_{i+1}	X	1	1	s_{i+1}	X	1	1
$i = 179, \ldots, 243$	1	1	1	1	s_{i+1}	s_{i-1}	1	1	s_{i+1}	s_{i-1}	1	1
$i = 244, \ldots, 264$	1		1		s_{i+1}	s_{i-1}	1	1	s_{i+1}	s_{i-1}	1	
$i = 265, \ldots, 285$	1				s_{i+1}	s_{i-1}	1		s_{i+1}	s_{i-1}	1	
$i = 286$	1				s_{i+1}	s_{i-1}	1		s_{i+1}	s_{i-1}	1	
$i = 287$	1					s_{i-1}	1			s_{i-1}	1	
$i = 288$	1						1				1	