Mutant Differential Fault Analysis of Trivium MDFA

Mohamed Saied Emam Mohamed and Johannes Buchmann

TU Darmstadt, FB Informatik
Hochschulstrasse 10, 64289 Darmstadt, Germany
{mohamed,buchmann}@cdc.informatik.tu-darmstadt.de

Abstract. In this paper we present improvements to the differential fault analysis (DFA) of the stream cipher Trivium proposed in the work of M. Hojsík and B. Rudolf. In particular, we optimize the algebraic representation of obtained DFA information applying the concept of Mutants, which represent low degree equations derived after processing of DFA information. As a result, we are able to minimize the number of fault injections necessary for retrieving the secret key. Therefore, we introduce a new algebraic framework that combines the power of different algebraic techniques for handling additional information received from a physical attack. Using this framework, we are able to recover the secret key by only an one-bit fault injection. In fact, this is the first attack on stream ciphers utilizing minimal amount of DFA information. We study the efficiency of our improved attack by comparing the size of gathered DFA information with previous attacks.

Keywords: Differential fault analysis, algebraic cryptanalysis , Mutants, MDFA, Trivium, eStream

1 Introduction

Stream ciphers are encryption algorithms that encrypt plaintext digits one at a time. Trivium is a hardware-oriented synchronous stream cipher [5]. It was selected in phase three of the eSTREAM project [17]. Due to its simplicity and speed, it provides strong security services 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. Trivium takes an 80-bit key and an 80-bit initial vector IV as inputs in order to generate up to 2^{64} key-stream bits. Trivium operates in two consecutive phases. In the initial phase Trivium iterates 1152 times before it actually starts to produce the key-stream bits of the second phase. The initial iterations are required in order to scramble the original vector to a random inner state.

The concept of differential fault analysis, firstly introduced for stream ciphers in [10], aims at generating additional information about the inner state of the cipher by inspecting and affecting its implementation. The so gathered additional information speeds up the process of key recovery. In this work, we are interested in DFA attacks on the stream cipher Trivium. As for other side-channel attacks, DFA consists of two stages, the online and the offline phase. In the online phase one induces a physical corruption to the cipher by injecting a fault to a random position in the internal state of the cipher. For instance, this is achieved using the technique proposed in [10]. An attacker needs to reset the cipher several times with the same secret information in order to inject more faults. As a consequence, with increasing number of injections one produces more information about the internal state that helps to recover the secret input.

The offline phase, however, is the process of analyzing the information gathered from the online phase. This part of the attack has a great impact on the practicability of the whole attack. To be more precise, the less information an attacker requires from the online part, the less assumptions we impose on the capabilities of the attacker and hence the success of the attack. So, the challenge here is to minimize the number of fault injection rounds.

Previous attacks aim at reducing the number of fault injections needed to find the secret key. The first DFA attack on Trivium was developed by M. Hojsík and B. Rudolf in [12]. The basic idea of this attack is to inject multiple one-bit faults into the inner state of Trivium. In this case, an attacker can generate, in addition to the system of equations that represent the inner state and key-stream bits, some lower degree equations deduced from the information obtained from the online phase. Classical linear algebra tools are used for the analysis of the information produced in the online part. Following this approach, the secret key can be recovered after approximately 43 fault injection rounds. In order to decrease the required fault injections, the authors introduced an improved DFA attack in [11]. This improvement is based on using the so-called floating representation of Trivium instead of the classical one. Applying this method, one needs approximately 3.2 fault injections on average and 800 proper and faulty key-stream bits to recover the inner state of Trivium at a certain time t, which subsequently allows to compute the secret key of the cipher.

Afterwards, in [13] an improvement to the Hojsík and Rudolf attack was presented using SAT-solver techniques instead of classical linear algebra. The idea of this approach consists in translating the additional DFA information together with the algebraic representation of Trivium into the satisfiability problem and then using a SAT technique to solve the underlying problem. This improvement enables the attacker to recover the secret key using two fault injections at a success rate of 100%. In fact, this is the first attack combining differential fault analysis with an advanced algebraic technique.

In order to minimize the number of required fault injections, the mentioned attacks face the following problem: under the assumption that an attacker has a large number of key-stream bits before and after inserting faults, he is capable of using only a limited amount of this information. This is due to the fact that the degree of the additionally generated polynomial equations increases rapidly with increasing number of key-stream rounds. But the high degree equations built during the online phase are useless for an attack. As a consequence, this forces the adversary to carry out multiple fault injections. The challenge is, therefore, to derive more information (low degree equations) after each injection round and thus minimizing the number of required fault injections.

1.1 Our contribution

After a careful analysis of the system of equations generated during the attack and the corresponding DFA information, we propose an improvement to the above two attacks. As a result, we present a more practical algebraic differential fault attack called Mutant DFA (MDFA). Specifically, instead of addressing the gathered DFA information with just one particular algebraic technique as suggested in [11] and [13], we combine several algebraic techniques. In this way we benefit from the advantages of each individual technique. Further, after studying the structure of the corresponding system of equations, we take advantage of Mutants to enrich the collected data after one injection. More specifically, we reduce the amount of DFA information required to solve the corresponding system of equations by enhancing the role of linear Mutants during the system generation stage. As a consequence, we efficiently derive new useful relations that were hidden. In addition to this, we even deduce more information about the system when using advanced algebraic techniques. For instance, we used an adapted version of MutantXL [7], a Gröbner basis technique, for exploring the polynomial ideal generated by the system of equations. This preprocessing step simplifies the corresponding satisfiability problem which subsequently speeds up SAT solver algorithms. Finally, we present a guessing strategy that deals with the high degree equations appearing during the attack in order to produce further Mutants. These improvements enable us to successfully find the key by injecting only a single bit fault into the inner state. Table 1 compares our results with other previous attacks on the selected eStream ciphers [17].

Table 1. Comparison of different attacks w.r.t required number of fault injections

Cipher	Attack	Key size	# Fault injections
MICKEY	DFA [1]	128	$2^{16.7}$
Grain	DFA [2]	80	$2^{8.5}$
Trivium	DFA [11]	80	3.2
Trivium	MDFA	80	1

1.2 Organization

This paper is organized as follows. In Section 2, we provide the relevant background of our attack. In Section 3, we explain how to generate the polynomial equation system that represents both the inner state of Trivium and the gathered

DFA information. A detailed description of the attack and our improved differential fault analysis (MDFA) of Trivium is given in Section 4. Our experimental results are presented in Section 5. Finally, Section 6 concludes this paper.

2 Preliminaries

2.1 Algebraic Cryptanalysis

The discipline of algebraic cryptanalysis uses a range of algebraic tools and techniques to assess the security of cryptosystems, which are essential for trusted communications over open networks. Algebraic cryptanalysis is a young and largely heuristic discipline, and the exact complexity of algebraic attacks is often hard to measure. However, it has proven to be a remarkably successful practical method of attacking cryptosystems, both symmetric and asymmetric, and provides a strong measure for the overall security of a scheme.

The first step in algebraic cryptanalysis is to model a given cipher as a system of polynomial equations. The challenge is then to find a solution to the system, which corresponds to secret information used in the cipher (e.g. plaintext or secret key). In general, finding a solution to a set of polynomial equations is NP-hard. But equations generated by a cipher (from e.g. plaintext/ciphertext pairs) often have structural properties which may be exploited to find a solution significantly faster than a brute force search for the key.

There are many approaches in use today for algebraic cryptanalysis, such as linearization, Gröbner basis, and SAT-solver approaches. These approaches have many tunable parameters. Choosing the right technique and choosing the right parameters has a big impact on the attack performance, the running time and the memory consumption. Each of these techniques has the advantage on the others for certain cases. For example, when the system has many linear connections among large number of terms, a linearization technique performs better than others. One also observes, that Gröbner basis techniques operate more efficiently when the systems are dense or tend to have many solutions. Although having been used for several algebraic attacks, Gröbner basis algorithms yield a large number of new polynomials or, equivalently, huge matrices. This requires long computing time and large memory resources rendering these approaches inapplicable for real-world problems. However, Gröbner bases provide an implicit representation of all possible solutions instead of just a single (random) one. This perfectly complements solvers for the satisfiability problem that we present next.

In [7], Ding et al. present the so-called MutantXL algorithm (a Gröbner basis algorithm) which came with the concept of Mutants, certain low degree polynomials appearing during the matrix enlargement step of the XL algorithm. In this paper we utilize this concept in order to derive new low degree equations in the offline phase of the DFA attack as we will explain later. Involving this particular algorithm also helps to understand the structure of the system of equations.

In the last decade solving the satisfiability problem of Boolean logic (SAT) was heavily researched and SAT-solvers became one of the main approaches of algebraic cryptanalysis. In our setting a system of equations is translated into a set of clauses constituting an equivalent SAT instance, which is then given to a SAT solver (e.g. Mini-SAT2 [15]). Converting a SAT solution into a root of the corresponding algebraic problem is usually straightforward. A SAT problem is to decide whether a given logical formula is satisfiable, i.e. whether we can assign Boolean values to its variables such that the formula holds. A variant is to prove that such an assignment does not exist. We may assume that the formula is in conjunctive normal form (CNF): The formula consists of a set of clauses. Each clause consists of a set of literals that are connected by disjunctions and each literal is either a variable or a negated variable. Moreover, all clauses are assumed to be related by conjunctions. SAT solvers are usually based on the DPLL algorithm. This algorithm uses depth-first backtracking in order to search the space of all possible assignments of the decision variables. Advanced SAT solvers operate with various heuristics allowing for an improved search quality. For instance, they learn new clauses (conflict clauses) from wrong variable assignments in order to guide the further search. Courtois et al. [6] employed the SAT solver MiniSAT2 in conjunction with the slide attack to break 160 rounds of the block cipher Keeloq.

Recently there have been several works combining algebraic attacks with other methods of cryptanalysis. For example, [16] shows how to combine algebraic attack with side-channel attacks, and in [9] DES was attacked by combining algebraic cryptanalysis with differential attacks. In this paper we present a new framework, which integrates the power of several algebraic techniques to optimize differential fault analysis of the Trivium stream cipher. The key idea underlying our proposal, when handling DFA information, is to exploit the advantages of multiple algebraic techniques in order to extract more useful equations such that a solution to the secret key is efficiently obtained with less assumptions imposed on an attacker.

2.2 Differential Fault Analysis (DFA)

Differential fault analysis can be considered as a type of implementation attacks. The DFA attack is divided into two phases, the online and the offline phases. During the online phase of DFA an attacker investigates stream or block ciphers by the ability to insert faults to random places of the inner state of a cipher. There are different types of DFA, in this work we are interested in DFA attacks associated to the scenario, where the attacker is assumed to be capable of injecting a fault to the inner state of a cipher. This requires to permit the attacker to reset the cipher to its initial state using the same key and initial vector. Knowing the plaintext and both the correct and the faulty ciphertext generated after the fault injection, the attacker can deduce new relations that help at solving the underlying problem. In previous attack scenarios, the attacker is allowed to repeat the previous steps several times in order to succeed. Our attack, however, restricts the attacker to inject at most one fault and hence requires less assumptions as compared to previous proposals. The fault position can be determined following the approach provided in [11].

The offline phase, on the other hand, is the process of analysing the information gathered from the online phase. This part of the attack has a great impact on the practicability of the whole attack. More specifically, minimizing the amount of required DFA information entails a more practical attack. As a result, the challenge here is to minimize the number of fault injections.

2.3 Notation

Let $X := \{x_1, \ldots, x_n\}$ be a set of variables and

$$R = \mathbb{F}_2[x_1, \ldots, x_n]/\langle x_1^2 - x_1, ..., x_n^2 - x_n\rangle$$

be the Boolean polynomial ring in X with the terms of R ordered by a certain polynomial ordering. We represent an element of R by its minimal representative polynomial over $\mathbb{F}_2$, where the degree of each term w.r.t any variable is 0 or 1. Let p be a polynomial and T be a set of polynomials in R. We define the head term of $p \in R$, denoted by $\mathrm{HT}(p)$, as the largest term in p according to the order defined on R and the degree of p, denoted by $\deg(p)$, as the degree of $\mathrm{HT}(p)$. A row echelon form $\mathrm{RE}(T)$ is simply a basis for $\mathrm{span}(T)$ with pairwise distinct head terms (see [14] for definition). We define $\mathrm{V}(p)$ and $\mathrm{V}(T)$ as the set of all variables in p and T, respectively.

We define the ideal $I(T)$ of T as the set of all polynomials in R generated by T. A polynomial p in the ideal $I(T)$ is called Mutant if p is a low degree polynomial obtained from the reduction of higher degree ones in I(T). For example, let $p = f + g$ and $\deg(f) = \deg(g) = d$; p is Mutant if $\deg(p) < d$. In this paper, we restrict ourselves with Mutants which have degree ≤ 2.

Finally, we define the difference between two sets S_1 and S_2, denoted by $\mathrm{diff}(S1, S2)$, as the number of elements that exist in S_1 and do not exist in S_2.

3 Algebraic Representation

In this section, we revisit the algebraic description of both Trivium and DFA information introduced in [11]. First, we start with the algebraic representation of the Trivium stream cipher. Trivium builds a huge number of key-stream bits using an 80-bit secret key K and an 80-bit initial vector IV. It consists of three quadratic feedback shift registers X, Y, Z with lengths 93, 84, 111 respectively. These registers represent the 288-bits inner state of Trivium. The initial inner state of Trivium at time $t = 0$ is filled with K and IV as shown in Figure 1. At time t_0 Trivium starts to produce key-stream bits.

In this paper we use the floating representation of Trivium according to [11]. Following this, the three shift registers X, Y, and Z are updated for $t \geq 1$ as follows:

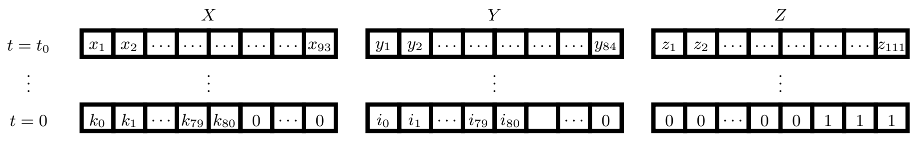
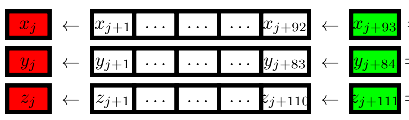


Fig. 1. Internal state of Trivium.


x_j ← x_{j+1} … x_{j+92} ← x_{j+93}
y_j ← y_{j+1} … y_{j+83} ← y_{j+84}
z_j ← z_{j+1} … z_{j+110} ← z_{j+111}

$$x_{j+93} = x_{j+68} + z_{j+65} + z_{j+110} + z_{j+109} \cdot z_{j+108} \tag{1.1}$$

$$y_{j+84} = y_{j+77} + x_{j+65} + x_{j+92} + x_{j+90} \cdot x_{j+91} \tag{1.2}$$

$$z_{j+111} = z_{j+86} + y_{j+68} + y_{j+83} + y_{j+81} \cdot y_{j+82} \tag{1.3}$$

The output key-stream sequence o_j produced by Trivium is generated as follows

$$o_j = x_{j+65} + x_{j+92} + y_{j+68} + y_{j+83} + z_{j+65} + z_{j+110}, \quad j \geq 1. \tag{2}$$

Therefore, let t_0 denote the starting point as explained before. Further, denote by n the number of key-stream bits and let the inner state be represented as depicted in Figure 1. By using the floating representation from (1.1)-(1.3) and (2), Algorithm 1 describes the procedure of generating a system of $4n$ polynomial equations (n linear and $3n$ quadratic). Solving this system requires to find the inner state of Trivium at time $t \geq t_0$ and then by clocking Trivium backwards, we get the secret key k. The best known algebraic attack on a scaled version of Trivium (called Bivium) was developed by Eibach et al. in [8] using SAT solvers and up to date there is no known algebraic attack better than the brute force on Trivium.

Algorithm 1 T($o = (o_1, \ldots, o_n)$)

```
$Sys \leftarrow \emptyset$
for $j = 1$ to $n$ do
    $Sys \leftarrow Sys \cup x_{j+93} + x_{j+68} + z_{j+65} + z_{j+110} + z_{j+109} \cdot z_{j+108} = 0$
    $Sys \leftarrow Sys \cup y_{j+84} + y_{j+77} + x_{j+65} + x_{j+92} + x_{j+90} \cdot x_{j+91} = 0$
    $Sys \leftarrow Sys \cup z_{j+111} + z_{j+86} + y_{j+68} + y_{j+83} + y_{j+81} \cdot y_{j+82} = 0$
    $Sys \leftarrow Sys \cup o_j + x_{j+65} + x_{j+92} + y_{j+68} + y_{j+83} + z_{j+65} + z_{j+110} = 0$
end for
return $Sys$
```

As explained above, we assume that an attacker obtains from the online phase after inserting m fault injections the following DFA information:

- The sequence $(o_j)_{j=1}^{n}$ of the correct key-stream bits.
- The set of sequences $\{(o_j^k)_{j=1}^{n}\}_{k=1}^{m}$ of the faulty key-stream bits for each fault injection $k \in \{1, \ldots, m\}$.

– The set $\{l_k\}_{k=1}^{m}$ corresponds to the indices of the k fault injections carried out on the inner state of Trivium at time t_0.

The second step of the attack is the process of analyzing the gathered information. Intuitively, one has to carry out as many fault injections as required linear equations that are sufficient to recover all bit values of an inner state at time $t \geq t_0$. As in [11], we study the difference between the correct and the faulty inner state at t_0 in order to generate a sequence of additional equations that we add to the algebraic representation of Trivium. In this case, the system of equations is enlarged without increasing the number of unknowns. In the beginning, the bit values of the inner state differences have zeros everywhere, except at the position of injection it has one. Figure 2 illustrates a potential attack, where the injection occurred at a random position of the second register Y.

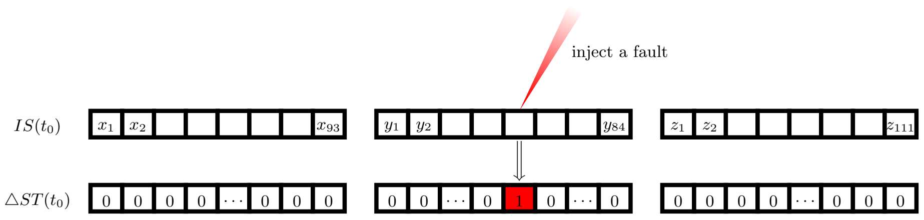


Fig. 2. Fault injection to the internal state of Trivium.

As already explained above, each one-bit fault injection provides a sequence of faulty key-stream bits. Under the assumption that an attacker has also the proper sequence, he uses this information to build the sequence $(\Delta o_j)_{j=1}^{n}$ of key-stream bit differences. Using equation (2), we have for $j \geq 1$

$$\Delta o_j = \Delta x_{j+65} + \Delta x_{j+92} + \Delta y_{j+68} + \Delta y_{j+83} + \Delta z_{j+65} + \Delta z_{j+110}. \tag{3}$$

Moreover, equations (1.1)-(1.3) are used to deduce the sequences of equations $(\Delta x_j)_{j=94}^{n+93}$, $(\Delta y_j)_{j=85}^{n+84}$, and $(\Delta z_j)_{j=112}^{n+111}$ of inner state differences as in (4.1)-(4.3).

$$\Delta x_{j+93} = \Delta x_{j+68} + \Delta z_{j+65} + \Delta z_{j+110} + \Delta(z_{j+109} \cdot z_{j+108}) \tag{4.1}$$
$$\Delta y_{j+84} = \Delta y_{j+77} + \Delta x_{j+65} + \Delta x_{j+92} + \Delta(x_{j+90} \cdot x_{j+91}) \tag{4.2}$$
$$\Delta z_{j+111} = \Delta z_{j+86} + \Delta y_{j+68} + \Delta y_{j+83} + \Delta(y_{j+81} \cdot y_{j+82}) \tag{4.3}$$

4 Generating Mutants

Now we are going to explain our improvements that aim at maximizing the amount of Mutants during the process of generating DFA equations. In previous attacks only the free Mutants obtained from DFA information and the deduced univariate equations were used to simplify the high degree equations. However, our approach also includes all new derived linear Mutants in order to provide more simplifications. This helps us to reduce the degrees of evaluated equations. Indeed, this helps to delay the appearance of high degree relations and enables us to generate further new Mutants.

To be more precise, our generator gets the set of equations T that establishes relations of the inner state and key-stream bits as depicted in (1.1)-(1.3) and (2). Furthermore, the generator is fed with a set of n key-stream bit differences $\{\Delta o_j\}_{j=1}^{n}$, which we use to build additional relations. As indicated in (3), the polynomials represented by the sequence (Δo_j) are linear w.r.t the three inner state bit difference sequences (Δx_j), (Δy_j), and (Δz_j). These sequences as provided in (4.1)-(4.3) have a great impact on the procedure of generating new relations, since the degrees of the constructed polynomial equations rapidly increase. For example, assuming the last term of equation (4.1) $\Delta(z_{j+109} \cdot z_{j+108})$, it is algebraically equivalent to the two quadratic terms $(z_{j+109} \cdot \Delta z_{j+108}) + (\Delta z_{j+109} \cdot z_{j+108})$. Hence, when any of Δz_{j+108} or Δz_{j+109} equals to one, then z_{j+109} or z_{j+108} starts to appear, respectively. By time the degrees of such terms grow into the process of the system generator. Therefore, we give the expansion of these three chains more attention in order to generate many Mutants.

As shown in Figure 2, the sequences (Δx_j), (Δy_j), and (Δz_j) start with constants (zeros everywhere and 1 only at the injection position). As j increased, the value "one" spreads to many positions of the three sequences. Afterwords, the variables express the Trivium inner state bit sequences (x_j), (y_j), and (z_j) are started to appear. In the early steps these linear terms transmit to (Δo_j) equations, which in turn leads to generating new linear Mutants. Thereafter, the relations become more complicated as j is incremented. We gathered all new deduced polynomials in M, the set of generated mutants.

In [11], the authors deal with the problem of building high degree relations in M by simplifying all constructed polynomial equations $H = T \cup M$ using any deduced univariate Mutants. Further, they used Gaussian elimination to evaluate the row echelon of H (see the notation in Section 2), which provides more univariates. In our approach, we use not only univariate Mutants but also we simplify H using new derived linear Mutants. 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 $Z^{(j)} = (z_1^{(j)}, \cdots, z_n^{(j)}), 1 \leq j \leq m$.

Table 2 illustrates the impact of giving Mutants a dominant role for building the system of equations. We compare our generator approach with the approaches used in the previous attacks. For this comparison, we assume an attacker has $n = 800$ proper and faulty key-stream bits. We denote the number of faults by m. We report the number of new derived linear equations (Mutants) and the total number of equations in the generated system. Clearly, the first attack generates only 192 mutants and the second one slightly improves it. Our generator produces 330 Mutants, which in turn improves the attack.

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

```
$Sys \leftarrow \mathrm{T}(Z)$
$x \leftarrow [x_1, \ldots, x_{n+93}]$
$y \leftarrow [y_1, \ldots, y_{n+84}]$
$z \leftarrow [z_1, \ldots, z_{n+111}]$
$S \leftarrow \emptyset$
for $j = 1$ to $m$ do
  $\Delta x \leftarrow [0, \ldots, 0]$   // length($\Delta x$) $= n + 93$
  $\Delta y \leftarrow [0, \ldots, 0]$   // length($\Delta y$) $= n + 84$
  $\Delta z \leftarrow [0, \ldots, 0]$   // length($\Delta z$) $= n + 111$
  InjectFault($\Delta x, \Delta y, \Delta z, l_j$)
  // Insert a one-bit fault to one of $\Delta x, \Delta y, \Delta z$ based on the value of $l_j$
  for $i = 1$ to $n$ do
    $S_1 \leftarrow \emptyset$
    $dz \leftarrow o_i + o_i^{(j)}$
    $Sys \leftarrow \Delta x[i] + \Delta x[i+27] + \Delta y[i] + \Delta y[i+15] + \Delta z[i] + \Delta z[i+45] + dz$ // (3)
    $\Delta x[i+93] \leftarrow$ right hand side of (4.1)
    $\Delta y[i+84] \leftarrow$ right hand side of (4.2)
    $\Delta z[i+111] \leftarrow$ right hand side of (4.3)
    repeat
      $S_2 \leftarrow \emptyset$
      $S_2 \leftarrow$ ExtractMutant($Sys$)
      $Sys \leftarrow$ Substitute($Sys, S_2$)
      $S_1 \leftarrow S_1 \cup S_2$
    until $S_2 = \emptyset$
    $\Delta x, \Delta y, \Delta z, x, y, z \leftarrow$ Substitute($\Delta x, \Delta y, \Delta z, x, y, z, S_1$)
    $S \leftarrow S \cup S_1$
  end for
end for
return $Sys \cup S$
```

Generator	m	#Linears	# mutants
-	0	800	0
H-R [11]	1	992	192
Attack in [13]	1	994	194
our	1	1130	330

Table 2. Compare the size of generated linear mutants and total system

In addition to that, we used a modified version of the MutantXL algorithm, in order to perform an intelligent exploration of the ideal generated by the set of equations H, where H is the set resulting from the process described above. This allows us to construct further additional Mutants. Specifically, we partition the set of equations H into subsets H_i, $i \in \{0, \ldots, 6\}$ based on the distance from the set of Mutants M. In this case, $H_i = \{p \in H | \text{ diff}(\text{V}(p), V(M)) = i\}$, see the notation in Section 2. In other words, H_0 is the set of all polynomials in H that relate only variables occurring in M, $\text{V}(M)$. Thereby, H_1 is the set of all polynomials in H such that only one variable does not appeared in M, and so on. Our smart MutantXL aims to enlarge each subset H_i using elements from H_{i+1}. Without loss of generality, consider the subset H_1. Let p be a polynomial in H_1 and the variable v in $\text{V}(p)$ is the different variable from $\text{V}(M)$, i.e. $v \notin \text{V}(M)$. If p is linear, then we use it to eliminate the variable v from the whole system. In case of p is quadratic, based on the structure of Trivium equations as in (1.1)-(1.3), only the head term $\text{HT}(p)$ is quadratic and the remaining terms are linear. Let v occur in $\text{HT}(p)$ and $\text{HT}(p) = u \cdot v$ where $u \in \text{V}(M)$. If v does not appeared in the linear part of p, then we eliminate it by replacing p with $f = p + u \cdot p$. However, if v occurs in both linear and quadratic parts then we replace p with $f = u \cdot p$. Then p is transmitted from H_1 to H_0. Consequently, after performing the previous reduction procedure, all such polynomials are included in H_0. We perform the same reduction procedure on each subset H_i. Moreover, we use a linear algebra technique, namely Gaussian elimination, in order to derive more mutants. We repeat this procedure until we gathered all possible mutants. Finally, we include all new Mutants to M. Figure 3 illustrates the previous procedure.

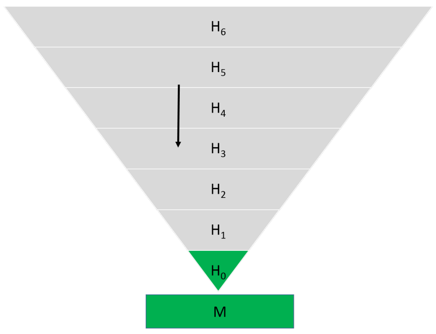


Fig. 3. Reducing polynomials from H_i to H_{i-1}, in order to derive additional mutants.

In order to find the secret information of Trivium, we need to recover sequential 288 inner state bits $(x_j, \ldots, x_{j+92})$, $(y_j, \ldots, y_{j+83})$, and $(z_j, \ldots, z_{j+110})$ at certain $0 < j < N$ using only one-bit fault injection. The above procedure improved the algebraic representation of the system derived from the attack, however as we will show in the next section, we still have to gather more information to be able to solve the system. To be more precise, we need to derive a sufficient number of Mutants from the high degree relations in order to solve the system. It may be argued that the only way to do this is to guess some variables. The challenge here is to generate the required additional Mutants with fewer variables to guess.

We present the following smart guessing strategy for achieving the goal that has been mentioned. As we explained earlier, we are able to gather more mutants from the sequences described the inner state and key-stream bit differences as in (4.1)-(4.3) and (3). We noticed from the constructed chains (Δx_j), (Δy_j), and (Δz_j) for $0 \leq j \leq n$ that, each unsolved variable v constructed in these chains builds a tree for its relations with the consecutive constructed equations. The first level of this tree contains all linear relations with v, the second level all quadratic relations and so on. In order to deduce more hidden relations from those sequences, we select for our guessing approach some of these variables which have the biggest trees. In other words, we chose to guess the most frequent variables in the Δ chains. Table 3 explains the additional Mutants produced after guessing g variables.

g	#L. Mutants
0	330
5	392
10	436
15	489
20	527
25	576
30	620
35	649

Table 3. The size of generated linear Mutants after guessing g variables

5 Experimental Results

In this section, we present our experimental results which establish the performance of our improved DFA attack. Since we used SAT tools to solve the system obtained from the attack as explained in the previous section, we need to build CNF instances corresponding to our optimized algebraic systems of polynomial equations. We consider this as the final step of our attack. Since our system is represented by the algebraic normal form ANF, we used the ANF to CNF converter implemented in SAGE, a mathematical open source tool [18]. It uses two different techniques for converting ANF to CNF. The first one is based on looking at a form of a polynomial itself [3]: representing each monomial in CNF and then representing sums of monomials (polynomials) in CNF. The latter is a bottleneck for SAT, since XOR chains need exponentially large representation in CNF. To overcome this issue, one "cuts" a sum (a XOR chain) into several ones by introducing new variables; each sum is now of moderate size. One then represents each sum separately. This method performs better when the system is slightly dense, we call it the dense technique. The second method is based on considering the truth (value) table of a Boolean polynomial seen as a function (PolyBoRi's CNF converter [4]). We call it the sparse technique. We use a combination of the two methods based on the value of a sparsity parameter called *sp*. So, for example, if we set $sp = 3$, then each polynomial p in the system with $|\,\mathrm{V}(p)| \leq 3$ will be translated to clauses using the sparse technique. Otherwise, the converters use the dense technique. As a result from the previous algebraic procedure, we optimize the representation of the satisfiabilty problem.

We used our C++ implementation to generate the optimized system of equations and the additional set of Mutants resulting from the input DFA information gathered in the online phase of the attack. Further, we use again SAGE and its Boolean polynomial ring to develop the smart version of MutantXL that we used in the attack. Finally, we use the SAT solver Minisat2 [15] to solve the system. The result is obtained as an average over 100 runs under a time limit 3600 seconds (one hour). We run all the experiments on a Sun X4440 server, with four "Quad-Core AMD Opteron$^{\text{TM}}$ Processor 8356" CPUs and 128 GB of RAM. Each CPU is running at 2.3 GHz.

g	time t (sec.)	attack complexity C (sec.)
22	9248.13	$\simeq 2^{35.17}$
23	4224.57	$\simeq 2^{35.04}$
24	939.40	$\simeq 2^{33.87}$
25	108.76	$\simeq 2^{31.76}$
26	76.38	$\simeq 2^{32.26}$
27	56.64	$\simeq 2^{32.82}$
28	42.37	$\simeq 2^{33.40}$
29	36.48	$\simeq 2^{34.19}$
30	22.15	$\simeq 2^{34.47}$
31	4.21	$\simeq 2^{33.07}$
32	2.82	$\simeq 2^{33.49}$
33	1.94	$\simeq 2^{33.96}$
34	1.32	$\simeq 2^{34.39}$
35	1.27	$\simeq 2^{35.35}$

Table 4. Results of our attack using only a one-bit fault injection

Table 4 reports our experiments. The number of guessed variables is denoted by g, the average time (in seconds) of the SAT-solver Minisat2 for solving one instance is denoted by t, and finally C is the total time complexity in seconds. In this scenario, we select g variables of the inner state sequences as explained in the previous section. We studied both the correct and the wrong guessing cases and took the average time. We found that at least 22 variables are needed to guess in order to solve the constructed system and guessing 25 bits gives us the lowest time complexity as explained in Table 4. Consequently, in order to recover the secret key of Trivium we require in (on average) about $2^{31.76}$ seconds using only one core. Since our generator can be used in parallel by dividing the 2^{25} possible cases on N cores and assuming we have a super computer with more than 1000 cores, then one can use our attack and recover the secret information in approximately 42 days.

6 Conclusion

In this paper, we presented an improvement of a differential fault attack (DFA) on the stream cipher Trivium. The main idea of the paper is to combine several algebraic tools to reveal the secret key by injecting only one single fault. First, we enhanced the role of linear Mutants (lower degree relations) during the step of processing the gathered DFA information. Secondly, we modified MutantXL in such a way that it generates additional sparse relations which in turn improves the CNF representation of the constructed system and speeds up the SAT-solver process. Finally, we presented a guessing strategy that deals with high degree relations appearing during the attack. Our attack methodology can be considered as a template for attacking other stream ciphers, such as MICKEY and Grain.

References

1. S. Banik and S. Maitra. A differential fault attack on mickey 2.0. In *CHES*, pages 215–232, 2013.
2. S. Banik, S. Maitra, and S. Sarkar. A differential fault attack on grain-128a using macs. *IACR Cryptology ePrint Archive*, 2012:349, 2012. informal publication.
3. G. Bard. *Algebraic Cryptanalysis.* Springer, 2009.
4. M. Brickenstein and A. Dreyer. Polybori: A framework for Gröbner-basis computations with boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326 – 1345, 2009.
5. C. D. Canniere and B. Preneel. Trivium specifications. *eSTREAM, ECRYPT Stream Cipher Project*, 2006.
6. 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.
7. J. Ding, J. Buchmann, M. S. E. Mohamed, W. S. A. Moahmed, and R.-P. Weinmann. MutantXL. In *Proceedings of the 1st international conference on Symbolic Computation and Cryptography (SCC08)*, pages 16 – 22, Beijing, China, April 2008. LMIB.
8. 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.

9. J.-C. Faugère, L. Perret, and P.-J. Spaenlehauer. Algebraic-Differential Cryptanalysis of DES. In *Western European Workshop on Research in Cryptology – WEWoRC 2009*, pages 1–5, July 2009.
10. J. J. Hoch and A. Shamir. Fault analysis of stream ciphers. In *Chryptographic Hardware and Embedded Systems CHES 2004, Lecture Notes in Computer Science*, pages 240–253. Springer-Verlag, 2004.
11. M. Hojsík 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.
12. M. Hojsk and B. Rudolf. Differential fault analysis of trivium. In K. Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2008.
13. M. S. E. Mohamed, S. Bulygin, and J. Buchmann. Using sat solving to improve differential fault analysis of trivium. In T.-H. Kim, H. Adeli, R. J. Robles, and M. O. Balitanas, editors, *ISA*, volume 200 of *Communications in Computer and Information Science*, pages 62–71. Springer, 2011.
14. M. S. E. Mohamed, D. Cabarcas, J. Ding, J. Buchmann, and S. Bulygin. MXL3: 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, December 2009. Accepted for publication.
15. N. S. Niklas Een. MinSat 2.0 – one of the best known SAT solvers. http://minisat.se/MiniSat.html, 2008.
16. M. Renauld and F.-X. Standaert. Algebraic side-channel attacks. In *Inscrypt*, pages 393–410, 2009.
17. M. Robshaw. The estream project. In M. Robshaw and O. Billet, editors, *New Stream Cipher Designs*, pages 1–6, Berlin, Heidelberg, 2008. Springer-Verlag.
18. W. Stein et al. *Sage Mathematics Software (Version x.y.z).* The Sage Development Team, YYYY. `http://www.sagemath.org`.