Using SAT Solving to Improve Differential Fault Analysis of Trivium

Mohamed Saied Emam Mohamed[1], Stanislav Bulygin[2] and Johannes Buchmann[1]

[1] *TU Darmstadt, FB Informatik*
Hochschulstrasse 10, 64289 Darmstadt, Germany

[2] *Center for Advanced Security Research Darmstadt (CASED)*

{fmohamed, buchmanng}@cdc.informatik.tu-darmstadt.de,
Stanislav.Bulygin@cased.de

Abstract

Combining different cryptanalytic methods to attack a cryptosystem became one of the hot topics in cryptanalysis. In particular, algebraic methods in side channel and differential fault analysis (DFA) attracted a lot of attention recently. In [9], Hojśik and Rudolf used DFA to recover the inner state of the stream cipher Trivium which leads to recovering the secret key. For this attack, they required 3.2 one-bit fault injections on average and 800 keystream bits. In this paper, we give an example of combining DFA attacks and algebraic attacks. We use algebraic methods to improve the DFA of Trivium [9]. Our improved DFA attack recovers the inner state of Trivium by using only 2 fault injections and only 420 keystream bits.

Keywords: *Differential Fault Analysis, algebraic attack, SAT-Solvers, Trivium*

1. Introduction

Stream ciphers are encryption algorithms that encrypt plaintext digits one at a time. Trivium is a hardware-oriented synchronous stream cipher [4]. It was selected in phase three of profile two of the eSTREAM project [12]. Due to its simplicity and speed, it can provide reliable security service for many hardware applications, such as wireless connections and mobile telecommunication. In order to assess the security of these applications, one can use cryptanalytic methods.

A differential fault analysis (DFA) is a method to analyse a cipher by examining and affecting its implementation. The idea is to induce a physical corruption to the internal state of the cipher. This leads to producing some information about the internal data that helps to recover the secret of the cipher. At INDOCRYPT 2008, M. Hojśik and B. Rudolf introduced a differential fault analysis of the stream cipher Trivium [9] by using the floating representation of Trivium instead of the classical representation of the cipher. The basic idea of this attack is to inject a one-bit fault into the inner state of Trivium. In this case an attacker can generate, in addition to the equations system that represents a set of keystream bits, some lower degree polynomial equations that relate a set of keystream bits generated after the injection performed. Using this method one needs 3.2 fault injections on average and 800 keystream bits to recover the inner state of Trivium at certain time $t = t_0$, which leads to recovering the secret key of the cipher.

Instead of attacking Trivium with only one method, an attacker can gain more power by combining di_erential fault analysis with algebraic techniques. In this paper, we improve the above attack by using a SAT solver to speed up the solving part of the attack, as well as we

improve the equation preprocessing phase. In this case, attacker needs exactly 2 one-bit fault injections and 420 keystream bits to recover the inner state of Trivium.

This paper is organized as follows. In Section 2 we describe the floating representation of Trivium. In Section 3 we briefly explain the differential fault analysis of Trivium and in Section 4 we explain the generation of the polynomial equation system that represents the inner state of Trivium and the DFA of it. Our attack description and the results are presented in Section 5 and Section 6 respectively. Finally, we conclude the paper in Section 7.

2. Algebraic Description of Trivium

In this section, we describe the algebraic representation of Trivium. Trivium generates a sequence of keystream bits from an 80-bit secret key and an 80-bit initial vector (IV). The inner state of Trivium consists of 288 bits which are stored in three shift registers respectively as explained in Figure 1. The so called floating representation [9] of Trivium is as follows.

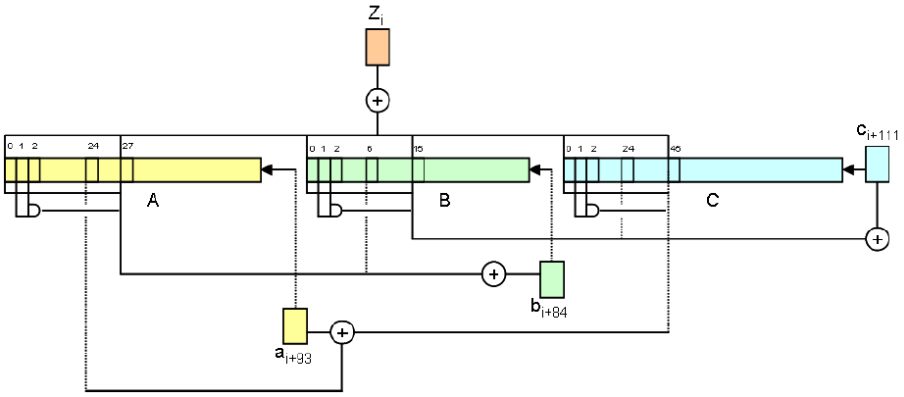

Figure 1. Trivium Construction, $i \geq 0$:

The inner state at time t is

$$(a_{t+1}, \ldots, a_{t+93}, b_{t+1}, \ldots, b_{t+84}, c_{t+1}, \ldots, c_{t+111})$$

We use the secret key $K = (k_1, \ldots, k_{80})$ and IV = $(v_1, \ldots, v_{80})$ to initialize the inner state as follows

$$(\underbrace{0, \ldots, 0, k_{80}, \ldots, k_1}_{A}, \underbrace{0, 0, 0, 0, v_{80}, \ldots, v_1}_{B}, \underbrace{1, 1, 1, 0, \ldots, 0}_{C})$$

In the initialization phase, Trivium loops 1152 times without producing any keystream. Let at time $t = 0$ (directly after the initialization phase), the inner state of Trivium be

$$(a_1, \ldots, a_{93}, b_1, \ldots, b_{84}, c_1, \ldots, c_{111})$$

The keystream bits z_i and the new inner state bits a_{i+93}, b_{i+84}, c_{i+111} of Trivium registers A, B, C are generated as follows.

$$z_i = a_i + a_{i+27} + b_i + b_{i+15} + c_i + c_{i+45}, \quad i \geq 1 \qquad (1)$$

$$a_{i+93} = a_{i+24} + c_{i+45} + c_i + c_{i+1} \cdot c_{i+2} \quad (2.1)$$
$$b_{i+84} = b_{i+6} + a_{i+27} + a_i + a_{i+1} \cdot a_{i+2} \quad (2.2)$$
$$c_{i+111} = c_{i+24} + b_{i+15} + b_i + b_{i+1} \cdot b_{i+2} \quad (2.3)$$

By using equations (2.1)-(2.3), we can clock Trivium forward to generate new inner states and backward to recover previous inner states.

3. Preliminaries on the DFA of Trivium

We use the same assumptions as in [9]. Namely, the attacker is able to inject a one-bit fault at a random position within the inner state at $t = 0$. Also, he can obtain the first n keystream bits z_i, $1 \le i \le n$, before any fault injections and after a fault injection which we call z'_i, $1 \le i \le n$. The attacker can do the previous step several times with the same secret key and IV.

Each one-bit fault injection leads to additional equations. We use the difference of keystream outputs ($\Delta z_i = z_i + z'_i$) and shift registers inputs (Δa_i, Δb_i, Δc_i) before and after performing an injection to generate additional polynomial equations as in (3) and (4.1)-(4.3).

$$\triangle z_i = \triangle a_i + \triangle a_{i+27} + \triangle b_i + \triangle b_{i+15} + \triangle c_i + \triangle c_{i+45}, \quad i \ge 0 \quad (3)$$

$$\triangle a_{i+93} = \triangle a_{i+24} + \triangle c_{i+45} + \triangle c_i + \triangle(c_{i+1} \cdot c_{i+2}) \quad (4.1)$$
$$\triangle b_{i+84} = \triangle b_{i+6} + \triangle a_{i+27} + \triangle a_i + \triangle(a_{i+1} \cdot a_{i+2}) \quad (4.2)$$
$$\triangle c_{i+111} = \triangle c_{i+24} + \triangle b_{i+15} + \triangle b_i + \triangle(b_{i+1} \cdot b_{i+2}) \quad (4.3)$$

In this case the difference values of the inner state at $t = 0$,

$$(\triangle a_1, \cdots, \triangle a_{93}, \triangle b_1, \cdots, \triangle b_{84}, \triangle c_1, \cdots, \triangle c_{111}),$$

are zeros everywhere except at the fault injection position. For example, suppose that the injected bit is a_{35} then the difference vector will be

$$(0, \cdots, 0, \triangle a_{35} = 1, 0, \cdots, 0)$$

The fault position is not known *a priori*, but it can be determined by observing the faulty keystream as in [9]. So we assume that we know it. In the next section, we explain how we use this differential model to generate the additional polynomial equations produced after inserting several one-bit fault injections.

4. Generating Low Degree Polynomial Equations

We explain the generation of the polynomial equation system that we use in our attack. As defined in Section 2, we let the inner state at time t = 0 of Trivium be

$$(a_1, \ldots, a_{93}, b_1, \ldots, b_{84}, c_1, \ldots, c_{111}).$$

Also, we suppose that we have an n-bit keystream output vector $Z = (z_1, \ldots, z_n)$.

The TRIV procedure constructs the polynomial equation system that describes the generation of n keystream bits using the 288 inner state bits at $t = 0$ as in Algorithm 1. It uses the strategy of generating low degree polynomials. In this case, we represent each new inner state bit generated by one of the shift registers A, B, C (a_{i+93}, b_{i+84}, c_{i+111}, $1 \le i \le n$, respectively) as a new internal variable. By adding them to the 288 initial inner state bit variables, we have totally 3n + 288 variables ($a_1, \ldots, a_{n+93}, b_1, \ldots, b_{n+84}, c_1, \ldots, c_{n+111}$). We

call them the *inner state variables*.

Algorithm 1 TRIV($Z = (z_1, \ldots, z_n)$)

1: $P \leftarrow \emptyset$
2: **for** $i = 1$ to n **do**
3: $P \leftarrow P \cup \{a_i + a_{i+27} + b_i + b_{i+15} + c_i + c_{i+45} + z_i\}$ // Eq. (1)
4: $P \leftarrow P \cup \{a_{i+93} + a_{i+24} + c_{i+45} + c_i + c_{i+1} \cdot c_{i+2}\}$ // Eq. (2.1)
5: $P \leftarrow P \cup \{b_{i+84} + b_{i+6} + a_{i+27} + a_i + a_{i+1} \cdot a_{i+2}\}$ // Eq. (2.2)
6: $P \leftarrow P \cup \{c_{i+111} + c_{i+24} + b_{i+15} + b_i + b_{i+1} \cdot b_{i+2}\}$ // Eq. (2.3)
7: **end for**
8: **return** P

TRIV takes the keystream bit vector *Z* as an input and uses equations (1) and (2.1)-(2.3) to generate the polynomial equations. It returns the set of polynomials *P* that describes *Z* using the inner state variables. The system of equations $\{p = 0, p \in P\}$ contains 4*n* polynomial equations (*n* linear and 3*n* quadratic produced by (1) and (2.1)-(2.3) respectively) in the 3*n* + 288 inner state variables. We call it the pure system of Trivium without using DFA.

Now we are going to explain how we generate the additional low degree polynomial equations that are obtained from the faulty key stream. For this we use the differential model that was explained in the previous section. The EQgenerator procedure constructs such equations as described in Algorithm 2. It takes as inputs *m* fault injection positions ($l_1, \ldots, l_m$), the keystream vector *Z* before any fault injections and *m* keystream vectors $Z^{(1)}, \ldots, Z^{(m)}$ obtained after each one of the *m* fault injections, where each keystream vector $Z^{(j)} = (z^{(j)}_1, \ldots, z^{(j)}_n)$, $1 \le j \le m$.

EQgenerator initializes the set of polynomial equations (*P*) with the set of the pure polynomials returned by the TRIV procedure (line 1). It initializes the arrays *a*, *b*, *c* with the inner state variables (lines 2...4).

For each fault injection *j*, $1 \le j \le m$, EQgenerator sets the arrays *da*, *db*, *dc* that store the polynomials representing the *inner state difference variables*

$$(\triangle a_i)_{i=1}^{n+93}, (\triangle b_i)_{i=1}^{n+84}, (\triangle c_i)_{i=1}^{n+111}$$

to zeros (lines 7...9). In line 10 (Inject(*da*, *db*, *dc*; l_j)), EQgenerator inserts a fault to the inner state ($a_1, \ldots, a_{93}, b_1, \ldots, b_{84}, c_1, \ldots, c_{111}$) as follows. Let the fault injection position be $l_j \le n$. EQgenerator sets $da[l_j]$ to 1 when $l_j \le 93$. In case of $93 < l_j \le 177$, it sets $db[l_j - 93]$ to 1. Otherwise, it sets $dc[l_j - 177]$ to 1.

For each keystream bit z_i and the corresponding faulty keystream bit $z^{(j)}_i$, $1 \le i \le n$, EQgenerator evaluates the key stream output difference *dz*. Then, it uses (3) to generate an additional polynomial and includes this polynomial to *P* (lines 13, 14). Also, by using the fact that

$$\triangle(x \cdot y) = \triangle x \cdot y + x \cdot \triangle y + \triangle x \cdot \triangle y,$$

we can reconstruct equations (4.1), (4.2), and (4.3) as follows.

$$\triangle a_{i+93} = \triangle a_{i+24} + \triangle c_{i+45} + \triangle c_i + \triangle c_{i+1} \cdot c_{i+2} + c_{i+1} \cdot \triangle c_{i+2} + \triangle c_{i+1} \cdot \triangle c_{i+2} \quad (5.1)$$

$$\triangle b_{i+84} = \triangle b_{i+6} + \triangle a_{i+27} + \triangle a_i + \triangle a_{i+1} \cdot a_{i+2} + a_{i+1} \cdot \triangle a_{i+2} + \triangle a_{i+1} \cdot \triangle a_{i+2} \quad (5.2)$$

$$\triangle c_{i+111} = \triangle c_{i+24} + \triangle b_{i+15} + \triangle b_i + \triangle b_{i+1} \cdot b_{i+2} + b_{i+1} \cdot \triangle b_{i+2} + \triangle b_{i+1} \cdot \triangle b_{i+2} \quad (5.3)$$

The EQgenerator procedure uses these equations to construct the polynomial entries of the difference polynomial arrays (*da*, *db*, *dc*).

EQgenerator extracts from *P* all possible univariate polynomials of the form $x + v$, where x is an inner state variable and v is 0 or 1 (line 20). If there are such univariates, it simplifies *P* by substituting the solved variables in each polynomial $p \in P$ (line 21). It repeats these two steps as long as it generates more univariates (lines 18...23). After that, EQgenerator uses all univariates produced from the previous loop to simplify the elements of the arrays *da*, *db*, *dc*; *a*, *b*, *c* (line 24).

Finally, the EQgenerator procedure returns the set of generated polynomials *P* together with the set of generated univariates *S*.

In the EQgenerator procedure, we use the same way of generating polynomial equations as in [9]. However, the authors of [9] substituted by the solved variables only in the higher degree generated polynomials (*P*), whereas we substitute solved variables in all the generated polynomials (linear and non-linear) and all constructed Δa_i, Δb_i, Δc_i polynomials. Moreover, we replace the solved variables by their values in the set of variables to prevent their occurrence in the remaining computation. In Table 1 we compare the Hojśik-Rudolf (H-R) polynomial system generator [9] with our generator. For this comparison, we have $n = 800$ and denote the number of fault injections by *m*. We report the number of the produced polynomial equations of degree ≤ 4. Clearly, our generator creates systems that contain more linear equations than those created by H-R. This leads to an easier system to solve. We evaluate the average over 1000 experiments.

Algorithm 2 EQgenerator$(l_1, \ldots, l_m, Z, Z^{(1)}, \ldots, Z^{(m)})$

```
 1: P ← TRIV(Z)
 2: a ← [a_1, ..., a_{n+93}]
 3: b ← [b_1, ..., b_{n+84}]
 4: c ← [c_1, ..., c_{n+111}]
 5: S ← ∅
 6: for j = 1 to m do
 7:    da ← [0, ..., 0]    // length(da) = n + 93
 8:    db ← [0, ..., 0]    // length(db) = n + 84
 9:    dc ← [0, ..., 0]    // length(dc) = n + 111
10:    InjectFault(da, db, dc, l_j)
       // Insert a one-bit fault to one of da, db, dc based on the value of l_j
11:    for i = 1 to n do
12:       S_1 ← ∅
13:       dz ← z_i + z_i^(j)
14:       P ← da[i] + da[i + 27] + db[i] + db[i + 15] + dc[i] + dc[i + 45] + dz // (3)
15:       da[i + 93] ← right hand side of (5.1)      // replace △a_{i+24} by da[i + 24], ..
16:       db[i + 84] ← right hand side of (5.2)      // replace △b_{i+6} by db[i + 6], ..
17:       dc[i + 111] ← right hand side of (5.3)     // replace △c_{i+24} by dc[i + 24], ..
18:       repeat
19:          S_2 ← ∅
20:          S_2 ← ExtractUnivariate(P)
21:          P ← Substitute(P, S_2)
22:          S_1 ← S_1 ∪ S_2
23:       until S_2 = ∅
24:       da, db, dc, a, b, c ← Substitute(da, db, dc, a, b, c, S_1)
25:       S ← S ∪ S_1
26:    end for
27: end for
28: return P ∪ S
```

Table 1. Comparison between our Generator and Hojśik-Rudolf Generator for *n* = 800 and *m* Fault Injections.

Generator	m	degree 1	degree 2	degree 3	degree 4
H-R	0	800	2400	0	0
H-R	1	825	2466	35	57
H-R	2	1017	2419	36	57
H-R	3	1258	2298	37	56
H-R	4	2402	498	8	12
our	0	800	2400	0	0
our	1	994	2394	28	27
our	2	1212	2362	64	76
our	3	1619	1990	82	66
our	4	2688	0	0	0

5. Attack Description

Algebraic cryptanalysis is based on solving a multivariate polynomial system that describes a cryptosystem. There are several methods for solving such systems. Computing Gröbner basis is one of the standard techniques for solving multivariate polynomial systems including F4 [7], F5 [8], and MXL_3 [10] algorithms. One of the main problems of all of these techniques is the memory usage when we try to solve large systems even if the systems are sparse. Recently SAT solvers have made a great progress in algebraic cryptanalysis. In [5], Bard et al used a SAT solver combined with the slide attack to break Keeloq block cipher. Using SAT solvers, Eibach et al. [6] attacked Bivium, a scaled version of Trivium. According to our experiments, SAT solvers yielded superior results to Gröbner basis techniques provided by Magma and PolyBoRi [3]. For our implementation of the MXL_3 algorithm, the best variant of the XL algorithm we have, solving such systems is not feasible since our implementation is based on the dense matrix representation, whereas systems considered in this paper are sparse.

The aim of our attack is to recover the inner state of the stream cipher Trivium at $3 = 0$ that leads to recovering the secret key. We assume that the attacker has the following information:

1.) m fault injection positions $(l_1, \ldots, l_m)$, $l_i \leq 288$; $i \in \{1, \ldots, m\}$.

2.) The vector of the output keystream bits before the fault injection, $Z = (z_1, \ldots, z_n)$.

3.) The vectors of the output keystream bits that are obtained after each fault injection, $(Z^{(1)}, \ldots, Z^{(m)})$.

Algorithm 3 explains the main part of this paper. It describes the steps of our attack. The first step has been explained in the previous section. It is important to note that we used only equations of degree ≤ 2 in the attack. In this case, the maximal degree of P is 2.

SAT solvers can deal with a formula in the conjunctive normal form (CNF), a set of clauses, which is a conjunction ($\wedge$) of disjunctions ($\vee$) of some variables or negation of variables. Since we used SAT solving in our attack and the equations from P are represented using the algebraic normal form (ANF), we need to construct the CNF representation of P. We used the ANF-to-CNF converter by Martin Albrecht and Mate Soos [1] to convert our generated system to CNF. This converter uses the method of Bard-Courtois-Jefferson [2] for converting the ANF of polynomial equations to the SAT problem in CNF.

We briefly explain the ANF-to-CNF converter of [2]. In the ANF representation, a Boolean polynomial p is a sum of terms ($t_1 + t_2 + \ldots + t_m$), where each term t_i is a product of variables. For each term t_i of degree ≥ 1, we define a new variable b such that $b_i = t_i$. In terms of corresponding the values of variables from CNF to ANF, we identify each 1 with "True" and 0 with "False". Then we generate CNF clauses that equivalent to $b_i = t_i$. For example, let $p = (x \cdot y + x \cdot z + y \cdot w + x + z + 1)$. We define three new variables b_1, b_2, b_3, where $b_1 = (x \cdot y)$; $b_2 = (x \cdot z)$; $b_3 = (y \cdot w)$. Since the multiplication ($\cdot$) of two variables is simply the conjunction ($\wedge$), then $(b_1 = x \cdot y) \equiv (b_1 \leftrightarrow x \wedge y)$ which is equivalent to the following clauses

$$(\overline{b}_1 \vee x) \wedge (\overline{b}_1 \vee y) \wedge (b_1 \vee \overline{x} \vee \overline{y}) \qquad (6)$$

In the same way, we construct the equivalent clauses of b_2 and b_3. The constant term 1 can be easily represented by the clause $(b \vee \overline{b})$ which is true in all cases. The addition (+) is equivalent to the logic operation (XOR). We can generate the set of clauses of $(b_1 + b_2)$ as

$$(b_1 \vee \overline{b}_2) \wedge (\overline{b}_1 \vee b_2) \wedge (\overline{b}_1 \vee \overline{b}_2) \qquad (7)$$

Using (6) and (7) we can convert a polynomial p from ANF to CNF. The method that we used is based on splitting the equations that contain more than a certain number (called the cutting number) of terms into shorter equations by adding new variables. This is motivated by the fact with this conversion method the number of variables grows exponentially when representing XOR chains. Then we write each equation in the CNF form as explained above. We found that 4 is the best cutting number for our attack.

In steps 3 and 4, we may pass the generated CNF _le to any SAT solver and extract the values of the inner state ($a_1, \ldots, a_{93}, b_1, \ldots, b_{84}, c_1, \ldots, c_{111}$). Then we clock Trivium backwards to recover the secret key K.

6. Experimental Results

We present our results to show how advanced solving techniques can improve the differential fault analysis of the stream cipher Trivium. We used our C++ implementation to generate the Trivium equations and the additional equations that are produced from DFA as in Section 4 and Albrecht's converter [1] as explained in the previous section.

Algorithm 3

Require: m fault injection positions $(l_1, \ldots, l_m)$ and the vectors Z, $Z^{(i)}, 1 \leq i \leq m$
1: $P \leftarrow$ EQgenerator$(l_1, \ldots, l_m, Z, Z^{(1)}, \ldots, Z^{(m)})$
2: CNF$(P) \leftarrow$ Converting P to a satisfiability problem in the CNF form
3: Solution $\leftarrow$ Solve the satisfiability problem CNF(P) by using a SAT solver
4: IS $\leftarrow$ extract the inner state values $(a_1, \ldots, a_{n+93}, b_1, \ldots, b_{n+84}, c_1, \ldots, c_{n+111})$
5: Recover the secret key K from IS
6: **return** IS

We used the SAT solver Minisat2 [11] to recover the inner state of Trivium at $t = 0$. We run all the experiments on an Intel(R) Core(TM)^2 Duo CPU, each CPU is running at 2.8 GHz, and we used only one out of the two cores.

Table 2. Results of using Minisat2.

n	m	degree 1	degree 2	time (sec.)
800	2	1216	2365	0.261
700	2	1088	2080	0.356
600	2	1005	1771	0.414
500	2	890	1437	0.127
450	2	831	1230	1.573
430	2	799	1138	1.936
420	2	769	1117	138.653

Table 2 shows the results of solving DFA Trivium equations when we insert several one-bit faults to random positions in any of the three registers (as explained in Algorithm 2). For each case (n = 800, . . . , 430) in Table 2, we have generated 100 systems and used only the equations of degree ≤ 2. Each system has been solved 100 times by Minisat2. In case of n = 420, we have generated 10 systems and each system solved 10 times by Minisat2. Starting from $n \leq 420$, Minisat2 has taken significantly more time to solve the generated systems. This is due to the fact that the number of low degree equations in the generated system becomes lower. We report the average of these experiments. We have observed that the complexity of solving these systems is based on the number of linear equations generated by the EQgenerator procedure and the heuristic of the SAT solver.

7. Conclusion

We introduced an improvement to the differential fault analysis of Trivium from [9]. By using the SAT solver Minisat2 we could reduce the number of fault injections needed to recover the inner state of the cipher which leads to recovering the secret key. We show that our attack can recover the secret key of Trivium by using only two fault injections and 420 keystream output bits. As a future work, we plan to improve our attack to recover the secret key of Trivium using only one fault injection by applying more advanced conversion techniques and tuning SAT solver parameters.

Acknowledgement

The first author is supported by the BMBF project RESIST. The second author is partially supported by the German Science Foundation (DFG) grant BU 630/22-1. We want to thank the useful comments from Marcel Medwed and anonymous referees of the COSADE workshop on this paper and their valuable suggestions which helped to improve the paper.

References

[1] M. Albrecht and M. Soos. ANF2CNF – Converting ANF to CNF for algebraic attack using SAT solver. http://bitbucket.org/malb/algebraicattacks/src, **(2008)**.

[2] G. V. Bard. Algebraic Cryptanalysis. Springer-Verlag, London, New York, **(2009)**.

[3] M. Brickenstein and A. Dreyer. PolyBoRi: A framework for Gröbner – basis computations with Boolean polynomials. Journal of Symbolic Computation, 44(9):1326–1345, **(2009)**. Effective Methods in Algebraic Geometry.

[4] C. D. Canniere and B. Preneel, Trivium specifications. eSTREAM, ECRYPT Stream Cipher Project, **(2006)**.

[5] N. T. Courtois, G. V. Bard, and D. Wagner. Algebraic and slide attacks on keeloq, In K. Nyberg, editor, Fast

Software Encryption, pages 97-115, Berlin, Heidelberg, **(2008)**, Springer-Verlag.

[6] T. Eibach, E. Pilz, and G. Völkel. Attacking bivium using sat solvers. In Proceedings of the 11th international conference on Theory and applications of satisfiability testing, SAT'08, pages 63{76, Berlin, Heidelberg, **(2008)**, Springer-Verlag.

[7] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F4). Pure and Applied Algebra, 139(1-3):61-88, **(1999)** June.

[8] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In Proceedings of the 2002 international symposium on Symbolic and algebraic computation (ISSAC), pages 75 -83, Lille, France, **(2002)** July, ACM.

[9] M. Hojśik and B. Rudolf. Floating fault analysis of trivium. In Proceedings of the 9th International Conference on Cryptology in India: Progress in Cryptology, INDOCRYPT '08, pages 239-250, Berlin, Heidelberg, **(2008)**, Springer-Verlag.

[10] M. S. E. Mohamed, D. Cabarcas, J. Ding, J. Buchmann, and S. Bulygin. MXL_3: An efficient algorithm for computing gröbner bases of zero-dimensional ideals. In Proceedings of The 12th international Conference on Information Security and Cryptology, (ICISC 2009), Lecture Notes in Computer Science. Springer-Verlag, Berlin, **(2009)** December, Accepted for publication.

[11] N. S. Niklas Een. MinSat 2.0 – one of the best known SAT solvers. http://minisat.se/MiniSat.html, **(2008)**.

[12] M. Robshaw. The estream project. In M. Robshaw and O. Billet, editors, New Stream Cipher Designs, pages 1-6, Berlin, Heidelberg, **(2008)**, Springer-Verlag.