Computation of Discrete Logarithms in $\mathbb{F}_{2^{607}}$

Emmanuel Thomé

Laboratoire d'Informatique (LIX)
École polytechnique
91128 Palaiseau Cedex
FRANCE
Emmanuel.Thome@polytechnique.fr

Abstract. We describe in this article how we have been able to extend the record for computations of discrete logarithms in characteristic 2 from the previous record over $\mathbb{F}_{2^{503}}$ to a newer mark of $\mathbb{F}_{2^{607}}$, using Coppersmith's algorithm. This has been made possible by several practical improvements to the algorithm. Although the computations have been carried out on fairly standard hardware, our opinion is that we are nearing the current limits of the manageable sizes for this algorithm, and that going substantially further will require deeper improvements to the method.

1 Introduction

Among the most common paradigms upon which public key cryptographic schemes rely are the difficulty of the factorization of large integers (for the RSA cryptosystem), and the difficulty of computing discrete logarithms in appropriate groups (for the Diffie-Hellman key exchange protocol [14], ElGamal cryptosystem [16], and ElGamal and Schnorr [38] signature schemes). Appropriate groups for discrete logarithm cryptosystems are multiplicative groups of finite fields, the group of points of elliptic curves [26,33], and also the jacobians of curves of higher genus [27,4,18]. The level of security reached by the use of these different groups varies a lot. Both the factorization of large numbers [29] and the computation of discrete logarithms in finite fields [11,19,3] can be addressed in subexponential time. This in turn has implications on the security of some elliptic curves cryptosystems, where the discrete logarithm problem on the curve reduces to the discrete logarithm problem on (an extension of) the curve's definition field [32,17]. This applies in particular to supersingular elliptic curves, where the MOV reduction [32] makes the discrete logarithm problem subexponential.

This being said, the existence of a subexponential attack does not automatically rule out a cryptosystem. A thorough account on which computations a cryptanalist can do with the current technology is necessary. While a tremendous amount of work (and CPU time) has been put towards the factorization of larger and larger numbers (S. Cavallar et al. used the Number Field Sieve to factor numbers as big as 512 bits [6,9], and even up to 774 bits numbers of a special form [7]), the computation of discrete logarithms in finite fields does

C. Boyd (Ed.): ASIACRYPT 2001, LNCS 2248, pp. 107–124, 2001.

not seem to looked at so frequently. For prime fields, a recent work by Joux and Lercier [22] computed logarithms in $\mathbb{F}_p$ with p having 120 decimal digits, i.e. 399 bits. For fields of characteristic 2, Gordon and McCurley [20] almost* computed logarithms in $\mathbb{F}_{2^{503}}$, but that was back in 1993. This makes it hard, today, to make a reasonable guess on how difficult a characteristic 2 finite field discrete logarithm problem actually is. Subsequently, when the discrete logarithm on an elliptic curve reduces to some finite field of characteristic 2, it is not easy to tell how big this field should be for the cryptosystem to be secure.

In this context, our goal was to investigate how far we could go today in computing discrete logarithms in $\mathbb{F}_{2^n}$. The fastest algorithm for this purpose is due to Coppersmith [11] and has complexity $O(\exp((c+o(1))n^{\frac{1}{3}}(\log n)^{\frac{2}{3}}))$, for a small constant $c \approx 1.4$. This complexity makes it comparable to the Number Field Sieve [29], when addressing the factorization of an n-bit number. The 503-bit discrete logarithm record of Gordon and McCurley [20] was done using massively parallel supercomputers at Sandia National Laboratories. As far as we know, no recent state-of-the-art computations have been achieved. For our computations, we used *standard* hardware: the typical computers we used were much like everybody's desktop PC. Nonetheless, we have been able to carry the record to a few digits higher than before by computing discrete logarithms in $\mathbb{F}_{2^{607}}$.

Section 2 of this article outlines Coppersmith's algorithm. Section 3 reviews the rationales that drive the choice of each individual parameter in the algorithm. Sections 4 to 8 detail how we addressed the difficulties showing up in several parts of the algorithm. Section 9 shows the technical data on how the computations went along.

At the time of this writing, the computations over $\mathbb{F}_{2^{607}}$ are not finished. The sieving part is completed, and the linear algebra is underway. The computation of the solution to the linear system is expected to be finished by the beginning of the autumn 2001. As a very last-minute news, Joux and Lercier [23] appear to have computed logarithms in $\mathbb{F}_{2^{521}}$, using the general function field sieve approach [2]. This approach is fairly different from the one adopted here, and is not addressed in this paper. However, the result presented by [23] is highly encouraging.

2 Coppersmith's Algorithm

Throughout this article, we will let K denote the field $\mathbb{F}_{2^n}$, which will be represented as the quotient $\mathbb{F}_2[X]/(f(X))$, where f is a monic irreducible polynomial of degree n over $\mathbb{F}_2$. We will often talk of the elements of K merely as polynomials. It will be understood that what we actually mean is a class of polynomials inside this quotient. Likewise, the *degree* of a non-zero element of K will be the minimum degree of the polynomials representing it (always between 0 and $n-1$).

* The computations had not been fully carried out, since the resulting linear system was never solved

It will sometimes be convenient to write f as $X^n + f_1$, where f_1 is a polynomial. For the purposes of the algorithm, f_1 will be chosen so as to have the smallest possible degree. It is believed, but not proven, that such an f_1 exists whenever we allow its degree to grow as $O(\log n)$.

Coppersmith's algorithm belongs to the family of *index-calculus* algorithms. This means that we first select a factor base $\mathcal{B}$, and aim at computing the logarithms of its elements. For this, we gather a collection of relations among them. The relations will be of the form $\prod_{i=1}^{l} \pi_i^{e_i} = 1$, where the π_i's are the elements of the factor base. For reasons that will become clear later, this is referred to as the sieving part. This part can easily be distributed. Once we have enough relations involving the elements of the factor base, we obtain their logarithms as the solution of a (usually huge) linear system (we take the log of each relation). This is the linear algebra part. Implementations can be done efficiently on multiprocessor shared-memory machines, but such computers are expensive. Distribution of the computation across a network of not-so-expensive computers is very hard. The knowledge of all these logarithms, if the factor base is big enough, enables us to compute any logarithm in K easily. We will not detail that third part here since it is far easier than the two others. The interested reader might consult Coppersmith's original article [11] for reference.

The factor base $\mathcal{B}$ consists of all irreducible polynomials with degree less than a chosen bound b. It is known that $\mathcal{B}$ has roughly $\frac{2^{b+1}}{b}$ elements (see for instance [31]). Up to now, Coppersmith's algorithm is very resemblant to Adleman's [1, 5,3], which computes discrete logarithms in any Galois field, no matter the characteristic (but with poorer complexity than Coppersmith's). The key difference is in the production of linear relations. To build relations among the elements of $\mathcal{B}$, we choose random relatively prime polynomials A and B of degrees d_A and d_B, respectively. Let k be a power of 2 near $\sqrt{n/d_A}$, and $h = \lceil \frac{n}{k} \rceil$. Then we write:

$$\begin{aligned} C &= AX^h + B, \\ D = C^k &= A^k X^{hk} + B^k \equiv A^k X^{hk-n} f_1 + B^k \ [f]. \end{aligned}$$

An appropriate choice of the parameters keeps the degrees of C and D balanced, around $\sqrt{nd_A}$. For each such produced pair, we want to know whether it is *smooth* or not. The pair (C, D) is *smooth* when both polynomials have their irreducible factors inside $\mathcal{B}$. Of course, the bigger the factor base, the more likely this is. A smooth pair will give us a linear relation among the logarithms of the elements of $\mathcal{B}$, since if we denote them $\pi_i,\ 1 \leq i \leq \#\mathcal{B}$, we can find integers α_i and β_i such that:

$$\begin{aligned} C = \prod_i \pi_i^{\alpha_i}, D = \prod_i \pi_i^{\beta_i}, \Rightarrow DC^{-k} &= \prod_i \pi_i^{\beta_i - k\alpha_i} = 1, \\ \Rightarrow \sum_i (\beta_i - k\alpha_i) \log \pi_i &\equiv 0 \ [2^n - 1] \end{aligned}$$

Once we have gathered enough relations, we are facing a (fairly big) linear system that has to be solved, the unknowns being the logarithms of the elements of $\mathcal{B}$.

3 Choice of the Parameters

Coppersmith's algorithm introduces many parameters that may seem arbitrary at first glance. In [11], Coppersmith computed the asymptotical optimum value for each of them. We will not redo this analysis here, but rather briefly discuss the practical importance of each of the parameters, especially taking care of implementation realities like available hardware.

The choice of b. This main parameter, whose asymptotical optimum value is $n^{1/3}(\log n)^{2/3}$ controls the ratio between the work amounts in the first and second stages. The bigger b, the easier the first stage (even if we have twice as many relations to produce, the probabilities of smoothness increase drastically with b). On the other hand, increasing b by 1 almost doubles the size of the linear system in the second stage. Since the linear algebra is hardly distributable, the available hardware enforces a strong limit on the size of this system (otherwise the matrix would not fit into memory).

The choice of d_A and d_B. Originally, Coppersmith grouped them as a single parameter chosen asymptotically "near b"[⋆]. These parameters account for the number of pairs to test. Taking into account the probability of smoothness, we have to make sure that the $2^{d_A+d_B+1}$ available coprime (A, B) pairs will be enough to produce the required number of relations among the elements of $\mathcal{B}$. Of course, the sad news is that increasing d_A and d_B raises the degrees of C and D, and hence lowers the probability of smoothness. We have split Coppersmith's single parameter in two because it is usually possible to choose d_B a little bit above d_A without increasing the degrees of C and D (the optimum difference between the two is $\frac{hk-n+\deg f_1}{k}$). Therefore, we can maximize the number of pairs which are available.

The choice of k. Ideally, C and D have almost the same degree, their optimal value being $n^{2/3}(\log n)^{1/3}$. In fact, these can be somewhat unbalanced from the practical point of view. The parameter k is there to keep these polynomials in the same range, but unfortunately the requirement that k be a power of 2 gives us little control over it. The asymptotical best value for k is $\sqrt{\frac{n}{d_A}} = \left(\frac{n}{\log n}\right)^{\frac{1}{3}}$. For the problems we are concerned about, $k = 4$ appeared to be the correct choice. It might be that, at $n = 607$, we are nearing the cross-over point between $k = 4$ and $k = 8$, but $k = 8$ is still inadequate. One other aspect about the choice of k is that half of the coefficients in the linear system are $-k$ (the other

[⋆] An asymptotic ratio is computed in [11], depending on the algorithm used for linear algebra

ones being 1's). This brings a complication to the linear algebra (the structured gaussian elimination, namely), which could only be worsened by the choice of a bigger k.

The choice of f_1. Another hidden parameter lies in the choice of f_1. Usually, one can choose among a couple of candidates for f_1. The ones of low degree have a clear advantage due to the influence on $\deg D$, but [20] shows that polynomials with small factors are also worth investigating. The reader is referred to [20] for a thorough discussion on the choice of f_1.

In our computations, for $n = 607$, the following parameters were chosen: $b = 23$ (hence $\#\mathcal{B} = 766,150$), $d_A = 21, d_B = 28, k = 4, h = 152$. As for the choice of f_1, it turned out that $X^9+X^7+X^6+X^3+X+1$ had an overwhelming advantage, being simultaneously the candidate of smallest degree and with only small factors: f_1 factorizes as $(X+1)^2(X^2+X+1)^2(X^3+X+1)$. Given these parameters, the respective degrees of C and D were 173 and 112.

4 Description of the Polynomial Sieve

In Coppersmith's original version of the algorithm, the smooth pairs were located by repeatedly applying a smoothness test to all pairs of the allowed range. Gordon and McCurley [20], as an alternative, designed an efficient polynomial sieve, which helped to reduce the time spent on each pair (smooth or not). The idea is as follows. For A fixed, we maintain a big array of integers (initially 0) associated to the different pairs to be tested, that is, all the possible B's. Let g be an irreducible polynomial. We want to add $\deg g$ to the values associated to the B's* satisfying:

$$B \equiv AX^h \ [g]. \tag{E}$$

Doing this sieve efficiently implies being able to step quickly through all multiples of g. This can be done without awkward polynomial multiplications using *Gray codes*. For any non-zero positive integer x, let $l(x)$ denote the index of the least significant bit set in the binary representation of x (starting at $l(1) = 0, l(2) = 1$). Then the congruence class of $AX^h \mod g$ among the polynomials of degree less than or equal to d_B is given by the set of values of the sequence defined by: $B_0 = AX^h \mod g$, $B_i = B_{i-1} + X^{l(i)}g$, for $0 < i < 2^{1+d_B-d_g}$. Of course, it is worthwhile to precompute the $X^j g$'s, since these differ from each other only by arithmetic shifts.

This sieve is done for a certain collection of irreducible polynomials. One can also take into account the contribution of powers of irreducible polynomials, adding $\deg g$ to all $B's$ satisfying $B \equiv AX^h \ [g^j]$. If the sieve is done for all irreducible polynomials g, and also their powers, the value in each table cell is precisely the degree of the smooth part in the factorization of the associated quantity $C = AX^h + B$ (an entry for which the congruence holds modulo

* Or, equivalently, the *pairs*, since A remains fixed.

g^j accumulates a total contribution of $j \deg g$ from the consecutive sieves with $g, g^2, \ldots, g^j$). Therefore, an entry in the table which has a value of $\deg C$ automatically corresponds to a pair with C smooth.

In real life, one does not use all the relevant irreducible polynomials for the sieve, and an important improvement comes from the use of incomplete sieves. Two parts of the sieve are actually very expensive: the sieve over small irreducibles on the one hand, because there are many cells to update for each small irreducible, and the sieve over big irreducibles on the other hand because the initialization cost is high (and the number of irreducibles of a given degree raises with the degree). Therefore, we considered skipping these parts. Doing so, we lose accuracy, because the smoothness of C is only evaluated from the contribution of medium-size polynomials. Instead of $\deg C$, we use as *qualification bound* the average contribution from medium-size polynomials to a smooth C. If the standard deviation of this quantity is high, it will be hard to recognize pairs yielding smooth C's among the set of all pairs to be considered. Since the subsequent distinction between useful and useless pairs is done on a per-pair basis (a factorization job, in fact) their number should not grow too much. We found interest in skipping the sieve over irreducibles of degree 1 to 9, because their total contribution to smooth polynomials did not deviate too much from its average value, whereas we only skipped high degree 23, because otherwise we would have had to lower drastically the qualification bound to catch sufficiently many of the pairs yielding a smooth C, which in turn would have made the factorization cost too high.

Based on the same ideas, it is not always worthwhile to sieve over powers of an irreducible polynomial. Locating cells corresponding to pairs divisible by g^j for an irreducible polynomial g and an integer i is practically pointless if the expected number of cells to update is too small (this number is $2^{d_B+1-j \deg g}$). In fact, the only powers that we found worthwhile to sieve with were squares of polynomials of degrees 10 and 11.

It could be tempting to try to also do a sieve with D, but the situation is quite different. The initialization of the sieve must be done with $B_0 = A(X^{hk-n} f_1)^{1/k} \bmod g$, for g an irreducible polynomial. This computation is more complicated than previously. Also, this only works when g is an irreducible polynomial, and not when it is a power of an irreducible, because a k-th root might not exist modulo g^j. This difficulty is due to the same particularity of D that Gordon and McCurley already noticed in [20]: this polynomial is more likely to be square-free than it would be if it were random (and therefore it is less likely to be smooth). As we have just seen, this last point is not too disturbing since one hardly uses powers of irreducibles for the sieve.

Sieving over D turned out not to be useful in our case, since the first sieve (over C) already eliminated most of the pairs, and eventually testing the smoothness of D on a per-pair basis was more efficient. Nonetheless, sieving over D only instead of C could be useful in different settings, depending on how $\deg C$ and $\deg D$ compare to each other. In $\mathbb{F}_{2^{607}}$, the parameter k seems to be better around 4, and as a consequence, the degrees of C and D are not really balanced:

$\deg C$ is much higher. If we were about to carry out computations in, say, $\mathbb{F}_{2^{997}}$, $k = 8$ would probably be a better choice. And $\deg D$ would become automatically bigger than $\deg C$. A sieve over D in this situation would therefore enable us to discard much more pairs than its counterpart (because there would be very few smooth D's), and the benefit in the factorization part would probably compensate for the sieve's relative drawback.

5 Using Large Primes

One well-known improvement to the sieving part of index calculus algorithms is the so-called *large prime variation*. The idea is that aside plain, *full* relations, we allow *partial* relations, corresponding to pairs which are smooth up to a certain number of big irreducible cofactors (above the factor base bound) called *large primes*. Afterwards, these partial relations are matched together when this is possible in order to eliminate the cofactors. The partial relations come almost for free in the sieving stage, since they would otherwise have been discarded at the end of the factorization stage and not earlier. The degree of large primes must of course be kept under a certain bound: allowing for too large "large primes" eventually brings no benefit. From our point of view, this approach fits well here. We merely have to lower the qualification bound from $\deg C$ to $\deg C - \mathcal{L}$, where $\mathcal{L}$ is the maximum allowed degree of large primes.

When we allow only one *large prime*, matching partial relations together involves only a hashing process in order to be able to spot partial relations containing an already met large prime. The number of full relations reconstructed this way grows quadratically vs. the number of partial relations. When up to two large primes are used (see [30]), an algorithm resembling "union-find" helps to find *cycles*: relation after relation, we build a graph whose vertices are the large primes. An edge connects two vertices if a partial relations exists involving them. There is also a special vertex named "1", to which all primes involved alone in a partial relation are connected. Under certain conditions*, a cycle in this graph will give us a free full relation. The overhead is small, but this cycle detection has to be implemented with care because managing a graph with more than 10^8 edges among 2.10^9 vertices can turn out to be quite awkward. More elaborate schemes allow the processing of partial relations with more large primes, see for instance [15]. Recently, in the course of the record-breaking factorization of RSA-155, S. Cavallar proposed in [8] an efficient scheme for this large prime matching task, inspired by *structured gaussian elimination* like in [37]. We lacked the required time to investigate the respective efficiency of all of these different strategies when applied to our case. This is a real concern here, because while the multi-large-prime schemes have proven to be very efficient in the factorization context, this is not completely clear for discrete logarithms. Factorization algorithms use relations that are defined up to squares, that is, with exponents defined over $\mathbb{F}_2$. For discrete logarithms, exponents are defined in a big finite ring, here

* Slight complications are brought by the fact that our coefficients are not defined over $\mathbb{F}_2$.

$\mathbb{Z}/(2^n - 1)\mathbb{Z}$. When combining partial relations with large primes in common, one can only cancel one large prime at a time. For this reason, the landscape is quite different.

Our computations have been carried out using the *double* large prime variation, that works well even with regard to the coefficient issue. Two large primes were allowed. For efficiency reasons (discussed in section 7), only the factorization of C could have two large primes, while D was restricted to only a single large prime. 10% of the relations had actually only one large prime, and among the remaining relations (that had two large primes), 30% had both large primes on the same side (the C side, actually), the rest of the relations having their large primes balanced on each side. We did the cycle detection using a straightforward union-find algorithm. Figures about the cycle detection can be found in section 9.

6 Grouping Sieves

As it is described above, the sieve algorithm uses an array of fixed size, namely 2^{d_B+1} bytes (assuming one byte per sieve location). Our setup had $d_B = 28$, so this makes a sieve area of 512MB, far above what is acceptable. Furthermore, it was not certain by the beginning of the sieve whether the outcome of pairs with a polynomial B of big degree would eventually be used or not. We decided to have a first look at the pairs from which we knew that the outcome would be better, that is, the pairs with smaller B's, and defer the analysis of less promising pairs to a later time. Our strategy was to decompose the whole sieving job in *chunks* indexed by fixed parts A_f and B_f of the polynomials A and B. The chunks consisted of areas of the form:

$$\begin{aligned}\text{chunk}(A_f, B_f) = &\{(A, B) = (A_f X^{\delta_A+1} + A_v, B_f X^{\delta_B+1} + B_v),\\ &\deg A_v \le \delta_A, \deg B_v \le \delta_B\}, \text{with } \delta_A = 6, \delta_B = 24.\end{aligned}$$

Each chunk could be sieved by the machine handling it in any suitable way. The most straightforward approach is to do $2^7 = 128$ sieves, each of them addressing 2^{25} bytes, that is 32MB, for the sieve area.

Since we ran the job using idle time on many not-so-powerful machines, this was still too much memory to be used for some of them. A further possibility is to divide the 32MB sieve area into yet more (say 2^γ, with γ a small integer), smaller sieve areas (of size $2^{-\gamma} \times$ 32MB). But when the sieve area becomes so small, the initialization cost becomes too important. The expensive task is the modular reduction $AX^h \mod g$, which is performed for each g. One can precompute the initialization data for the 2^γ sieves, but even after doing that, we were unsatisfied with the cost of the initialization, and tried to trim it down even more.

We wanted to achieve this without letting additional bits of B vary, but rather sieving over several A's at a time. This is possible because for reasonably close A's, the initialization for a given g is almost the same. In the following

paragraphs, g will denote either an irreducible polynomial, or a power of an irreducible polynomial. Inside a given sieve with A completely fixed, we want to find the solutions to:

$$B_f X^{\delta_B+1} + B_v \equiv AX^h \ [g].$$

If we allow some of the lowest bits of A, say ϵ of them (with $\epsilon \leq \delta_A + 1$, of course) to vary, the equation becomes:

$$B_v + \alpha X^h \equiv AX^h + B_f X^{\delta_B+1} \ [g], \text{ with } \deg \alpha < \epsilon. \tag{E'}$$

The solutions to this equation form an affine subspace $\mathbb{S}$ of the $\mathbb{F}_2$-vector space $\mathbb{V} = F \oplus G$, with $F = \langle 1, X, X^2, \dots X^{\delta_B} \rangle$ and $G = \langle X^h, \dots X^{h+\epsilon-1} \rangle$. The expected dimension of $\mathbb{S}$ is $\dim \mathbb{S} = \delta_B + 1 + \epsilon - d_g$. We will try to find $\mathbb{S}$ using linear algebra over $\mathbb{F}_2$. The idea behind this is that arithmetic shifts and logical operations take almost no time compared to a polynomial division or multiplication. We will consider two situations.

The easy case is when $d_g \leq \delta_B + 1$. $\mathbb{S}$ writes down as $s_0 + \mathbb{S}'$, with a point $s_0 = AX^h + B_f X^{\delta_B+1} \mod g$, and an underlying vector space $\mathbb{S}'$ spanned by the $X^i g$ for $0 \leq i \leq \delta_B - d_g$, and the $X^{h+i} + (X^{h+i} \mod g)$ for $0 \leq i < \epsilon$. We claim that the computation of these generators costs very little above 2 modular reductions since once $X^h \mod g$ has been computed, inferring the $X^{h+i} \mod g$ inductively is easy (one bit test and one exclusive-or if needed). If we did independent sieves we would have needed 2^ϵ modular reductions (which can be anything but cheaper).

If $d_g > \delta_B + 1$, we extend $\mathbb{V}$ to $\bar{\mathbb{V}} = \bar{F} \oplus G$, $\bar{F} = F \oplus \langle X^{\delta_B+1}, \dots X^{d_g-1} \rangle$. Let $\bar{\mathbb{S}}$ be the set of solutions of E' in $\bar{\mathbb{V}}$. A point $\bar{s}_0$ in $\bar{\mathbb{S}}$ is obtained as in the previous case, and generators of the underlying vector space $\bar{\mathbb{S}}'$ are the $u + \phi(u)$ for $u \in G$, ϕ being the linear map from G to $\bar{F}$ that reduces a polynomial mod g. Using gaussian elimination, we can find a point $s_0 \in \mathbb{S}$ deduced from $\bar{s}_0$ and $\bar{\mathbb{S}}'$ if such a point exists, and the generators of $\mathbb{S}'$ (the vector space underlying $\mathbb{S}$) are the $u + \phi(u)$ for $u \in \phi^{-1}(F)$. This involves finding the kernel of a $(\dim \bar{F} - \dim F) \times \epsilon$ matrix, which is expected to be quite easy (perhaps a dozen CPU cycles). Although the case where $d_g > \delta_B + 1$ is unlikely to be met often in practice (we don't want to sieve when $\deg g$ is too big), we will augment this quick description with an example. Suppose we have the following setup:

$$\begin{aligned} g &= (X^{14} + X^{13} + X^{12} + X^{10} + X^8 + X^5 + 1)^2, \\ \bar{s}_0 &= X^{27} + X^{26} + X^3 + 1, \\ h &= 152, \quad \delta_B = 24, \quad \epsilon = 3. \end{aligned}$$

The first three columns of the following matrix are the $\dim \bar{F} - \dim F$ most significant coefficients of the polynomials $u + \phi(u)$ for $u = X^{h+i}$ and $0 \leq i < \epsilon$. The last one contains the leading coefficients of $\bar{s}_0$:

$$T = \begin{pmatrix} 0\,1\,0\,1 \\ 1\,0\,1\,1 \\ 0\,0\,0\,0 \end{pmatrix}$$

Doing a gaussian elimination on the columns of T, one easily obtains:

$$s_0 = \bar{s}_0 + X^h + \phi(X^h) + X^{h+1} + \phi(X^{h+1}) \in \mathbb{S},$$
$$\phi^{-1}(F) = \langle X^h + X^{h+2} \rangle.$$

Of course, this example is a bit particular in that the last row of T is zero, therefore the dimension of $\mathbb{S}'$ is one, instead of the expected value 0. Other cases can occur: for instance, if $\bar{s}_0$ had had a non-zero coefficient in X^{25}, then we would have had no solutions to the equation E' inside $\mathbb{V}$.

We have shown two ways to play with the memory available to the siever. These can actually be mixed together. Using parameters γ and ϵ together, a chunk is divided in $2^{\delta_A+1+\gamma-\epsilon}$ sieves, each of them using $2^{\delta_B+1+\epsilon-\gamma}$ bytes of memory. The influence of the two parameters γ and ϵ is shown on figure 1. Timings are in seconds runtime on 450MHz Pentium II's. The percentages show the timing difference versus the standard sieve (which has $\gamma = \epsilon = 0$). Three figures are present in each table cell. The figures are always normalized to reflect the time needed to sieve a (fictitious) 128MB sieve area. The first one, on which the effect of both γ and ϵ is the most striking, shows the time spent in initializing the sieve (or, in re-reading again and again the precomputed initialization data when $\gamma > 0$. Precomputation time in this case is also included). The second figure is the time spent in the sieve itself, that is, adding $\deg g$ to each table cell corresponding to a pair divisible by g, for all possible g's. The effects of γ and ϵ on this sieving time are hardly noticeable (the variations are likely to be due to operating system overhead). The third figure is the total time spent including allocation overhead and final pair detection (but not the factorization, which comes afterwards, and is irrelevant here). On the right and the bottom, the actual memory sizes used by the sieve area are given. Since our jobs have been running in background on otherwise used machines, we preferred not to use too much memory. Using the setting $\gamma = 4,\ \epsilon = 3$ was a satisfying compromise, with a mere 16MB sieving area.

7 Factorization of the Pairs

Once good pairs have been located, the actual production of the relations (or partial relations) requires the factorization of the pairs (C, D). Efficient algorithms exist for polynomial factorization, but our actual problem here is not the usual one. Instead of the factorization of one huge polynomial (of degree several thousands for instance), we have to deal with the factorization of a huge number of relatively small polynomials (in our case, their degree is less than 200). Therefore, asymptotically better behaving algorithms might not be worthwhile. Furthermore, we are willing to give up as soon as we suspect the polynomial might not be smooth after all. In a few words, merely applying some classical distinct degree factorization algorithm can turn out to be a considerable waste of time. We built a factorization scheme based on several specific improvements that turned out to be worthwhile.

	$\epsilon = 0$	$\epsilon = 1$	$\epsilon = 2$	$\epsilon = 3$	
$\gamma = 0$	33.28 (0%) 105.68 (0%) 144.24 (0%)	22.76 (-31%) 109.96 (+4%) 138.20 (-4%)	12.26 (-63%) 107.75 (+1%) 125.35 (-13%)	7.01 (-78%) 110.62 (+4%) 122.98 (-14%)	
$\gamma = 1$	38.84 (+16%) 109.48 (+3%) 153.80 (+6%)	24.46 (-26%) 108.94 (+3%) 138.88 (-3%)	12.94 (-61%) 110.84 (+4%) 129.29 (-10%)	6.89 (-79%) 107.73 (+1%) 119.95 (-16%)	256MB
$\gamma = 2$	46.16 (+38%) 107.12 (+1%) 158.60 (+9%)	28.66 (-13%) 109.38 (+3%) 143.50 (0%)	14.87 (-55%) 105.92 (0%) 126.12 (-12%)	7.99 (-75%) 106.29 (0%) 119.63 (-17%)	128MB
$\gamma = 3$	63.04 (+89%) 108.92 (+3%) 179.08 (+24%)	35.22 (+5%) 109.98 (+4%) 152.66 (+5%)	19.37 (-41%) 109.46 (+3%) 134.31 (-6%)	10.18 (-69%) 105.58 (0%) 121.11 (-16%)	64MB
$\gamma = 4$	96.56 (+190%) 108.68 (+2%) 210.72 (+46%)	54.56 (+63%) 111.26 (+5%) 171.28 (+18%)	28.27 (-15%) 110.42 (+4%) 144.14 (0%)	14.69 (-55%) 106.84 (+1%) 126.87 (-12%)	32MB
	2MB	4MB	8MB	16MB	

Fig. 1. Influence of γ and ϵ on the sieving time.

The pairs that constitute the input to the factorization step are such that C has a reasonable probability to be smooth: it has been selected for this purpose. D, however, has no reason to be smooth. Therefore, the first thing to try out is a smoothness test on D, in order to avoid useless computations on all pairs with non-smooth D's. The smoothness test applied is the same as in [11], except that we want to allow large primes. The b-smooth part of D is computed as

$$D_{\text{smooth}} = \gcd\left(D, D' \prod_{j=1+\lfloor \frac{b}{2} \rfloor}^{b} (X^{2^j} + X) \mod D\right).$$

In some cases (if D has a very big square factor), D_{smooth} might not actually be b-smooth, but that's exceptional. Concerning the cofactor $\frac{D}{D_{\text{smooth}}}$, we are facing a design choice, since we can either allow only one large prime, that is, allow a cofactor of degree at most $\mathcal{L}$, or permit several large primes, setting for instance the cofactor bound to the looser $2\mathcal{L}$. However, in the latter case, the cofactor needs not factor kindly into two large primes of degree less than $\mathcal{L}$ (actually, it is most likely not to). The best choice depends on what we want to do with partial relations. In our experiments, less than 1% of the D's passed the former test, while around 12% passed the latter (hence there were more pairs to be factorized afterwards, resulting in a 25% increase of the factorization cost). Since we used

only two large primes in total, the yield was better when using the first, more restrictive test. If we were to allow three large primes or more (cf [15]), the second, looser, test would probably be more adequate.

Once D has passed the test, and has therefore an acceptable probability to be smooth (save the cofactors), we need to factorize C and D. Originally, when running our program on smaller examples like $\mathbb{F}_{2^{313}}$, we found it useful to track down small factors either by explicit trial division or by precomputations and table lookup. The idea is to quickly compute the valuation of a given polynomial with respect to some irreducible. Of course, this is trivial for the valuation with respect to X. Let us explain briefly how this can be done for the valuation with respect to $X+1$. We notice that $(X+1)^{16} = X^{16}+1$. Since our implementation represents the polynomials over $\mathbb{F}_2$ using one bit per coefficient, computing a remainder modulo $X^{16}+1$ is fast. Assuming we have a 32-bit machine, this requires less than $\frac{\deg P}{32} + 3$ operations (exclusive "OR"s, one shift and one "AND"). If we have a precomputed table holding the values of $\nu(Q)$ (ν is the valuation) for all polynomials Q of degree 0 to 15 (this requires 32KB), we can obtain $\nu_g(P)$ with high probability. Indeed, we have $\nu_g(P \mod X^{16}+1) = \nu_g(P)$ unless $P \equiv 0\ [X^{16}+1]$ (in which case we have an inequality $\leq$). Once we have this value, we merely have to do one division by the appropriate (precomputed) power of $X+1$. If the valuation is at least 16, we repeat the operation on the cofactor. From the basic observation that a remainder modulo a cyclotomic polynomial is easily computable, we could extend this approach for irreducibles of degree up to 4. Alas, the improvement obtained from this method was not significant for the case of $n = 607$, probably because the average degree of C and D made the contribution of little factors too small for this to be useful. We also tried to factor the relations by sieving with all or part of the possible irreducibles that could appear in the factorization, but this brought no significant improvement either.

Since it turned out that our attempts towards removing some of the factors by hand were not worthwhile, the whole factorization job was achieved by a general-purpose factorization algorithm (in any case, if we did remove some of the factors by trial division, the cofactor would have still had to be factorized via such an algorithm). We used Niederreiter's algorithm [35], which proved four times faster than a classical distinct degree factorization procedure. The explanation of this lies of course in the small degree of our polynomials, and in the fact that we work over $\mathbb{F}_2$, for which Niederreiter's algorithm is well suited.

8 Improvements to the Linear Algebra Stage

The sparse matrix emerging from the sieving has roughly $\frac{2^{b+1}}{b}$ columns, and a bigger number of lines (we had a 40% excess). This matrix is extremely sparse: the number of non-zero terms (called the weight) of a given line corresponding to a smooth pair (C, D) is actually the number of distinct factors in the factorization of DC^{-k}. Most relations are also obtained from recombinations of partial relations, so the weight for a recombination of s relations is s times

the average number of factors in a factorization like DC^{-k}. In our case, this amounts to an average weight of the lines for the whole matrix of 67.7. Handling such systems requires well-suited algorithms, designed to take advantage of the sparsity as much as possible. Actually, this is a well studied subject, since sparse matrices arise in many domains. For the literature about sparse matrices coming from discrete logarithm or factorization problems, one can consult [37,43,12,34, 25,28]. Two particularly annoying points are relevant to our case. Unlike linear system that arise from factorization problems, ours is defined over a big field, $\mathbb{Z}/(2^{607}-1)\mathbb{Z}$. Second, unlike what happens with Adleman's algorithm [1], or with the number field sieve when applied to the discrete logarithm problem [19], our coefficients are not always ± 1. As explained earlier in this article, half of them are $\pm k$.

In order to solve our system, we first apply the well-known *structured gaussian elimination* as described in [37]. This algorithm takes advantage of both the sparsity of the matrix, and also of the "unbalanced" shape of its lines: each line in the matrix corresponds to a relation, and the coefficients on the left correspond to small factors, while those on the right correspond to big factors. The probability of a given polynomial to be divisible by a given factor of degree d being $\frac{1}{2^d}$, the density of the matrix is much higher on the left part (small factors) than on the right part (big factors). The structured gaussian elimination starts from the right end of the matrix (which is extremely sparse) and tries to remove lines and columns without increasing (if at all) the matrix density.

We modified the original process described in [37] in the spirit of what is done in [42]: we evaluate, at each step, the influence of each possible operation to the cost of the linear system solving algorithm that follows the SGE. The better steps towards the reduction of the linear algebra cost are taken, until nothing interesting can be done anymore. This process is able to shrink down the matrix to a fraction of its original size. Here, having many coefficients equal to $\pm k$ on input causes lines to be multiplied quite often while pivoting is done. Since a given line cannot be multiplied too many times (otherwise we would have to allow the coefficient to grow above one machine word), this makes the elimination less efficient.

Afterwards, we found it enlightening to use the *block Wiedemann* algorithm. This algorithm has been proposed by Coppersmith in [12], extending a previous algorithm by Wiedemann [43]. Another algorithm, the block Lanczos algorithm [34], is often preferred to the block Wiedemann algorithm. We used the latter because it gave us an opportunity to successfully experiment the accelerating procedure described in [39]: the crux of the block Wiedemann algorithm is the computation of a linear generator for a matrix sequence (a matrix analogue to the Berlekamp-Massey algorithm), and [39] uses FFT to reduce the complexity of this task from $O(N^2)$ to $O(N \log^2 N)$, achieving a 50 times speedup for the computation undertaken here. The block Wiedemann algorithm performs well both theoretically and in practice. See [40,41,24,25,39] for several insights on the algorithm. The block Wiedemann algorithm is interesting in the fact that at least for one part of the algorithm, several machines holding a private copy

of the matrix (for which they need to have the proper amount of memory) can each do a part of the work without communication between them. Therefore, one can regard this as a partial distribution. We found that the optimal number of machines to be used simultaneously in this computation was 4 (luckily, we had that number of machines able to hold the 400MB matrix in RAM).

9 Computations over $\mathbb{F}_{2^{607}}$

The comprehensive sieving part took about $19,000$ MIPS years. As a comparison, the factorization of RSA-155 required $8,000$ MIPS years. The outcome of the sieving processes, in terms of relations per hour, dropped from 1000 relations (full or partial) per hour with the very first chunks (the degrees were still small) to 400 afterwards, and eventually 100 for the very last ranges of data. Almost all the sieving area up to $d_B = 28$ has been needed (a more thorough usage of this area could have been achieved if we did not use incomplete sieves, but the trade off was clear in their favor). Of these relations, of course, most were partial ones. The total amount of data produced by these sub-processes nears 10GB. The cycle detection algorithm ran approximately for one day and produced the biggest part of the relations at the end: $815,726$ relations were reconstructed using cycles of length going from 2 to 40. All of these cycles were linked to the special vertex "1", which is not surprising given the size of the corresponding connected component. More than $650,000$ relations were obtained from cycles of length 3 or more, which shows that using the *double* large prime variation was a winning choice. Meanwhile, we only produced $217,867$ genuine full relations. Additional data can be found in table 1. The average weight of these $1,033,593$ relations in total was 67.7, the maximum weight being 524. We discarded the relations whose weight was above 120, since these were definitely too heavy to be useful. We were left with $904,004$ relations, involving $765,427$ columns (the average weight dropped to 64.3).

We ran a structured gaussian elimination algorithm (SGE) on this matrix. The schedule time for SGE was approximately one day. We were able to divide by two the cost of the subsequent block Wiedemann algorithm. The matrix obtained after the SGE had size $484,603 \times 484,603$ with an average line weight of 106.7. One can find this reduction ratio quite disappointing compared to ratios typically achieved in other contexts. This could be a consequence of the fact that most relations were recombined ones. These were therefore denser, and had coefficients somewhat bigger than other lines, which impairs the reduction ratio of the SGE.

The block Wiedemann algorithm is currently underway, in the process of finding an element of the kernel of this matrix. We expect it to be finished by the beginning of the autumn. It should be noted that since $2^{607} - 1$ is prime, the linear algebra task cannot be eased anyhow by the chinese remainder theorem.

Table 1. Data from computations in $\mathbb{F}_{2^{607}}$

Size of the factor base	766,150 polynomials
Total number of relations	1,033,593 relations
Full relations	217,867 relations
Cycles obtained	815,726 cycles
Partial relations used ($\leq$ 2 large primes)	60,128,419 relations
Large primes involved	85,944,405 polynomials
Relations with only one large prime	5,992,928 relations
Cycles of length 2	150,566 cycles
Cycles of length 3	142,031 cycles
Cycles of length 4	123,900 cycles
Cycles of length 5	101,865 cycles
Cycles of length 6 or more	297,364 cycles
Size of the biggest cycle	40 edges
Size of the biggest connected component	22,483,158 edges
Size of the second biggest connected component	167 edges
Number of connected components with 1 edge	22,025,908 components
Number of connected components with 2 edges	2,726,940 components
Number of connected components with 3 edges	848,691 components

10 Conclusion

Computation of discrete logarithms in $\mathbb{F}_{2^{607}}$ is now a matter of weeks (linear algebra is in its last phase). As was predicted by Gordon and McCurley in the conclusion of their article [20], this was far from an easy task, and the computation took enormous proportions. Today's supercomputers might achieve the work we did in quite a reasonable time, but going further will necessarily imply more advanced techniques, including, but probably not limited to, the use of four large primes (taking into account the remark on the coefficient issue in section 5). The conclusion of our computation is that one can not seriously claim that discrete logarithms in, say, $\mathbb{F}_{2^{997}}$, are within the reach of a computation of the type we have undertaken. A very well-funded institution (e.g. governmental) could perhaps go that far, but this is much likely to involve a tremendous (and highly expensive) computational effort. An implication of our work to how we should regard the security of an elliptic curve cryptosystem with a MOV reduction [32] of the discrete logarithm problem to the discrete logarithm problem in a field $\mathbb{F}_{2^n}$, is that if n is around $1,000$, attacking such a problem is very hard, and if n is around $1,200$, this size is twice above the computational mark that we have just set. Therefore, the security of such a cryptosystem in the latter case can be seen as no lower than the security of an RSA-1024 cryptosystem, given that RSA-512 schemes have been successfully attacked using computational means comparable to ours.

Acknowledgements. Our program has been written in `C`, using the `ZEN` computer algebra package [10] and the `GMP` package [21] for multiprecision integer arithmetic. CPU time has been (and is being) provided by several institutions. Three units at École polytechnique, Palaiseau, France, provided most of the sieving time: the student computer clusters, UMS MEDICIS, and LIX (computer science research group). On a smaller scale, CPU time has been used for the sieving at École normale supérieure, Paris, France, and also at the department of Mathematics of the University of Illinois at Chicago, while the author was visiting this institution in 1999/2000. Large prime matching has been entirely done at LIX. Linear algebra involved mostly resources from LIX, and also from UMS MEDICIS to the extent possible given the distribution constraints for this task. We would like to express our grateful thanks to all the users at these places, and to the IT staff who have always been very helpful.

A special thank goes to François Morain, who has been helping the author with questions and comments since the beginning of this work, and throughout the preparation and achievement of this record.

References

1. L. M. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. In *Proc. 20th FOCS*, pp. 55–60. IEEE, 1979.
2. L. M. Adleman. The function field sieve. In L. Adleman and M.-D. Huang, eds., *ANTS-I*, vol. 877 of *Lecture Notes in Comput. Sci.*, pp. 108–121. Springer-Verlag, 1994. Proc. 1st Algorithmic Number Theory Symposium, Cornell University, May 6–9, 1994.
3. L. M. Adleman and J. DeMarrais. A subexponential algorithm for discrete logarithms over all finite fields. *Math. Comp.*, 61(203):1–15, July 1993.
4. S. Arita. Algorithms for computations in Jacobians of C_{ab} curve and their application to discrete-log-based public key cryptosystems. In *Proceedings of Conference on The Mathematics of Public Key Cryptography, Toronto, June 12–17*, 1999.
5. I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone. Computing logarithms in finite fields of characteristic two. *SIAM J. Alg. Disc. Meth.*, 5(2):276–285, June 1984.
6. CABAL. Factorization of RSA-140 using the number field sieve. Available online at `ftp://ftp.cwi.nl/pub/herman/NFSrecords/RSA-140`, Feb. 1999.
7. CABAL. 233-digit SNFS factorization. Available online at `ftp://ftp.cwi.nl/pub/herman/SNFSrecords/SNFS-233`, Nov. 2000.
8. S. Cavallar. Strategies in filtering in the number field sieve. In W. Bosma, ed., *Proc. ANTS-IV*, vol. 1838 of *Lecture Notes in Comput. Sci.*, pp. 209–231. Springer-Verlag, 2000.
9. S. Cavallar et al. Factorization of a 512-bit RSA modulus. In B. Preneel, ed., *Proc. EUROCRYPT 2000*, vol. 1807 of *Lecture Notes in Comput. Sci.*, pp. 1–18. Springer-Verlag, 2000.
10. F. Chabaud and R. Lercier. `ZEN`, a toolbox for fast computation in finite extensions over finite rings. Homepage at `http://www.di.ens.fr/~zen`.
11. D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Trans. Inform. Theory*, IT–30(4):587–594, July 1984.

12. D. Coppersmith. Solving linear equations over GF(2) via block Wiedemann algorithm. *Math. Comp.*, 62(205):333–350, Jan. 1994.
13. T. Denny and V. Müller. On the reduction of composed relations from the number field sieve. In H. Cohen, ed., *Proc. ANTS-II*, vol. 1122 of *Lecture Notes in Comput. Sci.*, pp. 75–90. Springer-Verlag, 1996.
14. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT–22(6):644–654, Nov. 1976.
15. B. Dodson and A. K. Lenstra. NFS with four large primes: an explosive experiment. In D. Coppersmith, ed., *Proc. CRYPTO '95*, vol. 963 of *Lecture Notes in Comput. Sci.*, pp. 372–385. Springer-Verlag, 1995.
16. T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory*, IT–31(4):469–472, July 1985.
17. G. Frey and H.-G. Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Math. Comp.*, 62(206):865–874, Apr. 1994.
18. S. D. Galbraith, S. M. Paulus, and N. P. Smart. Arithmetic on superelliptic curves. To appear in *Mathematics of Computation*, 2001.
19. D. M. Gordon. Discrete logarithms in GF(p) using the number field sieve. *SIAM J. Discrete Math.*, 6(1):124–138, Feb. 1993.
20. D. M. Gordon and K. S. McCurley. Massively parallel computation of discrete logarithms. In E. F. Brickell, ed., *Proc. CRYPTO '92*, vol. 740 of *Lecture Notes in Comput. Sci.*, pp. 312–323. Springer-Verlag, 1993.
21. T. Granlund. `GMP`, the GNU multiple precision arithmetic library. Homepage at `http://www.swox.se/gmp`.
22. A. Joux and R. Lercier. Discrete logarithms in GF(p) (120 decimal digits). Email to the `NMBRTHRY` mailing list; available at `http://listserv.nodak.edu/archives/nmbrthry.html`, Apr. 2001.
23. A. Joux and R. Lercier. Discrete logarithms in GF(2^n) (521 bits). Email to the `NMBRTHRY` mailing list; available at `http://listserv.nodak.edu/archives/nmbrthry.html`, Sept. 2001.
24. E. Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comp.*, 64(210):777–806, July 1995.
25. E. Kaltofen and A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica*, 24:331–348, 1999.
26. N. Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48(177):203–209, Jan. 1987.
27. N. Koblitz. Hyperelliptic cryptosystems. *J. of Cryptology*, 1:139–150, 1989.
28. B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. In A. J. Menezes and S. A. Vanstone, eds., *Proc. CRYPTO '90*, vol. 537 of *Lecture Notes in Comput. Sci.*, pp. 109–133. Springer-Verlag, 1990.
29. A. K. Lenstra and H. W. Lenstra, Jr., eds. *The development of the number field sieve*, vol. 1554 of *Lecture Notes in Math.* Springer, 1993.
30. A. K. Lenstra and M. S. Manasse. Factoring with two large primes. *Math. Comp.*, 63(208):785–798, Oct. 1994.
31. R. Lidl and H. Niederreiter. *Finite fields.* Number 20 in Encyclopedia of mathematics and its applications. Addison–Wesley, Reading, MA, 1983.
32. A. Menezes, T. Okamoto, and S. A. Vanstone. Reducing elliptic curves logarithms to logarithms in a finite field. *IEEE Trans. Inform. Theory*, IT–39(5):1639–1646, Sept. 1993.
33. A. J. Menezes. *Elliptic curve public key cryptosystems.* Kluwer Academic Publishers, 1993.

34. P. L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In L. C. Guillou and J.-J. Quisquater, eds., *Proc. EUROCRYPT '95*, vol. 921 of *Lecture Notes in Comput. Sci.*, pp. 106–120, 1995.
35. H. Niederreiter. A new efficient factorization algorithm for polynomials over small finite fields. *Appl. Algebra Engrg. Comm. Comput.*, 4:81–87, 1993.
36. A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In T. Beth, N. Cot, and I. Ingemarsson, eds., *Proc. EUROCRYPT '84*, vol. 209 of *Lecture Notes in Comput. Sci.*, pp. 224–314. Springer-Verlag, 1985.
37. C. Pomerance and J. W. Smith. Reduction of huge, sparse matrices over finite fields via created catastrophes. *Experiment. Math.*, 1(2):89–94, 1992.
38. C. P. Schnorr. Efficient signature generation by smart cards. *J. of Cryptology*, 4(3):161–174, 1991.
39. E. Thomé. Fast computation of linear generators for matrix sequences and application to the block wiedemann algorithm. In B. Mourrain, ed., *Proc. ISSAC '2001*, pp. 323–331. ACM Press, 2001.
40. G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Research Report 975, LMC-IMAG, Grenoble, France, Apr. 1997.
41. G. Villard. Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. In W. W. Küchlin, ed., *Proc. ISSAC '97*, pp. 32–39. ACM Press, 1997.
42. D. Weber and T. Denny. The solution of McCurley's discrete log challenge. In H. Krawczyk, ed., *Proc. CRYPTO '98*, vol. 1462 of *Lecture Notes in Comput. Sci.*, pp. 458–471. Springer-Verlag, 1998.
43. D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inform. Theory*, IT–32(1):54–62, Jan. 1986.