Using Wiedemann's algorithm to compute the immunity against algebraic and fast algebraic attacks

Frédéric Didier

Projet CODES, INRIA Rocquencourt, Domaine de Voluceau,
78153 Le Chesnay cédex
frederic.didier@inria.fr

Abstract. We show in this paper how to apply well known methods from sparse linear algebra to the problem of computing the immunity of a Boolean function against algebraic or fast algebraic attacks. For an n-variable Boolean function, this approach gives an algorithm that works for both attacks in $O(n2^nD)$ complexity and $O(n2^n)$ memory. Here $D = \binom{n}{d}$ and d corresponds to the degree of the algebraic system to be solved in the last step of the attacks. For algebraic attacks, our algorithm needs significantly less memory than the algorithm in [ACG+06] with roughly the same time complexity (and it is precisely the memory usage which is the real bottleneck of the last algorithm). For fast algebraic attacks, it does not only improve the memory complexity, it is also the algorithm with the best time complexity known so far for most values of the degree constraints.

Keywords : algebraic attacks, algebraic immunity, fast algebraic attacks, Wiedemann's algorithm.

1 Introduction

Algebraic attacks and fast algebraic attacks have proved to be a powerful class of attacks against stream ciphers [CM03,Cou03,Arm04]. The idea is to set up an algebraic system of equations satisfied by the key bits and to try to solve it. For instance, this kind of approach can be quite effective [CM03] on stream ciphers which consist of a linear pseudo-random generator hidden with non-linear combining functions acting on the outputs of the generator to produce the final output.

For such an attack to work, it is crucial that the combining functions satisfy low degree relations. The reason for this is that it ensures that the algebraic system of equations verified by the secret key is also of small degree, which is in general essential for being able to solve it. This raises the fundamental issue of determining whether or not a given function admits non-trivial low degree relations [MPC04,Car04,DGM04,BP05,DMS05].

For algebraic attacks, in order to find relations of degree at most d satisfied by an n-variable combining Boolean function f, only two algorithmic approaches are known for the time being. The first one relies on Gröbner bases [FA03] and consists of finding minimal degree elements in the polynomial ideal spanned by the ANF of f and the field equations. The second strategy relies on linear algebra, more precisely we can associate to f and d a matrix M such that the elements in the kernel of M give us low degree relations. Building M is in general easy so the issue here is to find non-trivial elements in the kernel of a given matrix or to show that the kernel is trivial.

The linear approach has lead to the best algorithm known so far [ACG+06] that works in $O(D^2)$ where $D = \binom{n}{d}$. There is also the algorithm of [DT06] which performs well in practice and which is more efficient when d is small. Actually, when d is fixed and $n \to \infty$, this last algorithm will be able to prove the non-existence of low degree relation in $O(D)$ for almost all Boolean functions. Note however that if the ANF of f is simple or has a lot of structure, it is possible that the Gröbner basis approach outperforms these algorithms, especially if the number of variable is large (more than 30).

For fast algebraic attacks, only the linear algebra approach has been used. There are now two degree constraints $d < e$ and the best algorithms are the one of [ACG+06] working in $O(ED^2)$ where $E = \binom{n}{e}$ and the one of [BLP06] working in $O(ED^2 + E^2)$.

All these algorithms relying on the linear algebra approach use some refinements of Gaussian elimination in order to find the kernel of a matrix M. Efficiency is achieved using the special structure of M. We will use here a different approach. The idea is that the peculiar structure of M allows for a fast matrix vector product that will lead to efficient methods to compute its kernel. This comes from the following facts:

- There are algorithms for solving linear systems of equations which perform only matrix vector products. Over the finite fields, there is an adaptation of the conjugate gradient and Lanczos algorithm [COS86,Odl84] or the Wiedemann algorithm [Wie86]. These algorithms were developed at the origin for solving large sparse systems of linear equations where one can compute a matrix vector product efficiently. A lot of work has been done on the subject because of important applications in public key cryptography. Actually, these algorithms are crucial in the last step of the best factorization or discrete logarithm algorithms.
- Computing a matrix vector product of the matrix involved in the algebraic immunity computation can be done using only the binary Moëbius transform. It is an involution which transforms the two main representations of a Boolean function into each other (namely the list of its ANF coefficients and the list of its images).
- The Moëbius transform of an n-variable Boolean function can be computed efficiently in $O(n2^n)$ complexity and $O(2^n)$ memory. We will call the corresponding algorithm the fast Moëbius transform by analogy with the fast Fourier transform (note that they both rely on the same principle).

We will focus here on the Wiedemann algorithm and derive an algorithm for computing algebraic attacks or fast algebraic attacks relations in $O(n2^n D)$. When d or e are close to $n/2$ the asymptotic complexity is very good for fast algebraic attacks but is a little bit less efficient for algebraic attacks than the one presented in [ACG+06]. However this algorithm presents another advantage, its memory usage is very efficient, $O(n2^n)$ to be compared with $O(D^2)$. This may not seem really important, but in fact the memory is actually the bottleneck of the other algorithms.

The outline of this paper is as follows. We first recall in Section 1 some basic facts about Boolean functions, algebraic and fast algebraic attacks, and the linear algebra approach used by almost all the known algorithms. Then, we present in Section 2 the Wiedemann algorithm and how we can apply it to our problem. We present in Section 3 some benchmark results of our implementation of this algorithm. We finally conclude in Section 4.

2 Preliminary

In this section we recall basic facts about Boolean functions, algebraic attacks and fast algebraic attacks. We also present the linear algebra approach used by almost all the actual algorithm to compute relations for these attacks.

2.1 Boolean functions

In all this paper, we consider the binary vector space of n-variable Boolean functions, that is the space of functions from $\{0,1\}^n$ to $\{0,1\}$. It will be convenient to view $\{0,1\}$ as the field over two elements, what we denote by $\mathbf{F}_2$. It is well known that such a function f can be written in an unique way as an n-variable polynomial over $\mathbf{F}_2$ where the degree in each variable is at most 1 using the Algebraic Normal Form (ANF) :

$$f(x_1,\ldots,x_n) = \sum_{u\in\mathbf{F}_2^n} f_u \left(\prod_{i=1}^{n} x_i^{u_i}\right) \quad f_u \in \mathbf{F}_2, u=(u_1,\ldots,u_n)\in\mathbf{F}_2^n \qquad (1)$$

By *monomial*, we mean in what follows, a polynomial of the form $\prod_{i=1}^{n} x_i^{u_i}$. We will heavily make use in what follows of the notation f_u, which denotes for a point u in $\mathbf{F}_2^n$ and an n-variable Boolean function f, the coefficient of the monomial associated to u in the ANF (1) of f. Each monomial associated to a u in $\mathbf{F}_2^n$ can be seen as a function having only this monomial as its ANF. Such function is only equal to 1 on points x such that $u \subseteq x$ where

$$\text{for } u,x\in\mathbf{F}_2^n \quad u\subseteq x \quad \text{iff} \quad \{i,u_i\neq 0\}\subseteq\{i,x_i\neq 0\} \qquad (2)$$

The *degree* of f is the maximum weight of the u's for which $f_u \neq 0$. By listing the images of a Boolean function f over all possible values of the variables, we

can also view it as a binary word of length 2^n. For that, we will order the points of $\mathbf{F}_2^n$ in lexicographic order

$$(0,\ldots,0,0)(0,\ldots,0,1)(0,\ldots,1,0)\ldots(1,\ldots,1,1) \tag{3}$$

The *weight* of a Boolean function f is denoted by $|f|$ and is equal to $\sum_{x\in\mathbf{F}_2^n} f(x)$ (the sum being performed over the integers). We also denote in the same way the (Hamming) weight of a binary 2^n-tuple or the cardinal of a set. A *balanced* Boolean function is a function with weight equal to half its length, that is 2^{n-1}.

There is an important involutive (meaning its own inverse) transformation linking the two representations of a Boolean function f, namely its image list $(f(x))_{x\in\mathbf{F}_2^n}$ and its ANF coefficient list $(f_u)_{u\in\mathbf{F}_2^n}$. This transformation is known as the *binary Moëbius transform* and is given by

$$f(x)=\sum_{u\subseteq x} f_u \quad \text{and} \quad f_u=\sum_{x\subseteq u} f(x) \tag{4}$$

Here u and x both lie in $\mathbf{F}_2^n$ and we use the notation introduced in (1) for the ANF coefficients of f.

Dealing with algebraic attacks, we will be interested in the subspace of all Boolean functions of degree at most d. Note that the set of monomials of degree at most d forms a basis of this subspace. By counting the number of such monomials we obtain that its dimension is given by $D\stackrel{\text{def}}{=}\sum_{i=0}^{d}\binom{n}{i}$. In the following, a Boolean function g of degree at most d will be represented by its ANF coefficients $(g_i)_{|i|\leq d}$. Notice as well that we will need another degree constraint e and that we will write E for the dimension of the subspace of Boolean functions with degree at most e.

2.2 Algebraic and fast algebraic attacks

We will briefly describe here how algebraic and fast algebraic attacks work on a filtered LFSR. In the following, L is an LFSR on n bits with initial state $(x_1,\ldots,x_n)$ and a filtering function f. The idea behind algebraic attacks is just to recover the initial state given the keystream bits $(z_i)_{i\geq 0}$ by solving the algebraic system given by the equations $f(L^i(x_1,\ldots,x_n))=z_i$. However the algebraic degree of f is usually too high, so one has to perform further work.

In the original algebraic attacks [CM03], the first step is to find *annihilators* of f. This means functions g of low degree such that $fg=0$ where fg stands for the function defined by

$$\forall x\in\mathbf{F}_2^n,\quad fg(x)=f(x)g(x) \tag{5}$$

in particular $fg=0$ if and only if

$$\forall x\in\mathbf{F}_2^n,\quad f(x)=1 \quad\Longrightarrow\quad g(x)=0 \tag{6}$$

So we obtain a new system involving equations of the form

$$g(L^i(x_1,\ldots,x_n))=0 \quad \text{for} \;\; i\geq 0 \;\; \text{and} \;\; z_i=1 \tag{7}$$

that can be solved if the degree d of g is low enough. Remark that for the i's such that $z_i = 0$, we can use the same technique with the annihilators of $1 + f$ instead.

In order to quantify the resistance of a function f to algebraic attacks, the notion of algebraic immunity was introduced in [MPC04]. By definition, the algebraic immunity of f is the smallest degree d such that f or $1 + f$ admits a non-trivial annihilator of degree d. It has been shown in [CM03] that for an n-variable Boolean function a non-trivial annihilator of degree at most $\lceil n/2 \rceil$ always exists.

Sometimes, annihilators of low degree do not exist, but another relation involving f can be exploited. That is what is done in fast algebraic attacks introduced in [Cou03] and further confirmed and improved in [Arm04,HR04]. The aim is to find a function g of low degree d and a function h of larger degree e such that $fg = h$. We now get equations of the form

$$z_i \; g(L^i(x_0, \ldots, x_n)) = h(L^i(x_0, \ldots, x_n)) \quad \text{for} \quad i \geq 0 \tag{8}$$

In the second step of fast algebraic attacks, one has to find a linear relation between successive equations [Cou03] in order to get rid of the terms with degree greater than d. Remarks that these terms come only from h and so such a relation does not involve the keystream bits z_i. More precisely, we are looking for an integer l and binary coefficients c_i such that all the terms of degree greater than d cancel out in the sum

$$\sum_{i=0}^{i<l} c_i \; h(L^i(x_0, \ldots, x_n)) \tag{9}$$

One can search this relation offline and apply it not only from time 0 but also shifted at every time $i \geq 0$. In the end, we get an algebraic system of degree d that we have to solve in the last step of the attack.

In this paper, we will focus on the first step of these attacks. We are given a function f and we will discuss algorithms to compute efficiently its immunity against algebraic and fast algebraic attacks.

2.3 Linear algebra approach

We will formulate here the problem of finding low-degree relations for a given function f in terms of linear algebra. In the following, all the lists of points in $\mathbf{F}_2^n$ are always ordered using the order defined in Subsection 2.1.

Let us start by the case of the classical algebraic attacks. Recall that a function g of degree at most d is an annihilator of f if and only if

$$\forall x \in \mathbf{F}_2^n, \quad f(x) = 1 \quad \Longrightarrow \quad g(x) = 0 \tag{10}$$

So, for each x such that $f(x) = 1$ we get from (4) a linear equation in the D coefficients $(g_u)_{|u| \leq d}$ of g, namely

$$\sum_{u \subseteq x} g_u = 0 \quad u \in \mathbf{F}_2^n, \; |u| \leq d \tag{11}$$

This give rise to a linear system that we can write $M_1((g_u)_{|u|\leq d})^t = 0$ where M_1 is an $|f| \times D$ binary matrix and the t indicates transposition. Each row of M_1 corresponds to an x such that $f(x) = 1$. Actually, we see that the matrix vector product of M_1 with $((g_u)_{|u|\leq d})^t$ is just an evaluation of a function g with ANF coefficients $(g_u)_{|u|\leq d}$ on all the points x such that $f(x) = 1$. We will encounter again such type of matrices and we will introduce a special notation. Let $\mathcal{A}$ and $\mathcal{B}$ be two subsets of $\mathbf{F}_2^n$, we will write

$$V_{\mathcal{B}}^{\mathcal{A}} = (v_{i,j})_{i=1\ldots|\mathcal{B}|,j=1\ldots|\mathcal{A}|} \tag{12}$$

for the matrix corresponding to an evaluation over all the points in $\mathcal{B}$ of a Boolean function with non-zero ANF coefficients in $\mathcal{A}$. The V stands for evaluation, and we have $v_{i,j} = 1$ if and only if the j-th point of $\mathcal{B}$ is included (notation $\subseteq$ over $\mathbf{F}_2^n$) in the i-th point of $\mathcal{A}$. With this notation, we get

$$M_1 = V_{\{x,f(x)=1\}}^{\{u,|u|\leq d\}} = V_{\{x,f(x)=1\}}^{d} \tag{13}$$

The exponent d is a shortcut for $\{u, |u| \leq d\}$, in particular an exponent n means all the points in $\mathbf{F}_2^n$. It is important to understand this notion of evaluation because if we look a little ahead, we see that performing a matrix vector product for such a matrix is nothing but performing a binary Moëbius transform.

Now, a function g with ANF $(g_u)_{|u|\leq d}$ is an annihilator of f if and only if $((g_u)_{|u|\leq d})^t \in \ker(M_1)$. A non-trivial annihilator of degree smaller than or equal to d exists if and only if this matrix is not of full rank.

For the fast algebraic attacks, we obtain the same description with a different linear system. Functions g of degree at most d and h of degree at most e such that $fg = h$ exist if and only if

$$\forall x \in \mathbf{F}_2^n \quad h(x) + f(x)g(x) = 0 \tag{14}$$

Here the unknowns are the D coefficients $(g_u)_{|u|\leq d}$ of g and the E coefficients $(h_u)_{|u|\leq e}$ of h. So, for each point x we derive by using (4) the following equation on these coefficients

$$\sum_{u\subseteq x,|u|\leq e} h_u \quad + \quad f(x) \sum_{u\subseteq x,|u|\leq d} g_u = 0 \tag{15}$$

And we obtain a system what we can write $M_2((h_u)_{|u|\leq e}, (g_u)_{|u|\leq d})^t = 0$ where M_2 is an $N \times (E + D)$ binary matrix given by

$$M_2 = \left(V_n^e \;\mid\; \mathrm{Diag}((f(x))_{x\in\mathbf{F}_2^n})V_n^d\right) \tag{16}$$

The multiplication by f in (15) corresponds here with the product by the diagonal matrix. With this new matrix, each kernel element corresponds to functions g and h such that $fg = h$.

There is a way to create a smaller linear system for the fast algebraic attacks. This follows the idea in [BLP06]. Actually, as pointed out in [DM06], the matrix V_e^e is an involutive $E \times E$ matrix. The idea is then to start by computing the

$(h_u)_{|u|\leq e}$ using the values that h has to take on the points x with $|x| \leq e$. That is, taking for the unknowns the D coefficients $(g_u)_{|u|\leq d}$ of g, the values $(h(x))_{|x|\leq e}$ of h are given by

$$((h(x))_{|x|\leq e})^t = \mathrm{Diag}((f(x))_{|x|\leq e})V_e^d((g_u)_{|u|\leq d})^t \tag{17}$$

We can then find the ANF coefficients $(h_u)_{|u|\leq e}$ of h by applying V_e^e on the left because this matrix is involutive. We can then evaluate h over all $\mathbf{F}_2^n$ by multiplying on the left by V_n^e. In the end, we obtain a new linear system $M_3((g_u)_{|u|\leq d})^t = 0$ where M_3 is the following $N \times D$ matrix

$$M_3 = \mathrm{Diag}((f(x))_{x\in\mathbf{F}_2^n})V_n^d + V_n^e V_e^e \mathrm{Diag}((f(x))_{|x|\leq e})V_e^d \tag{18}$$

Here $\mathrm{Diag}((f(x))_{x\in\mathbf{F}_2^n})V_n^d$ corresponds to the evaluation of fg on all the points in $\mathbf{F}_2^n$ and the other part to the evaluation of h on the same points. Remark that the rows corresponding to $|x| \leq e$ are null by construction, so M_3 can be reduced to an $(N - E) \times D$ matrix.

Up to now, all the known algorithms relying on the linear algebra approach ([MPC04,DT06,BLP06,ACG+06]) worked by computing the kernel of these matrices using some refinements of Gaussian elimination. Efficiency was achieved using the very special structure involved. We will use here a different approach. The idea is that the special structure behind these linear systems allows a fast matrix vector product. We will actually be able to compute $M_1((g_i)_{|i|\leq d})^t$ or $M_3((g_i)_{|i|\leq d})^t$ in $O(n2^n)$. This will lead to an algorithm in $O(n2^n D)$ complexity and $O(n2^n)$ memory.

3 Using Wiedemann's algorithm

In this section we describe how the Wiedemann algorithm [Wie86] can be used efficiently on our problem. We focused on this algorithm (instead of Lanczos' or conjugate gradient algorithm) because it is easier to analyze and it does not need any assumption on the matrix.

3.1 Fast evaluation

The first ingredient for Wiedemann's algorithm to be efficient on a given matrix, is that we can compute the matrix vector product for this particular matrix efficiently. This is for example the case for a sparse matrix and will also be the case for the matrices M_1, M_2 or M_3 involved in the algebraic immunity computation. In the following we still use the order defined in Subsection 2.1 for all the lists of points in $\mathbf{F}_2^n$.

So we want to compute efficiently a matrix vector product of M_1, M_2 or M_3. For that, looking at the definition of these matrices (see (13), (16) and (18)) it is enough to be able to compute efficiently a matrix vector product of diagonal matrices and of the matrices $V_{\mathcal{B}}^{\mathcal{A}}$ ($\mathcal{A}$ and $\mathcal{B}$ being two subsets of $\mathbf{F}_2^n$). Then,

we will just compute this kind of product for all the matrices appearing in the previous definitions to get the final product.

Computing a product between a diagonal matrix and a vector is easy, it can be computed in $O(2^n)$ using a binary AND between the vector and the list of the diagonal elements. Regarding the matrices $V_{\mathcal{B}}^{\mathcal{A}}$, performing the product is almost the same as doing a Moëbius transform as we have seen in Subsection 2.3. The details are explained in the following algorithm.

Algorithm 1 (Matrix vector product of $V_{\mathcal{B}}^{\mathcal{A}}$) Given n, two subsets ($\mathcal{A}$ and $\mathcal{B}$) of $\mathbf{F}_2^n$ and a vector $v = (v_i)_{i=1\ldots|\mathcal{A}|}$, this algorithm computes the matrix vector product of $V_{\mathcal{B}}^{\mathcal{A}}$ and v.

1. [pack] Initialize a vector $s = (s_u)_{u\in\mathbf{F}_2^n}$ as follows If u is the i-th point in $\mathcal{A}$ then set $s_u = v_i$. Otherwise (that is $u \notin \mathcal{A}$) set $s_u = 0$.
2. [Moëbius] Compute the fast binary Moëbius transform of s in place.
3. [Extract] The result is given by the (s_u) with $u \in \mathcal{B}$.

So, the key point in a fast matrix vector product here is that we can compute the binary Moëbius transform efficiently. The following algorithm called the fast Moëbius transform works in $\mathrm{O}(n2^n)$ and uses the same idea as the fast Fourier transform algorithm. In the end, we are able to perform a matrix vector product of M_1, M_2 or M_3 with the same complexity. Remark as well that for all these algorithms, the memory usage is in $O(2^n)$.

Algorithm 2 (Fast binary Moëbius transform) Given an n-variable Boolean function f in the form of a list of ANF coefficients $(f_u)_{u\in\mathbf{F}_2^n}$, this algorithm computes its image list $(f(x))_{x\in\mathbf{F}_2^n}$ recursively. In both cases the list must be ordered using the order described in section 2.1. The algorithm can work in place (meaning the result overwrites the $(f_u)_{u\in\mathbf{F}_2^n}$) without modifications.

1. [stop] If $n = 0$ then $f(0) = f_0$. Exit the function.
2. [left recursion] Perform the Moëbius transform for a $n-1$ variable function $f^{(0)}$ whose coefficients are given by the first half of the coefficient list of f, that is the f_u's with $u = (u_1, \ldots, u_n)$ and $u_1 = 0$.
3. [right recursion] Perform the Moëbius transform for a $n-1$ variable function $f^{(1)}$ whose coefficients are given by the second half of the coefficient list of f, that is the f_u with $u = (u_1, \ldots, u_n)$ and $u_1 = 1$.
4. [combine] We have $f(x_1, \ldots, x_n) = f^{(0)}(x_2, \ldots, x_n) + x_1 f^{(1)}(x_2, \ldots, x_n)$.

The complexity in $O(n2^n)$ comes from the fact that at each call, we apply the algorithm over two problems of half the size of the original one. The correctness is easy to prove provided that the equality at step 4 is correct. But using the definition of the Moëbius transform, we have

$$f(x) = \sum_{u\subseteq x} f_u(x) = \sum_{u\subseteq x, u_1=0} f_u(x) + \sum_{u\subseteq x, u_1=1} f_u(x) \tag{19}$$

The second sum is zero if $x_1 = 0$, so we can write

$$f(x) = \sum_{(u_2,\ldots,u_m)\subseteq(x_2,\ldots,x_n)} f_{(0,u_2,\ldots,u_n)}(x) \quad + \quad x_1 \sum_{(u_2,\ldots,u_n)\subseteq(x_2,\ldots,x_n)} f_{(1,u_2,\ldots,u_n)}(x) \quad (20)$$

and we retrieve the equality at step 4.

3.2 The Wiedemann Algorithm

We present here the Wiedemann algorithm for an $n \times n$ square matrix A. We will deal with the non-square case in the next subsection.

The approach used by Wiedemann's algorithm (and more generally blackbox algorithms) is to start from a vector b and to compute the so called Krylov sequence

$$b, Ab, A^2b, \ldots, A^nb, \ldots \quad (21)$$

This sequence is linearly generated and admits a minimal polynomial $P_b \in F_2[X]$ such that $P_b(A)b = 0$. Moreover, P_b divides the minimal polynomial of the matrix A and is of maximum degree n.

The idea of Wiedemann's algorithm is to find P_b using the Berlekamp-Massey algorithm. For that, we take a random vector u^t in $\mathbf{F}_2^n$ and compute the inner products

$$u.b, u.Ab, u.A^2b, \ldots, u.A^{2n}b \quad (22)$$

The complexity of this step is in $2n$ evaluations of the matrix A. This sequence is still linearly generated, and we can find its minimal polynomial $P_{u,b}$ in $O(n^2)$ using the Berlekamp-Massey Algorithm ([Mas69]). Moreover, $P_{u,b}$ divides P_b and they are equal with probability bounded away from 0 by $1/(6 \log n)$ (see [Wie86]). Notice that if X divides $P_{u,b}$, then A is singular since 0 is then one of its eigenvalues.

Now, let us assume that we have computed P_b and that $P_b(x) = c_0 + XQ(x)$ with $Q(x) \in \mathbf{F}_2[X]$. If $c_0 \neq 0$ (and therefore $c_0 = 1$) then $AQ(A)b = b$ and $Q(A)b$ is a solution x of the system $Ax = b$. If $c_0 = 0$ then $AQ(A)b = 0$ and $Q(A)b$ is a non-trivial (by minimality of P_b) element of $\ker(A)$. So, we can either find a solution of $Ax = b$ or a non-trivial element in $\ker(A)$ with complexity the number of steps of computing $Q(A)$ that is n evaluations of A.

Remark that in both cases we can verify the coherence of the result (even when $P_{u,b} \neq P_b$) with only one evaluation of A. Moreover, when A is singular, we are sure to find a non-trivial kernel element if b does not lie in $\mathrm{Im}(A)$. This happens with probability always greater than $1/2$ over $\mathbf{F}_2$.

If we are only interested in knowing if a matrix is singular then, as already pointed out, we just need that X divides $P_{u,b}$ and we have:

Theorem 1 *If an $n \times n$ square matrix A over F_2 is singular, then applying Wiedemann's algorithm with a random choice of b and u will prove that the matrix is singular with probability greater than $1/4$ and $O(n)$ evaluations of A.*

Proof. Let us decompose $E = \mathbf{F}_2^n$ into the characteristic subspaces of A. In particular we have $E = E_0 \oplus E_1$ where E_0 is the subspace associated with the eigenvalue 0 and A restricted to E_1 is invertible. Using this decomposition, let us write $b = b_0 + b_1$. Let P_0 and P_1 be the minimal polynomial associated to the sequences $(u.b_0, u.Ab_0, \dots)$ and $(u.b_1, u.Ab_1, \dots)$. We know that P_0 is just a power of X and that the LFSR generating the second sequence is non degenerate. So the minimal polynomial associated to the sum is equal to P_0P_1. To conclude, we see that $X/P_{u,b}$ if $u.b_0 \neq 0$ and this happens for a random choice of u and b with a probability greater than $1/4$.

In the end, the algorithm consists in trying different values of b and u until we have a large enough probability that A is singular or not. When b is fixed, we just described a Monte-Carlo algorithm here but there is also a Las-Vegas version (Algorithm 1 of [Wie86]) that works better in this case. It gives P_b (so at the end a kernel element or a solution to $Ax = b$) in $O(n \log n)$ matrix evaluations on average.

Remark that when X divides $P_{u,b}$ we are sure that A is singular. So the three possible outputs of the algorithm are the following :
- either A is singular and we know it for sure,
- or we know that A is non-singular with a very high probability,
- finally when $P_{u,b}$ is of maximum degree (that is n) then we know the minimal polynomial of A. So if it is not divisible by X then we are sure that the matrix is of full rank.

3.3 Non square case

The square case could be applied directly when we try to show the maximum algebraic immunity of a balanced Boolean function with an odd number of variables (because in this case $d = (n-1)/2$ and $|f| = 2^{n-1}$ is equal to D). However, in the general case we do not have a square matrix.

One method to extend this could be to select randomly a subsquare matrix until we find an invertible one or until we have done so many choices that we are pretty sure that the initial matrix is not of full rank. This method is however quite inefficient when the matrix is far from being a square matrix and in this case there is a better way to perform this task.

Let us consider an $n \times k$ matrix A with $k < n$, the idea is to generate a $k \times n$ random sparse matrix Q such that with high probability QA will be of rank k if A is non-singular. From [Wie86] we have the following result

Theorem 2 *If A is a non-singular $n \times k$ matrix with $k < n$, let us construct a $k \times n$ matrix Q as follows. A bit of the row i from 1 to k is set to 1 with probability $w_i = \min(1/2, 2\log n/(k+1-i))$. Then with probability at least $1/8$ the following statements hold*
- *The $k \times k$ matrix QA is non-singular.*
- *The total Hamming weight of the rows in Q is at most $2n(2+\log n)^2$.*

Notice that in [Wie86], another generating method is given to generate Q such that QA is non-singular with a probability bounded away from 0 and such that the total Hamming weight of Q is in $O(n \log n)$. This is better asymptotically but less applicable in practice since the probability is smaller than the 1/8 of Theorem 2.

We are now back to the square case with a little overhead because we have to compute Q times a vector at each step of the Wiedemann algorithm. This is why we have generated Q as sparse as possible to minimize this overhead. In particular, when Q has a total Hamming weight in $O(n \log n)$ then we can perform the matrix vector product in $O(n \log n)$. Notice as well that we need $O(n \log n)$ extra memory in order to store the matrix Q.

In order to know if A is singular or not, the algorithm is the following. We generate a matrix Q and test the non singularity of QA with Wiedemann's Algorithm. If this matrix is non-singular, then we know that A is non-singular as well (with the failure probability of Wiedemann's algorithm). Otherwise, we can go on a few times (say i) with different matrices Q and if all the products are singular then A is singular with probability greater than $1 - (7/8)^i$.

Remark that with negligible complexity overhead, we can compute a non-trivial kernel element x of QA when this matrix is singular. And if A is singular, with a probability greater than 1/8 we will also have $Ax = 0$. So we can run the algorithm until we are sure that A is singular (when we get a non-trivial kernel element) or until we have a very high probability that A is non-singular.

4 Implementation results

We have implemented all the algorithms described in this article and we give their performances in this section. All the experiments were done on a Pentium 4 running at 3.2Ghz with 2Gb of memory.

First of all, let us summarize the complexity of our algorithms. Both for algebraic and fast algebraic attacks, we will have to perform Wiedemann's algorithm on a $D \times D$ matrix. This requires $O(D)$ matrix vector products of this matrix plus $O(D^2)$ operations for the Berlekamp-Massey algorithm. We have seen that we can perform the product in $O(n2^n)$ operations, so we get in all cases a final complexity in $O(n2^n D)$. Notice that in order to get a small failure probability, we will have to perform this task a few times. This is especially true for the non-square case since we will have to check different matrices Q. However, only a constant number of times is needed to get a given probability, so the asymptotic complexity is still the same.

Regarding the memory, the matrix evaluation needs $O(2^n)$ memory for the square case and an extra $O(n2^n)$ memory when we have to store a matrix Q for the non-square case. All the other operations need only an $O(D)$ memory.

The running time of the algorithms is almost independent of the function f involved except for the Berlekamp-Massey part. However, this part is clearly not the most time consuming. In particular, it is a good idea in Wiedemann's algorithm to perform the computation with more than one random u per vector

b. If we perform 4 Berlekamp-Massey steps per random vector b, then we will be able to detect a singularity with a probability almost one half and a very small overhead. Remark as well that all the experiments where done for random balanced functions but the running time will be roughly the same for real functions used in stream cipher.

When we compute the immunity against normal algebraic attacks for a square matrix, we can implement the code in a very efficient way. In particular, using the transposition of the Moëbius transform, we can merge step 1 and 3 of Algorithm 1 between two consecutive evaluations. This is because this transposition will map back all the positions in the set $\mathcal{B}$ into the set $\mathcal{A}$. Moreover, if M_1 is square and invertible, applying $(V^d_{\{x,f(x)=1\}})^t$ on the left will result in another invertible matrix. In this way, we obtain a fully parallelizable algorithm because we can perform a fast Moëbius transform (or its transpose) dealing with 32 bits at a time (even more with SSE2). In the end we get a very efficient implementation with the running time for a random choice of b and four random choices of u given in Table 1. Moreover, the memory usage is negligible in this case since there is no matrix Q to store.

d,n	4,9	5,11	6,13	7,15	8,17	9,19	10,21	11,23
time	0s	0s	0.01s	0.3s	5s	102s	30m	11h

Table 1. running time for the square case : $n = 2d + 1$, $D = 2^{n-1}$ and f is a random balanced n-variable Boolean function. Optimized implementation using the SSE2 instructions set.

In this case, since $n = 2D + 1$, the complexity is in $O(D^2 \log D)$ and the memory usage in $O(D)$. So the asymptotic complexity is a little worse than the one of [ACG+06] (in $O(D^2)$) but the memory usage is a lot better (to be compared with $O(D^2)$). We see that two consecutive sets of parameters differ by a factor of 16 in the computation time as could be inferred from the asymptotic complexity. It is then reasonable to think that we can go even further in a reasonable time. Notice that this implementation also breaks the previous record which was 20 variables in the papers [DT06] and [ACG+06].

For the non-square case however, the results are less impressive. The reason is that there is no simple way to do the multiplication by Q in a parallel way. Hence we loose a factor 32 in the process. To overcome this difficulty a block version of Wiedemann's algorithm might be use (see [Cop94]), but we did not have time to implement it. Another issue is that the memory to store the sparse matrix Q may become two large. Moreover, the code used for the following experiments is not as optimized as the one for the square case. We might divide the time and memory by a factor 2 roughly with a careful implementation.

We give in Table 2 the time to compute the immunity against normal algebraic attacks. Wiedemann's algorithm is executed for one random matrix Q a random b and four random u's. In order to obtain a small enough probability of

error, one will have to execute this a few times (16 gives an error probability of 0.1 and 32 of 0.01).

d,n	2,22	2,23	3,19	3,20	3,21	4,19	5,19
D	254	277	1160	1351	1562	5036	16664
time	113s	264s	100s	252s	630s	640s	2706s
memory	656Mb	1397Mb	118Mb	255Mb	547Mb	160Mb	194Mb

Table 2. time and memory for computing the immunity against normal algebraic attacks using Wiedemann's algorithm.

What is interesting is that the time for computing immunity against fast algebraic attacks is almost the same as the one for normal algebraic attacks (see Table 3). There is only little influence of the degree e (see Table 4) on the performance because the size of Q depends on it. But the time and memory will always stay within a factor two compared to the case where e is equal to $n/2$.

d/e,n	2/8,17	3/8,17	3/9,19	3/10,21	4/8,17	5/8,17	6/8,17
D	154	834	1160	1562	3214	9402	21778
time	1s	15s	101s	614s	82s	297s	801s
memory	14Mb	25Mb	118Mb	547Mb	33Mb	40Mb	45Mb

Table 3. time and memory for computing the immunity against fast algebraic attacks using Wiedemann's algorithm. Here we chose $n = 2e + 1$.

d/e,n	3/7,19	3/8,19	3/9,19	3/10,19	3/11,19
time	154s	130s	101s	70s	43s
memory	192Mb	159Mb	118Mb	77Mb	43Mb

Table 4. dependence of fast algebraic attacks immunity computation in the parameter e. In all cases.

5 Conclusion

In this paper, we devised a new algorithm to compute the immunity of a Boolean function against both algebraic and fast algebraic attacks. This algorithm presents a few advantages :

- It is easy to understand since it is based on a well known sparse linear algebra algorithm.
- Its complexity is quite good, especially for the fast algebraic attacks.
- It uses little memory compared to the other known algorithms which make it able to deal with more variables.
- And it is quite general since it can work for both attacks with little modification. In particular, if in the future one is interested in other kind of relations defined point by point, then the same approach can be used.

Acknoweldegement

I want to thank Jean-Pierre Tillich for his great help in writting this paper and Yann Laigle-Chapuy for his efficient implementation of the fast binary Moëbius transform in SSE2.

References

[ACG+06] Frederik Armknetcht, Claude Carlet, Philippe Gaborit, Simon Künzli, Willi Meier, and Olivier Ruatta. Efficient computation of algebraic immunity for algebraic and fast algebraic attacks. *EUROCRYPT 2006*, 2006.

[Arm04] Frederick Armknetch. Improving fast algebraic attacks. In *Fast Software Encryption, FSE*, volume 3017 of *LNCS*, pages 65–82. Springer Verlag, 2004. http://eprint.iacr.org/2004/185/.

[BLP06] An Braeken, Joseph Lano, and Bart Preneel. Evaluating the resistance of stream ciphers with lenear feedback agasint fast algebraic attacks. *To appears in ACISP 06*, 2006.

[BP05] An Braeken and Bart Preneel. On the algebraic immunity of symmetric Boolean functions. 2005. http://eprint.iacr.org/2005/245/.

[Car04] Claude Carlet. Improving the algebraic immunity of resilient and nonlinear functions and constructing bent functions. 2004. http://eprint.iacr.org/2004/276/.

[CM03] Nicolas Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. *Advances in Cryptology – EUROCRYPT 2003*, LNCS 2656:346–359, 2003.

[Cop94] Don Coppersmith. Solving linear equations over GF(2) via block Wiedemann algorithm. *Math. Comp.*, 62(205):333–350, January 1994.

[COS86] D. Coppersmith, A. Odlyzko, and R. Schroeppel. Discrete logarithms in GF(p). *Algorithmitica*, 1:1–15, 1986.

[Cou03] Nicolas Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology-CRYPTO 2003*, volume 2729 of *LNCS*, pages 176–194. Springer Verlag, 2003.

[DGM04] Deepak Kumar Dalai, Kishan Chand Gupta, and Subhamoy Maitra. Results on algebraic immunity for cryptographically significant Boolean functions. In *INDOCRYPT*, volume 3348 of *LNCS*, pages 92–106. Springer, 2004.

[DM06] Deepak Kumar Dalai and Subhamoy Maitra. Reducing the number of homogeneous linear equations in finding annihilators. *to appears in SETA 2006*, 2006.

[DMS05] Deepak Kumar Dalai, Subhamoy Maitra, and Sumanta Sarkar. Basic theory in construction of Boolean functions with maximum possible annihilator immunity. 2005. http://eprint.iacr.org/2005/229/.

[DT06] Frédéric Didier and Jean-Pierre Tillich. Computing the algebraic immunity efficiently. *Fast Software Encryption, FSE*, 2006.

[FA03] J.-C. Faugère and G. Ars. An algebraic cryptanalysis of nonlinear filter generator using Gröbner bases. *Rapport de Recherche INRIA*, 4739, 2003.

[HR04] P. Hawkes and G. C. Rose. Rewriting varaibles: The complexity of fast algebraic attacks on stream ciphers. In *Advances in Cryptology-CRYPTO 2004*, volume 3152 of *LNCS*, pages 390–406. Springer Verlag, 2004.

[Mas69] J. L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theory*, IT-15:122–127, 1969.

[MPC04] Willi Meier, Enes Pasalic, and Claude Carlet. Algebraic attacks and decomposition of Boolean functions. *LNCS*, 3027:474–491, April 2004.

[Odl84] Andrew M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Theory and Application of Cryptographic Techniques*, pages 224–314, 1984.

[Wie86] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trand. on Inform. theory*, IT-32:54–62, January 1986.