Advanced Algebraic Attack on Trivium

October 29, 2014

Frank-M. Quedenfeld[1] and Christopher Wolf[1]

[1]Ruhr University Bochum, Germany

frank.quedenfeld@ruhr-uni-bochum.de

Abstract

This paper presents an algebraic attack against Trivium that breaks 625 rounds using only 4096 bits of output in an overall time complexity of $2^{42.2}$ Trivium computations. While other attacks can do better in terms of rounds (799), this is a practical attack with a very low data usage (down from 2^{40} output bits) and low computation time (down from 2^{62}).

From another angle, our attack can be seen as a proof of concept, how far algebraic attacks can be pushed when several known techniques are combined into one implementation. All attacks have been fully implemented and tested; our figures are therefore not the result of any potentially error-prone extrapolation.

Keywords: Trivium, algebraic modelling, similar variables, ElimLin, sparse multivariate algebra, equation solving over $\mathbb{F}_2$

1 Introduction

Algebraic attacks against symmetric ciphers are more than a decade old. Actually, they can be traced back to Claude Shannon [Sha49]. While being promising, they did not deliver exactly what promised and were mostly dismissed in cryptanalytic literature.

Recently, the so-called "ElimLin" algorithm was used to attack several ciphers, in particular CTC2, LBlock and MIBS. According to [CSSV12] from FSE 2012, only 6 rounds can be broken for CTC2. This attack requires up to 180h on a standard PC and requires 210 guessed bits and 64 chosen cipher texts (CC). Guessing 220 bits and 16 CP brings the attack down to 3h. For LBlock, the paper reports 8 rounds (out of 32) for 32 guessed bits (out of 80) for 6 known plain texts (KP). For MIBS (32 rounds), the paper reports a break for 3 to 5 rounds with 0/16/20 guessed bits, respectively. An initial implementation of the ElimLin algorithm was employed on DES in [CB07]. Here, plain ElimLin could break 5 rounds of DES with 3 KP and 23 guessed bits; using a SAT solver, this number can be increased to 6 rounds for an unspecified number of KP (most likely 1) and 20 key bits fixed. In 1.2 we discuss this more detailed and referenced.

In this article, we show that ElimLin can be greatly improved when employed together with other well known techniques from algebraic cryptanalysis such as eXtended Linearization or proper monomial ordering. In particular, we use the Trivium stream cipher as test bed for algebraic attacks, mainly due to its simple algebraic structure and its good scalability: Full Trivium has 1152 rounds, so we could see the effect of adding some component to our equation solver well, cf. Sec. 3 for all building blocks. In addition, we restricted ourselves to attacks that can be fully implemented on a nowadays computer. Our implementation was able to break round-reduced Trivium with 625 rounds. In particular, our data complexity is far better than for non-algebraic attacks. Non-algebraic attacks need at least 2^{40} to 2^{45} output bits of Trivium with $767 - 799$ as we present in 1.2. We are able to bring this down to 2^{11} or 2^{12}. Further non-algebraic attacks can use up to 2^{60} to 2^{72} Trivium computations which is not feasible on modern computers. This is because they are guessing a huge amount of variables.

In particular, we show that algebraic attacks become specifically efficient against Trivium if we do not use "much" output for one instance but few output bits for many instances. However, this new type of attack only works if we have access to a sparse equation solver over $\mathbb{F}_2$ that can deal with many variables and also many equations ($\approx 10^6$ in both cases). This equation solver is the second major contribution of this paper. To the best of our knowledge, such a solver does not exist in the open literature yet.

1.1 Organization

We start with a review of existing work in the area of algebraic cryptanalysis and specifically cryptanalysis of Trivium in Sec. 1.2. In addition, we discuss several ways to solve systems over $\mathbb{F}_2$. This is followed by a discussion of Trivium and the idea of *"similar variables"* in Sec. 2. The overall solver and the tweaks we need to deal with a full representation of Trivium are given in Sec. 3. This is followed by practical experiments on round reduced Trivium in Sec. 4. The paper concludes with some remarks, open questions and directions for further research in Sec. 5.

1.2 Related Work

Before going into details about our attack, we review related work in algebraic cryptanalysis, cryptanalysis of Trivium and solving systems of equations over $\mathbb{F}_2$.

Algebraic cryptanalysis. Algebraic cryptanalysis works on a simple assumption: We are able to express any cryptographic primitive in simple non-linear equations over a finite field (*e.g.* $\mathbb{F}_2$ or $\mathbb{F}_{256}$), cf. [BC03] for an overview on some ciphers. This part of the attack is called *"modelling"*. If we now use this description and solve the overall system for a given output (stream ciphers) or a given plaintext / ciphertext pair (block cipher) we obtain the secret key.

Most prominently, the AES has been investigated under algebraic attacks [MR02, CP02]. While initial research concluded that the AES may be vulnerable against algebraic attacks, this was shown to be unlikely in later research [LK07]. In any case, there are two different models of the AES: the first uses a modelling over $\mathbb{F}_2$, the second over $\mathbb{F}_{256}$, see *e.g.* [CMR05]. Apart from this, both are equivalent.

However, for stream ciphers, algebraic attacks [AK03, CM03] seem to work fine, as for some public key systems [FJ03, FOPT10] and other primitives [SKPI07]. We want to note that Trivium has escaped all efforts to be broken by purely algebraic methods.

Attacks on Trivium. We briefly sketch some of the most important attacks against Trivium. We want to stress that Trivium is still secure—despite its simple and elegant design; and the combined effort of the cryptanalytic community up to now.

At first we note that the described cube attacks against Trivium using the same attack scenario as we do. Namely this is the chosen IV scenario. Therefore our comparison to them is valid.

The attacks from [DS09, FV13] are both cube attacks. They recover the full key of a 799 round-reduced variant of Trivium in 2^{62} computations guessing 62 variables. The main problem here is their way of *finding* new cubes. Until now, all cube attacks are equivalent to the statement: *"We have found cubes up to round X with some clever algorithm. Therefore we can break Trivium up to this round."* This makes it difficult to see how they would evolve, say, for Trivium up to round 1152. Nevertheless, more attacks on Trivium are known so far. The articles [KMNP11, ADMS09, Sta10] report some distinguishers attacks based on cube testers. The best covers up to 961 rounds but only works in a reduced key space of 2^{26} keys (out of 2^{80}). It requires 2^{25} Trivium computations.

Other attacks are not that succesful. For instance [KHK06] describes a fully linear attack on Trivium breaking a 288 round-reduced variant with a likelihood of 2^{-72}.

There exist further variants of Trivium using two instead of three shift registers [CP08]. They are called Bivium-A and Bivium-B and were mainly used to demonstrate specific attacks that do not work against full Trivium. In particular the purely algebraic attacks from [SFP08, T+13, SR12, Rad06] make use of these reduced variants to highlight their method. All in all, they successfully break them (for the full number of rounds). They fail for Trivium though, even in its round-reduced version. Taking the insights from these papers into account, the main reason is that they use only *one* instance of Trivium and plenty of output. As we see in the following, this is not the best strategy when dealing with Trivium in an algebraic model. In our work we use *many* instances of Trivium, but only very little output per instance.

Solving systems over $\mathbb{F}_2$. As pointed out above, any algebraic cryptanalysis has two steps: The first is the algebraic modelling, the second is solving the corresponding system of equations. For simplicity, we assume that all equations are at most quadratic over $\mathbb{F}_2$. When we have an equation of total degree greater or equal 3 we can introduce new intermediate variables to reduce the degree of the equation. Further more all our algorithms work on systems of equations with higher degree. We will explain how to adapt our algorithms when we introduce them.

Basically, the most promising algorithms come from the F-family of Gröbner basis algorithms [Fau02b, Fau02a], see [Alb10] for an overview. They have been successfully applied in the case of public key schemes [FJ03, FOPT10], but also stream ciphers [FA03]. The main disadvantage is the high memory consumption. Although there are some counter examples for special cases, the running time of Gröbner basis algorithms is inherently exponential in the number of variables. Even worse, the memory consumption increases with $O(n^r)$ for n the number of variables and r the *degree of regularity*. In particular the latter makes Gröbner bases unusable for our purpose as we have $r \geq 2$ and $n \approx 2^{20}$. Still, we want to note a special and useful property of Gröbner basis algorithms that can also be found in our solver: If the system is unsolvable, the running time becomes significantly lower. This has been used in the so-called "hybrid strategy" to speed up the overall computation [BFP09] over $\mathbb{F}_2$. We will also exploit this, cf. Sec. 3.4.

Another line of algorithms comes from the X-family of so called "XL—eXtended Linearization" [CKPS00]. Here, the main operation is multiplying all known equations with all known variables. While these new equations are trivially true, some of them are linearly independent and can hence be used in a so-called *Macaulay matrix* to reduce the overall problem to linear algebra over $\mathbb{F}_2$. In a Macaulay matrix, the rows represent polynomials while the columns represent monomials, cf. [Alb10] for an overview of the idea. Although it has been shown that techniques from the XL-family are strictly less efficient than from the F-family [YC04b, YC04a, AFI+04, Die04], XL does have its merits as it is far easier to understand and also easier to adapt to different settings. Moreover, we know that XL has a *similar* performance in practice [TW12]. Hence we have used a specialized version of XL in our solver to improve its efficiency, cf. Sec. 3.

Last but not least, there is the `ElimLin` family [CB07, CSSV12] where linear equations are used to eliminate variables. After that, the system is simplified with linear algebra techniques, cf. Sec. 3.1. Although the overall technique seems trivial when compared to the ones discussed above, it does have its merits. In particular, it can handle large (sparse) systems and it is so simple that it can easily be tweaked for specific purposes. While this is possible—to some extent—for the XL-family, it is much more difficult to envision for the F-family described above. We have hence used ElimLin as the core for our solver, cf. Sec. 3.1. However, we want to stress that *plain* ElimLin without further modifications is not efficient enough to deal with systems that arise from the modelling of Trivium.

1.3 Our contributions

Our first contribution are techniques to model many instances of Trivium as an quadratic equation system. We also introduce strategies to handle the large number of variables within this model. The modelling techniques and strategies can be applied to any symmetric cipher since the upcoming system of equation is structured according to the update function of the cipher.

The second contribution of this paper is a solver which is able to solve structured quadratic equation systems. Based on ElimLin and eXtended Linearization we introduce a monomial order to have more control in the ElimLin-Step and change XL so that it preserves the monomial structure of the system. Furthermore we introduce a method that is able to make computations with higher algebraic degree while the result of this computations is still quadratic.

With the above mentioned techniques we settle an attack on a round reduced variant of Trivium with $R = 625$ rounds in $2^{42.2}$ time and 2^{12} data complexity on an average computer.

2 Trivium

The main point of our attack is an algebraic system of equations over $\mathbb{F}_2$. As soon as we have it, we will solve it with a special purpose solver, cf. Sec. 3. Trivium generates up to 2^{64} keystream bits

from an 80 bit IV and an 80 bit key. The cipher consists of an initialisation or "clocking" phase of R rounds and a keystream generation phase. For $R = 1152$ we obtain *full*, otherwise *reduced* Trivium, denoted by Trivium-R. There are several ways to describe Trivium—below we use the most compact one with three quadratic, recursive equations for the state bits and one linear equation to generate the output. The two only operations in Trivium are addition and multiplication over $\mathbb{F}_2$ as this can be implemented extremely efficient in hardware via `XOR, AND` gates.

2.1 Definition and direct considerations

Consider three shift registers $A := (a_i, \ldots, a_{i-92}), B := (b_i, \ldots, b_{i-83})$ and $C := (c_i, \ldots, c_{i-110})$. They are called the *state* of Trivium. The state is initialized with $A = (k_0, \ldots, k_{79}, 0, \ldots, 0), B = (v_0, \ldots, v_{79}, 0, \ldots, 0)$ and $C = (0, \ldots, 0, 1, 1, 1)$. Here $(k_0, \ldots, k_{79})$ is the *key* and $(v_0, \ldots, v_{79})$ is the *initialization vector (IV)* of Trivium. Recovering the vector A is the prime aim of attackers. Note that the vectors B, C are actually known to an attacker. We make the additional assumption that an attacker has control over the IV used within the cipher and obtain a stream of output bits for a fixed key and any choice of initialization vector (IV). This is in line, *e.g.* with cube attacks.
The state is updated using to the following three *state update functions*:

$$\begin{aligned} b_i &:= a_{i-65} + a_{i-92} + a_{i-90}a_{i-91} + b_{i-77}\,, \\ c_i &:= b_{i-68} + b_{i-83} + b_{i-81}b_{i-82} + c_{i-86}\,, \\ a_i &:= c_{i-65} + c_{i-110} + c_{i-108}c_{i-109} + a_{i-68}\,. \end{aligned}$$

After a clocking phase of R rounds, we additionally produce n_o bits of output z_i for $i = (R+1)\ldots(R+n_o)$. The corresponding system is now called *"full"*. The z_i are defined by the function

$$z_i := c_{i-65} + c_{i-110} + a_{i-65} + a_{i-92} + b_{i-68} + b_{i-83}\,.$$

To launch our attack, we use several Trivium instances that share the same key but different values for the IV. We see in Sec. 4 that this will lead to successful attacks for round reduced Trivium.

Obviously, we need approx. $3RT + n_oT$ intermediate variables, respectively, if we want to represent T instances of Trivium-R with n_o output bits each. Before discussing strategies for solving such rather large systems, we start with an observation on Trivium.

2.2 Similar variables

As already mentioned, previous algebraic attacks such as [SFP08, T+13, SR12] are based on the algebraic representation of Trivium given in Section 2.1. Hence, we would expect similar results. However, we will not consider only *one* instance of Trivium but several (thousand). Consequently, the relation between these instances becomes important for the overall success of our attack.

Let $I \subset V$ be a subset of all IV variables V. We call I the *master cube* of the attack. In addition, we consider the first n_o output bits of Trivium instances that are all defined by the same key and all vectors that lie in the master cube. All other IV variables are set to zero.

We set up all Trivium instances with symbolic key variables $k_0, \ldots, k_{79}$. Denote the current Trivium instance by $t \in \mathbb{N}$. We initialize these instances for a given number of rounds R and introduce three new variables for very round i for the entries $a_{t,i}, b_{t,i}$ and $c_{t,i}$ in the three registers A_t, B_t and C_t. This produces a quadratic system with a large amount of variables and monomials.

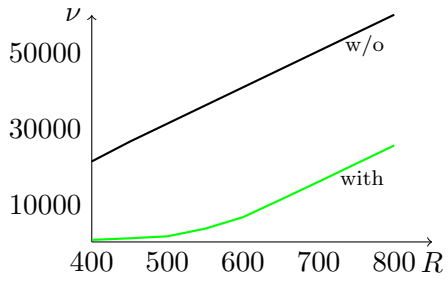
Now we take a more general point of view and introduce similar variables for generalized systems of equations. In particular, we denote all intermediate variables by $y_0, y_1, \ldots$ and we write $\mathbb{K}[x_1, \ldots, x_n]$ short for $\mathbb{K}[x_1, \ldots, x_n]/\langle x_1^2 + x_1, \ldots, x_n^2 + x_n\rangle$

Definition 1. *Let* $\mathbb{R} = \mathbb{F}_2[k_0, \ldots, k_{79}, y_0, y_1, \ldots] =: \mathbb{F}_2[K, Y]$ *be the Boolean polynomial ring in the key variables* K *and all intermediate variables* Y*.*

We call the two intermediate variables y_i *and* y_j *similar iff* $y_i + y_j = p(K, Y\backslash\{y_i, y_j\})$ *where* $p(K, Y\backslash\{y_i, y_j\})$ *is a polynomial of degree* $\deg(p) \leq 1$*.*

Furthermore if we have the set F of polynomial equations in $\mathbb{R}$ introducing the intermediate variables, the so-called set of system equations, we can generalize the definition above as follows:

Figure 1: Number of variables for $T = 32$ instances of Trivium and $n_o = 66$ output bits; Number of Rounds R against number of variables ν with and without similar variables



Definition 2. *Let* $\mathbb{R} = \mathbb{F}_2[k_0, \ldots, k_{79}, y_0, y_1, \ldots] =: \mathbb{F}_2[K, Y]$ *be the Boolean polynomial ring in the key variables* K *and all intermediate variables* Y*. Denote by* F *the* system equations *which defines the intermediate variables already introduced.*

Denote by $Y_k \subset Y, Y_k \neq \emptyset$ *for* $1 \leq k < |Y|$ *the sets of already introduced intermediate variables up to step* k*. We call the intermediate variable* y_i similar *to the set* F *iff there exist a non-empty set* Y_k *such that* $y_i + \sum_{y \in Y_k} y = p(K, (Y \backslash Y_k) \backslash \{y_i\})$ *where* $p(K, (Y \backslash Y_k) \backslash \{y_i\})$ *is a polynomial of degree* $\deg(p) \leq 1$*.*

With similar variables we linearize the system. Instead of using a new intermediate variables for quadratic monomials already introduced we use a linear combination of already introduced intermediate variables.

In generic systems, similar variables could be not of much use. However, if all equations stem from one algebraic model for one given cipher, we are likely to find many similarities. The following example illustrates how we work with similar variables in the case of Trivium.

Example 1. *Consider the equations defining the first two intermediate variables within the Trivium stream cipher for instances* $0 \leq i < T$*:*

$$\begin{aligned} b_{i,12} &= k_{78}k_{79} + k_{53} \\ b_{i,13} &= k_{77}k_{78} + k_{79} + k_{52} \end{aligned}$$

in the system of equations F*. Hence, we already have similar variables for* $b_{j,12} = b_{0,12}$ *and* $b_{j,13} = b_{0,13}$ *for* $0 < j < T$*. Assume at some stage of the computation we have*

$$y' = k_{79}k_{78} + k_{78}k_{77} + k_5 + k_{61}\,.$$

We have $y' = b_{0,12} + b_{0,13} + k_{53} + k_{79} + k_{52} + k_5 + k_{61}$*. So we can eliminate* y' *as a new variable and can go on computing with* $b_{0,12}, b_{0,13}$*. This does not only save us one variable but we also have replaced a quadratic equation by a (potentially more useful) linear one.*

The nice bit is that these variables carry through: As soon as we have identified that all variables $b_{i,12}$ for $i = 0..T-1$ will always have the same value, we can replace *all* of them in all polynomials by only one, say $b_{0,12}$. Obviously, the same is true for $b_{i,13}$ but also many other ones. And as soon as we have replaced $b_{i,12}b_{i,13}$ by $b_{0,12}b_{0,13}$, we obtain even more similar variables. Consequently, we will obtain a substantially smaller number of monomials than before. Note that there are different ways to talk about similar variables. In any case, we need a solver that can first identify them and second make use of them by replacing all linear relations within a given system.

While the above definition captures any behaviour for any system of equations, we see that it applies very well to Trivium, see Figure 1 for some experimental results on Trivium-R. Here, we have generated $T = 32$ instances of Trivium with $n_o = 66$ output bits. On the $x-$axis we see the number of initialization rounds; on the $y-$axis we have the total number of variables in use. As we can see the number of intermediate variables has greatly decreased; even for only 32 instances of Trivium. For $R = 600$ rounds we also produce $66 \cdot 32 = 2112$ output equations. The model in [Rad06] can just handle one instance of Trivium. So it would need $288 + 3 \cdot 2112 = 6624$ variables to produce that amount of output equations. When using more instances of Trivium we get even more efficient.

These systems are still out of reach for nowadays Gröbner basis implementations like PolyBoRi [BD09]. The number of variables is simply too high. With PolyBoRi and our model we are not able to break more than Trivium-420.

We have hence designed a solver which can handle such large numbers of equations and variables and will describe it in the following section.

3 Solving the system

Before going into details for the experiments, we describe our strategy to solve rather sparse systems over $\mathbb{F}_2$ arising from the above representation of Trivium. We have based our solver on multivariate *quadratic* polynomials over $\mathbb{F}_2$ as this is generally enough to capture full Trivium. Specifically, our goal was to develop a *working implementation* than can handle around 10^6 variables and 10^6 equations, respectively, over $\mathbb{F}_2$. To the best of our knowledge, software with such special properties is not available at the moment. Our solver is organized around a specialized C++-core that natively handles quadratic polynomials over $\mathbb{F}_2$ and also the `ElimLin` strategy. In addition, we have used several other building blocks which we describe in the remainder of the section. We report experimental results in Sec. 4.

3.1 Main core

ElimLin or *Elimination Linear* has been investigated in [CSSV12]. We generalize it to the case of quadratic equations and to a monomial ordering, so the algorithm becomes

1. First we generate the Macaulay matrix for the system according to some monomial ordering τ.
2. Echelonize the matrix according to τ. This naturally splits up the system into linear equations L and quadratic equations Q.
3. For each element $p \in L$, use the leading term $\mathrm{LT}(p)$. If there is at least one equation in Q that also contains the variable $\mathrm{LT}(p)$, eliminate $\mathrm{LT}(p)$ in Q.
4. If we substitute at least one variable in Q, go back to step 2.

We want to stress that ElimLin preserves the overall degree of our system Q. In addition, it automatically detects all similar variables (see def. 1). Moreover ElimLin is able to deal with rather large but sparse systems of equations.

The original ElimLin algorithm did not have any ordering but used heuristics to determine which variable to eliminate in the non-linear part of the overall system. We found this approach fine for small systems but difficult to use for larger ones: The likelihood to fall into local optima was simply too high—even with advanced heuristics. Determining the correct order proved to be challenging and required careful experiments. Hence, we used the *degree reverse lexical (degrevlex)* ordering. Note that this also works well in case of Gröbner basis algorithms. In the case of Trivium, we take the key variables first and sort the intermediate variables ascending according to rounds and instances of Trivium. We want to stress that the ordering is crucial in our analysis. Like in Gröbner techniques the results differ significantly depending on the ordering.

Sparse polynomial core. The core of our algorithm is substitution of variables from linear equations and echelonization. While the first requires polynomials, the second needs linear matrices. In particular Gröbner basis algorithms would construct a so-called *Macaulay matrix* and go back and forth between a matrix and a polynomial representation, see [Alb10] for an overview. In our implementation, we used the polynomials over $\mathbb{F}_2$ directly but also implemented matrix-like operations (*e.g.* row addition) directly for polynomials. To this aim, each polynomial is stored as a (sorted) list of monomials rather than sparse vectors over $\mathbb{F}_2$. To make computations fast, we also keep a dictionary of lead terms, monomials in use by each polynomial and also a list of variables in use per polynomial. This way, addition of two polynomials with the same lead term and elimination of variables does not

depend on the overall number of polynomials anymore. For speed, this part of the code is written in C++ (approx. 2500 lines).

M4RI. While the sparse strategy from above turned out to be efficient for sparse matrices, it fails if the matrices become increasingly dense. Note that this is inevitable when solving such a system: In all experiments, we had a degeneration from sparse to dense shortly before solving the overall system. To remedy this, we incorporated the fastest known, open source linear algebra package for matrices over $\mathbb{F}_2$, namely the Method of the 4 Russians Implementation (M4RI) [AAB+]. Experimentally, we have found that matrices with less than $\approx 1/1000$ non-zero coefficients in the corresponding matrix over $\mathbb{F}_2$ should be handled by our *sparse* strategy described above and by M4RI otherwise.

3.2 Further building blocks

While the core is already pretty efficient, it can be made more suitable for solving systems of equations coming from Trivium-instances when coupled with the techniques below. Note that all these techniques except `SL` have been implemented in Sage [S+14] as they are neither time nor memory critical. The full code (including test cases and comments) consists of ≈ 6500 lines.

Sparse Linearization. ElimLin cannot conclude $a * b + a = 1 \Rightarrow a = 1, b = 0$ for some variables $a, b \in \mathbb{F}_2$. Therefore we use a variant of Extended Linearization that we call *Sparse Linearization*. It has been used in the past to attack different cryptographic systems [CP02]. In addition, different variations of the basic algorithm have been described. We give a toy example here: Let $p(x)$ be a multivariate polynomial and v some variable. In a nutshell, we use the simple observation that the second equation follows directly from the first one; independent of the concrete choice of p, v:

$$p(x) = 0 \Rightarrow v \cdot p(x) = 0\,.$$

While the second is only a redundant version of the first, it can be used to produce linearly independent rows in the Macaulay matrix. This observation was used in a cryptanalysis of the *HFE—Hidden Field Equations* crypto system [KS99]. For our purpose, the following specialization proved useful: Let p be a multivariate quadratic polynomial over $\mathbb{F}_2$. Let Π be the list of all monomials in the current system. Then consider all variables in quadratic monomials $Var_2(p)$ in p and compute $\forall v \in Var_2(p)$: vp. If vp contains any monomial *not* already present in Π, delete it. Otherwise, add vp as new row to our system. Note that this implies $\deg(vp) = 2$ and hence $v \in Var_2(p)$. This way, we will not add new monomials to the overall system (*i.e.* columns to the Macaulay matrix) but potentially new linearly independent rows. So all in all, we add between 0 and 2 new polynomials for each given polynomial p because we can only satisfy the conditions above with the variables within the quadratic monomials of p. As it preserves sparsity, we call this technique *sparse linearization*.

Sparse Target Linearization or *STL* is a generalization of the SL technique from above. Instead of limiting ourselves to degree 2 polynomials, we work with polynomials up to some degree $d > 2$. In our calculations we are using a degree up to $d = 4$, which has proven optimal for equations coming from Trivium. At the end of the STL step, we "harvest" all degree 2 equations and feed them into our original system. In the language of Gröbner basis algorithms, we are working with a degree of regularity higher than 2 here.

In a nutshell, the algorithm generates all polynomials that do not introduce *"too many"* new monomials. This is done by selecting monomials of degree three and four that we aim to *eliminate* in the overall system and then compute the corresponding polynomials. While this notion is quite loose, we can make it more explicit by considering different cases for the polynomials we are dealing with:

- p Single quadratic polynomial p: In this case, we add all polynomials vp with $v \in \mathrm{Var}(p)$. This may introduce new monomials to the overall system. This is Extended Linearization.
- p, q Consider two quadratic polynomials p, q that share one variable v in a quadratic monomial $\mu \in p$ and $\nu \in q$. In this case we add the polynomials $(\mu/v)q$, $(\nu/v)p$ and $(\mu/v)q + (\nu/v)p$ to our system. This could lead to cubic polynomials in our system.

p, q' Consider two quadratic polynomials p, q. Compute all possible products between one polynomial and the monomials of the other polynomial. If we can eliminate the upcoming monomials of degree four by adding the new polynomials we add them to the system. Let μ be a monomial of p and the polynomial q. If $\mu \cdot q + \nu p$ *does not* introduce a new monomial of degree four to the overall system, add it to our system. Note that the monomial $\mu\nu$ is not present in $\nu p + \mu q$ and can hence be eliminated from the overall system. We calculate a monomial of degree four here but we just use it in an intermediate calculation.

p, μ Consider some quadratic polynomial p and a monomial μ such that μp does not introduce new monomials to the overall system having now degree three. Then add μp to the overall system. Note that this monomial may come from some other polynomial q'.

Note that we calculate syzygies with a degree of regularity of three and four respectively in the second and third step.

Moreover, we mainly add new polynomials that will not introduce new monomials to our overall system. Our hope is that this will actually lead to more degree 2 polynomials. In addition, we mainly keep the sparsity of the overall system. After STL is applied to the overall system of quadratic equations, we "harvest" all newly generated degree 2 equations and apply all other solving steps. For random systems of equations, the STL strategy is doomed to fail due to the lack of useful quadratic equations afterwards. Our experiments show that it works fine for structured systems like Trivium, cf. Sec. 4.

Evolutionary Strategy. While STL works fine, it has a slightly too narrow view on the system of equations: It blindly creates new polynomials without keeping the overall goal in mind. That is: We want to obtain (quadratic) equations that contains both key variables (x_i) and output bits (z_j). More generally: these equations should cover as many rounds of Trivium as possible. That means if we have an equation involving variables a_{r_x} and a_{r_y} where $r_x < r$ and $r_y > R$ with $r < R$ and a_s is a intermediate variable introduced in round s we want to maximize $r_y - r_x$.

Unfortunately, it proved difficult to implement into the STL algorithm.

Hence, we have formulated this as a goal for an evolutionary strategy (ES) which is used to figure out the useful equations for the STL step: The more rounds are covered, the higher the score for the corresponding polynomial. And the higher the chance that it can replicate within the evolutionary algorithm. All equations of degree 4 or lower are then harvested from the ES and fed into the STL step. Usually, cryptanalysis does not allow the successful application of genetic or evolutionary strategies. In this case, however, the problem seems to have enough degrees of freedom and the optimization goal can be formulated clearly enough so it empirically works. *Note:* This strategy does *not* work when using degree 2 only. It requires degree 3 or higher. Otherwise, the ES does not have enough freedom to find new equations.

Guessing. In most cryptographic attacks, we start with some (deterministic) strategy to single out the promising keys and then bridge the gap for the full attack with brute force. We use the same strategy here by guessing the value of some variables and feed them into the overall solver before starting key recovery. To maximize the effect of the guessing step, we use the variables that occur most often in all polynomials system-wide. As expected, other selection strategies did not work that well for Trivium. More details on this idea are given in Sec. 3.4.

3.3 Big picture

Online- and Offline-Phase. Our solving algorithm is divided into an online and an offline-phase (precomputation). The main goal is to use trade-offs between the different algorithms. For example, the STL-step is quite powerful, but also quite slow. Hence, we only use it once and for all for a given, general system of equations (offline-phase). Once this system has been simplified, we insert all known output-bits, guessed variables and then recover the key. In detail we have

Offline: Generating the raw equations, ElimLin, SL, ES, STL and determine the variables which will be guessed later.

Online: Add guessed variables, output-bits, ElimLin and SL.

As mentioned above, the solver contains approx. 2500 lines of C++-code and approx. 6500 lines of Sage-code. Sage is a CAS described in [S+14]. It was used on an AMD-Opteron-6276@2.3GHz with 256 nodes and 1 TB of RAM. Each node had access to at most 256 GB of RAM at a time. After the system has been simplified, the online-phase runs on a normal PC with 16 GB RAM.

3.4 Scaling the attack

In all algebraic attacks, a big question is scalability and also comparability to the underlying cipher. To tackle the scalability we guess some variables. This way, we can easily scale our attack and also control its running time. We give more details below. For the second, we express both our attack and a brute force scenario in *seconds*. While other bases for comparison are possible such as time-area-product (mostly for hardware implementations), skill and resources of an attacker (*e.g.* see [Ecr12]) or time-memory-product, we stick to time as the only factor for its simplicity. In particular, it is difficult to assign a sensible number for "memory" to a hardware implementation. Moreover, "memory" does not really speed up our attack but is a pure necessity: If we do not have enough memory, the attack does not work at all. Up to now, there are no clever ways to trade time for memory (or vice versa) using the solver described here.

All in all, we use a throughput oriented hardware implementation of Trivium (Table 2, Trivium64 in [GB08]) as basis for our comparison [GB08]. To the best of our knowledge, this is the fastest, throughput oriented implementation of Trivium. The authors report a throughput of $B = 22.2996 \times 10^9$ bps. To implement Trivium-R, we assume that we need to clock this implementation a total of $(R+2)$ times on average: the first R clocks for the preclocking phase and the other 2 clocks to derive 2 output bits on average. The last number is justified as follows: When the attacker learns the first output bit, she can terminate the brute force attack for around $1/2$ of all keys. This goes on for all further bits. With the second output bit she can terminate the second half of all keys, with the third another half and so on. So we need on average two output bits to make our decision for some $\nu = 80 \dots 100$. So we have $\frac{B}{R+2}$ Trivium applications per second.

To scale up our attack, we assume that we guess a total of r bits and achieve an overall time complexity (in Trivium computations) of

$$T_p := \frac{B((2^r - 1)T_f + T_s)}{R + 2},$$

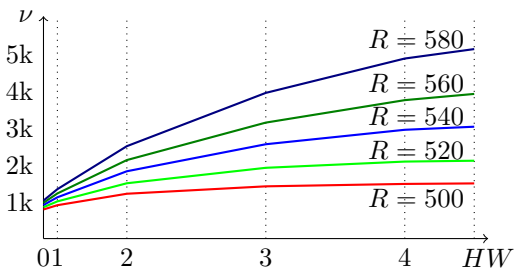
where T_s is the time of a successful computation of a solution and T_f is the time our solver needs to recognize a failure in the guessed variables; both are average times for a given system. As we see, T_f is dominant in our computations. This is in line with other algorithms in this area such as [BFP09]. Note that we assume that we get the data for all Trivium output as an input. So we do not calculate it which is a quite realistic assumption. Furthermore we know that we have a unique solution, namely the key of the cipher.

4 Experiments

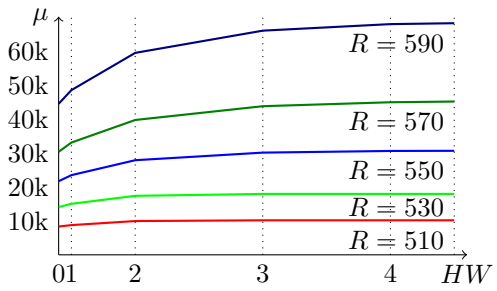
This section consists of three parts. First we consider the model and we see a saturation of variables and monomials when adding Trivium instances. Second we present on some parameter studies to further strengthen our system of equations. Finally, we use our insights to actually attack Trivium using the techniques described in the previous sections. We stress that the overall system of equations can be generated before we get the actual data for the output. This way our attack splits into an online and offline phase as outlined in Sec. 3.4. All experiments were done on the cluster described in 3.3.As we do not use any parallelizing techniques we are only using one core. Furthermore we want to stress that the online phase only requires a standard computer with 16Gb of RAM.

Figure 2: Saturation in the model of Trivium

(a) Hamming Weigth against number of variables ν for R rounds of Trivium in a master cube of dimension 5



(b) Hamming Weigth against number of quadratic monomials μ for R rounds for Trivium in a master cube of dimension 5



Saturation. When adding different Trivium instances with identical key variables but different IV constants that lie in the same master cube, the overall number of variables and monomials in the quadratic monomials of the overall system tends towards a saturation point (cf. Fig. 2a–2b). More specifically; we fix $80 - i$ IV bits to zero and set the remaining i bits to all possible values from $\mathbb{F}_2^i$.

We have plotted saturation for 32 instances in Figures 2a–2b, counting both the number of variables and the number of quadratic monomials needed for the system consisting of all instances of Trivium. In these figures, we have first added the IV with Hamming weight 0, then all IV with Hamming weight 1 and so forth. Note that instances with the same Hamming weight yield the same number of variables. As we can see in these graphs, the amount of variables needed to generate an instance becomes significantly *lower* if we generate instances with higher Hamming weight.

The saturation of monomials needs a lower Hamming weight of the IV, so the saturation of monomials is much flatter than the saturation of variables. Note that variables that are not in the quadratic (saturated) part of the system are only found in the linear terms. Furthermore, we stress that Figures 2a–2b are chosen only as an example. This effect also exists if the number of rounds increases (up to $R = 1152$). However, if the number of rounds grows we need to generate more instances to see this effect; it seems that we need to generate exponentially more instances to see the saturation. All in all, this points to a kind of "basis": Trivium instances for IV with small Hamming weight serve as a kind of basis for Trivium instances of higher Hamming weight. While this seems obvious when looking at the generating equations, it is still interesting to see how *strong* this effect is in practice. Unfortunately, we were unable to derive a closed formula depending on the number of instances and rounds but have to leave this as an open question.

In conclusion, saturation means that we can obtain more defining equations from many instances than we would expect from one instance alone. In a sense, this is the key observation to launch our attack.

Saturation should occur in other ciphers as well since the system of equation is generated by a repeatedly execution of an update function.

Note that we did not take the output equations into account yet but only those equations defining the system itself. While the output introduces additional, unknown monomials it will not add new variables as we will see in the next paragraph.

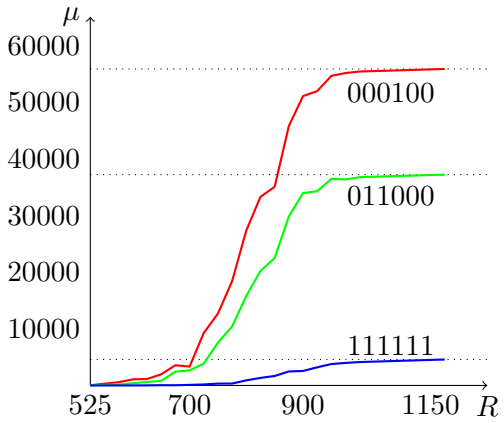
Output and parameters. While output equations clearly help us to linearize the system, the very structure of Trivium in our model yields a lot of additional monomials. Therefore, we do not add additional values for output equations or the output equations at all until the full (structural) system is completely simplified. Consider the output function:

$$z_i := c_{i-65} + c_{i-110} + a_{i-65} + a_{i-92} + b_{i-68} + b_{i-83}\,.$$

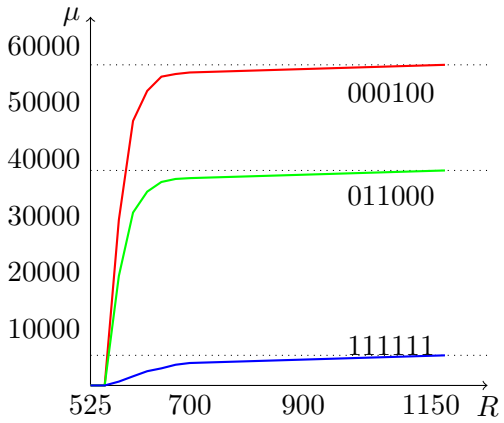
It uses 6 state bits from different rounds. If we insert for either of these state bits it produces 5 more monomials for each occurrence of the corresponding state bit in a *quadratic* monomial. Hence using more output bits per instance leads to far more monomials than we can afford.

Figure 3: Comparing monomials with $n_o = 1$ and $n_o = 66$; Initialization rounds R against number of quadratic monomials μ; The numbers next to the lines are the significant parts of the IV (binary). The rest of the IV is zero. Consider the Hamming weight of these numbers.

(a) Initialization rounds R against number of quadratic monomials μ for $n_o = 1$



(b) Initialization rounds against number of quadratic monomials μ for $n_o = 66$



To produce the smallest number of monomials possible in our system we change the algorithm to generate the system (cf. Sec 2). Instead of going forward and generating the Trivium instance we start with the output and go backwards and just generate the variables we need. This way, we just generate the variables and monomials we need. Note that this is contrary to earlier algebraic modellings of Trivium such as [SR12, Rad06].

Figure 3 compares systems with $T = 64$ instances and increasing number of rounds R. In the first experiment we used $n_o = 1$ and in the second $n_o = 66$. We see that many output bits do not necessarily lead to a more useful system because we get much more monomials. Even if we use two output equations we get a system with nearly double the number of monomials which we cannot solve easier. Note that figure 3 reflects the monomials needed for one instance while figure 2b shows the number of monomials needed for the whole system of equations. Therefore the saturation can also be seen in figure 3.

When we use $n_o = 66$ output bits the number of monomials at $R = 700$ rounds is negligibly smaller than the number of monomials for full Trivium ($R = 1152$). We choose $n_o = 66$ because for $n_o > 66$ we need to introduce new intermediate variables even for the output and that destroys the purpose of (over)defining the system for the linearization step. For $n_o = 1$ output bits, the same effect occurs for $R = 925$ rounds. Since we want to linearize our system to derive a solution we get the following conjecture.

Conjecture 1. *The complexity of our attack on Trivium does not grow after $R = 925$ rounds using $n_o = 1$ output bit and after $R = 700$ rounds using $n_o = 66$ output bits. That means if we are able to break $R = 925$ rounds with one output bit we are able to break full Trivium with one output bit.*

In a nutshell: Since the number of monomials does not increase, neither does the difficulty of the attack. Unfortunately, both settings are out of reach for a practical test at the moment.

In addition, we obtained the following result: When choosing 2^i instances of Trivium, it is optimal to arrange them in a master cube rather than sampling them "at random" from all possible 2^{80} IV. In particular, such a system becomes fully unsolvable with our methods.

Attacks. In this paragraph we describe the attacks and their complexities.

In Figure 3 we have illustrated the number of monomials depending on the number of output bits. With this in mind we have specialized our attack to the case $n_o = 1$. Based on this, we generate a system with Trivium-625 instances.

The following table shows the number of monomials and variables needed for the full system. This also includes "dummy" variables for the output. When we add concrete data for the output bits to

our system these numbers decrease rapidly. Furthermore we can see that there is a time-data trade-off when guessing variables. When we guess fewer variables we need more data to launch the attack. When we guess more variables we need less data but the time complexity increases.

R	μ	ν	#guessed variables	data complexity	time complexity
625	$\mathbf{499,741}$	$\mathbf{15,869}$	23	2^{11}	$2^{59.7}$
625	$1,135,858$	$32,518$	$\mathbf{0}$	$\mathbf{2} \cdot 2^{11}$	$\mathbf{2^{42.2}}$

When our guess was incorrectly, it takes our solver on average $2^{11.6}$ seconds to report that the system was inconsistent. However, when we guess correctly the system is solved in $2^{13.8}$ seconds on average.

We tested 101 keys each both for correct and incorrect guesses. We only present the numbers for the "incorrect guess"; the ones for the "correct guess" were similar: The fastest run was $2^{10.7}$, the slowest $2^{13.48}$ seconds, 95% of all runs were completed in under $2^{12.8}$ seconds with a variance of 1435 seconds. Using the reasoning from Sec. 3.4, our attack equals $2^{59.7}$ Trivium-625 computations and is therefore more efficient than brute force (2^{80}). We note that we do not actually compute the guessing step of this attack since it is not feasible to do so.

When we do not guess variables, we need more data and though more instances in our symbolic system. Generating the full symbolic system becomes a challenge due to the size of the system and RAM usage in the offline-phase. Thus we generate two systems consisting of 2^{11} instances each. The two systems do not profit from each other through similar variables in the offline-phase so the number of variables and the number of monomials is more than doubled. In the online-phase of the attack each system is reduced due to the linear output equations and similar variables. In our example in the table above we solved the system in $2^{17.1}$ seconds which leads to $2^{42.2}$ *Trivium computations* on average. Again, this experiment was conducted 101 times.

Unfortunately, we are unable to find a closed formula to predict the number of instances we would need to solve a system for a given number of rounds, as the behaviour of Trivium and the solver is quite erratic in this respect.

The real bottleneck of our attack is the generation of a symbolic system for a useful number of instances in the offline-phase. We can overcome this problem with a better implementation of the linear algebra or the ElimLin algorithm. However, we still cannot really resolve the exponential growth starting at $R = 700$ (Fig. 3) which works as a kind of barrier for our techniques used to attack Trivium. We want to encourage others to further improve or enhance the techniques used in this paper.

5 Conclusions

In this paper we have shown that algebraic attacks can be significantly improved. We achieve this by enhancing the ElimLin algorithm with techniques from eXtended Linearization and using the proper ordering; in particular the last proved crucial in our experiments. In addition, we have seen that using *many* instances of Trivium rather than only one with a long key stream significantly improves the attack. All in all, we were able to break a 625 round reduced version of Trivium in practical time ($2^{42.2}$ Trivium computations) and a data complexity of 2^{12}. Other key recovery attacks on Trivium can do better in terms of rounds with $R = 799$ but they requires a large amount of data (2^{40} bits) and time 2^{62} while guessing 62 bits. It is doubtful if this rate can be achieved in practice. An advantage of our approach is that we actually computed the full attack and did *not* make extrapolations from our results, as we do not want to make promises which are hard to keep.

An open questions is the existence of weak keys. They would improve our attack by reducing the overall number of monomials. While some experiments point at their existence they still elude a full characterization. Another line of research is the integration of more toolboxes into our solver, most notably SAT-solvers and more efficient sparse linear algebra packages.

While our experiments were conducted only on Trivum, we are confident that the ideas and lessons learnt are also useful for the algebraic cryptanalysis of other symmetric primitives, such as

block ciphers or hash functions. We want to stress that the potential of algebraic cryptanalysis can only be unleashed if equal stress is put on modelling techniques and the corresponding solver.

Acknowledgements

The first author wants to thank Wolfram Koepf (University of Kassel) for fruitful discussions and guidance. Both authors gratefully acknowledges an Emmy Noether Grant of the Deutsche Forschungsgemeinschaft (DFG).

References

[AAB+] Tim Abbott, Martin Albrecht, Gregory Bard, Marco Bodrato, Michael Brickenstein, Alexander Dreyer, Jean-Guillaume Dumas, William Hart, David Harvey, Jerry James, David Kirkby, Clément Pernet, Wael Said, and Carlo Wood. M4RI(e)—Linear Algebra over F_2 (and F_2^e). http://m4ri.sagemath.org/.

[ADMS09] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. In *FSE 2009, Fast Software Encryption*, pages 1–22. Springer, 2009.

[AFI+04] Gwnol Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between XL and Gröbner basis algorithms. In *ASIACRYPT 2004, LECTURE*, pages 338–353. Springer-Verlag, 2004.

[AK03] Frederik Armknecht and Matthias Krause. Algebraic Attacks on Combiners with Memory. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2003.

[Alb10] Martin Albrecht. *Algorithmic Algebraic Techniques and their Application to Block Cipher Cryptanalysis*. PhD thesis, Royal Holloway, University of London, 2010.

[BC03] Alex Biryukov and Christophe De Cannière. Block ciphers and systems of quadratic equations. In *FSE*, volume 2887 of *Lecture Notes in Computer Science*, pages 274–289. Thomas Johansson, editor, Springer, 2003. ISBN 3-540-20449-0.

[BD09] Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Groebner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326 – 1345, 2009. Effective Methods in Algebraic Geometry.

[BFP09] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *In Journal of Mathematical Cryptology*, 3:177–197, 2009.

[CB07] Nicolas T. Courtois and Gregory V. Bard. Algebraic Cryptanalysis of the Data Encryption Standard. In *Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 152–169. Springer Berlin Heidelberg, 2007.

[CKPS00] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Bart Preneel, editor, Springer, 2000. Extended Version: http://www.minrank.org/xlfull.pdf.

[CM03] Nicolas T. Courtois and Willi Meier. Algebraic Attacks on Stream Ciphers with Linear Feedback. In *Proceedings of the 22Nd International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'03, pages 345–359. Springer-Verlag, 2003.

[CMR05] Carlos Cid, Sean Murphy, and Matthew J. B. Robshaw. Small Scale Variants of the AES. In *FSE*, volume 3557 of *Lecture Notes in Computer Science*, pages 145–162. Henri Gilbert and Helena Handschuh, editors, Springer, 2005. ISBN 3-540-26541-4.

[CP02] Nicolas T. Courtois and Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of equations. In *ASIACRYPT*, pages 267–287, 2002.

[CP08] C. De Cannire and B. Prenel. Trivium. In *New Stream Cipher Designs*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008.

[CSSV12] Nicolas T. Courtois, Pouyan Sepehrdad, Petr Susil, and Serge Vaudenay. Elimlin Algorithm Revisited. In *Fast Software Encryption—FSE 2012*, pages 306–325, 2012.

[Die04] Claus Diem. The XL-Algorithm and a Conjecture from Commutative Algebra. In *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2004.

[DS09] I. Dinur and A. Shamir. Cube Attacks on tweakable black box polynomials. In *EUROCRYPT. Lecture Notes in Computer Science*, volume 5479, pages 278–299. Springer, 2009.

[Ecr12] ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012), rev. 1.0. http://www.ecrypt.eu.org/documents/D.SPA.20.pdf, Date: 2012-09-30 2012.

[FA03] Jean-Charles Faugère and Gwénolé Ars. An Algebraic Cryptanalysis of Nonlinear Filter Generators using Gröbner bases. Rapport de recherche 4739, February 2003. www.inria.fr/rrrt/rr-4739.html.

[Fau02a] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *International Symposium on Symbolic and Algebraic Computation — ISSAC 2002*, pages 75–83. ACM Press, July 2002.

[Fau02b] Jean-Charles Faugère. A New Efficient Algorithm for Computing Grbner Bases (F4). In *IN: ISSAC 02: PROCEEDINGS OF THE 2002 INTERNATIONAL SYMPOSIUM ON SYMBOLIC AND ALGEBRAIC COMPUTATION*, pages 75–83. Springer, 2002.

[FJ03] Jean-Charles Faugère and Antoine Joux. Algebraic Cryptanalysis of Hidden Field Equation (HFE) Cryptosystems Using Gröbner Bases. In *In Advances in Cryptology CRYPTO 2003*, pages 44–60. Springer, 2003.

[FOPT10] Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. Algebraic cryptanalysis of McEliece variants with compact keys. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 279–298. Henri Gilbert, editor, Springer, 2010. ISBN 978-3-642-13189-9.

[FV13] P.A. Fouque and T. Vannet. Improving Key Recovery to 784 and 799 rounds of Trivium using Optimized Cube Attacks. FSE 2013, Fast Software Encryption, 2013.

[GB08] Tim Good and Mohammed Benaissa. Hardware performance of eStream phase-III stream cipher candidates. SASC 2008 - The State of the Art of Stream Ciphers, 2008. http://www.ecrypt.eu.org/stvl/sasc2008/, pages 163–173.

[KHK06] Shahram Khazaei, Mahdi M. Hasanzadeh, and Mohammad S. Kiaei. Linear Sequential Circuit Approximation of Grain and Trivium Stream Ciphers. Cryptology ePrint Archive, Report 2006/141, 2006. http://eprint.iacr.org/2006/141/.

[KMNP11] Simon Knellwolf, Willi Meier, and María Naya-Plasencia. Conditional Differential Cryptanalysis of Trivium and KATAN. In *Selected Areas in Cryptography*, pages 200–212, 2011.

[KS99] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE public key cryptosystem. In *Advances in Cryptology — CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 19–30. Michael Wiener, editor, Springer, 1999. http://www.minrank.org/hfesubreg.ps or http://citeseer.nj.nec.com/kipnis99cryptanalysis.html.

[LK07] Chu-Wee Lim and Khoongming Khoo. An Analysis of XSL Applied to BES. In *Fast Software Encryption, FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 242–253. Springer, 2007.

[MR02] Sean Murphy and Matthew J.B. Robshaw. Essential algebraic structure within the AES. In *Advances in Cryptology — CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Moti Yung, editor, Springer, 2002.

[Rad06] H. Raddum. Cryptanalytic results on Trivium. http://www.ecrypt.eu.org/stream/triviump3.html, 2006.

[S+14] W. A. Stein et al. *Sage Mathematics Software (Version 6.1)*. The Sage Development Team, 2014. http://www.sagemath.org.

[SFP08] Ilaria Simonetti, Jean-Charles Faugère, and Ludovic Perret. Algebraic Attack Against Trivium. In *First International Conference on Symbolic Computation and Cryptography, SCC 08*, LMIB, pages 95–102, Beijing, China, April 2008. http://www-polsys.lip6.fr/~jcf/Papers/SCC08c.pdf.

[Sha49] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–p715, 4th October 1949.

[SKPI07] Makoto Sugita, Mitsuru Kawazoe, Ludovic Perret, and Hideki Imai. Algebraic Cryptanalysis of 58-Round SHA-1. In *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.

[SR12] T.E. Schilling and H. Raddum. Analysis of Trivium using compressed right hand side equations. In *Information Security and Cryptology. Lecture Notes in Computer Science*, volume 7259, pages 18–32. Springer, 2012.

[Sta10] Paul Stankovski. Greedy Distinguishers and Nonrandomness Detectors. In *INDOCRYPT*, pages 210–226, 2010.

[T+13] S. Teo et al. Algebraic analysis of Trivium-like ciphers, 2013. http://www.eprint.iacr.org/2013/240.pdf.

[TW12] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In *Public Key Cryptography*, volume 7293 of *Lecture Notes in Computer Science*, pages 156–171. Marc Fischlin, Johannes Buchmann, Mark Manulic, editors, Springer, 2012. ISBN 978-3-642-30056-1.

[YC04a] Bo-Yin Yang and Jiun-Ming Chen. All in the XL family: Theory and practice. In *ICISC 2004*, pages 67–86. Springer, 2004.

[YC04b] Bo-Yin Yang and Jiun-Ming Chen. Theoretical analysis of XL over small fields. In *ACISP 2004*, volume 3108 of *LNCS*, pages 277–288. Springer, 2004.