On Improving Integer Factorization and Discrete Logarithm Computation using Partial Triangulation

Fabrice Boudot

fabrice.boudot@orange.fr

Abstract. The number field sieve is the best-known algorithm for factoring integers and solving the discrete logarithm problem in prime fields. In this paper, we present some new improvements to various steps of the number field sieve. We apply these improvements on the current 768-bit discrete logarithm record and show that we are able to perform the overall computing time in about 1260 core·years using these improvements instead of 2350 core·years using the best known parameters for this problem. Moreover, we show that the pre-computation phase for a 768-bit discrete logarithm problem, that allows for example to build a massive decryption tool of IPsec traffic protected by the Oakley group 1, was feasible in reasonable time using technologies available before the year 2000.

Keywords: discrete logarithm, integer factorization, number field sieve, sparse linear algebra

1 Introduction

1.1 The discrete logarithm problem

The hardness of the discrete logarithm problem in prime fields is one of the most used assumptions in asymmetric cryptography, alongside with the hardness of integer factorization and discrete logarithm computations on elliptic curves. The security of well-known cryptographic primitives, such as the Diffie-Hellman key exchange protocol [12], the El-Gamal encryption [13], and the Digital Signature Algorithm [14], are based on the discrete logarithm problem, and these primitives are used in the most used security protocols such as TLS, IPsec or SSH.

The discrete logarithm problem over prime fields can be defined as follows. Let p be a large prime number and let $q = p - 1$. Let g be an element of order q in $\mathbf{Z}_p$. Let y be an element of $\mathbf{Z}_p$. We have to find a number $x \in [0, q-1]$, named the discrete logarithm of y in base g modulo p, that is such that $y \equiv g^x \bmod p$.

When p is large enough, the best algorithm to solve the discrete logarithm problem is the number field sieve [29]. The asymptotic complexity of this algorithm, when p has no specific form, is $L_p(1/3, (64/9)^{1/3} + o(1))$, where L_p is defined by

$$L_p(\alpha, c) = \exp(c(\ln p)^{\alpha}(\ln \ln p)^{1-\alpha})$$

The most time consuming part of this algorithm consists in computing a database of small discrete logarithms that only depends on the modulus p. Once this database has been computed, the resolution of discrete logarithms modulo p requires comparatively a very small amount of computing power [1]. Many security protocols use a small set of predefined prime moduli: for example, the IKE protocol used to create cryptographic keys in the IPsec protocol uses only one common modulus for a given security size. It means that the computation of the database of discrete logarithms for such modulus and its public release would allow anyone, even with a very small computing power, to decrypt the IPsec traffic encrypted with this modulus.

1.2 Records for the discrete logarithm problem in prime fields

Until recently, the discrete logarithm record in prime fields was about 200 bits below the integer factorization record:

- In 2005, the discrete logarithm record was held by Joux and Lercier [18] who computed a 431-bit discrete logarithm, whereas Franke, Kleinjung *et al.* [3] have factored a 663-bit integer.
- A few years later and until 2014, Thorsten Kleinjung held both records: he sets the discrete logarithm record to 530 bits in 2007 [22], and joined an international team that factored a 768-bit RSA challenge in 2009 [25].

The best-known algorithms to factor large integers and to solve discrete logarithms are both called the number field sieve. The two algorithms have a lot of steps in common, and the main difference is that we have to solve a large system of equations modulo a large prime q to solve discrete logarithms, while we only have to solve this system modulo 2 to factor a large integer. This difference partly explains the 200-bit gap between the two records.

Another reason explains this gap: the scientific community has given much more attention to factorization records instead of discrete logarithm records, and the record breakers have used much more computing power for their factorization records that for discrete logarithms. For example, the 768-bit RSA record has been done using 1500 core·years of computing power taken from various sites during a two-year period, whereas the corresponding 530-bit discrete logarithm record was computed using only 17 core years.

This lack of interest for discrete logarithm records was in contradiction with the practical impact of records. If the factorization of an RSA challenge is an outstanding scientific achievement, it only allows an attacker with several million of dollars of hardware to reproduce such computation and break the RSA key of only one user. But if the pre-computation of the database of logarithms of a popular modulus is done and is publicly released, then anyone is able to decrypt the corresponding traffic and a state agency can perform massive decryptions of these communications.

After being neglected for several years, the discrete logarithm record in prime fields has recently received much more attention. In 2014, Bouvier *et al.* [7] have set the record to 596 bits. Then, in 2016, Kleinjung [27] *et al.* have computed a discrete logarithm modulo a 768-bit strong prime. In addition, a discrete logarithm modulo a 1024-bit prime has been done by [17] in a special case where the modulus has particular weaknesses.

Two targets appeared to be the most wanted pre-computations for cryptanalysts, as these two moduli belong to widely used standards:

- The first one is the 768-bit strong prime that defines the Oakley group 1 used in IKE [28]. This group was the default security association recommended when the IPsec standard was established in the late 1990's and, according to [1], is still used by 5.8% of servers running IKEv2.
- The other one is the 1024-bit strong prime that defined the Oakley group 2. Although the worldwide ComSec authorities have asked to withdraw 1024-bit keys since 2010, this group is still used by default by 65% of the servers for IPsec and by 25% of the servers for the SSH protocol according to [1].

The goal of this paper is to carefully study the pre-computation phase for a 768-bit prime and to bring some theoretic improvements to the number field sieve. We compare our results to the current record set by Kleinjung [27] and show that the overall computing power to perform such pre-computation can be divided by a factor of about two using the same source code.

1.3 Organization of the paper

In this paper, we first recall in section 2 the state of the art to perform discrete logarithms using the number field sieve. Then we present some new ideas to improve the efficiency of the algorithm and provide parameters and precise estimates to perform the pre-computation for a 768-bit modulus in different settings.

In section 3, we show that the filtering step can be seen as a simple triangulation process. Then we present a new algorithm to perform the filtering step that transforms the original system of equations into a partial triangular system. This new vision of the filtering step leads to new ideas for the other parts of the algorithm.

After some observations on the triangulation process, we present in section 4 some new types of parameterizations for the sieving step. These parameterizations involve the use of huge composite special-q and improve the overall performance of the algorithm.

In section 5, we improve the linear algebra step by applying the idea of the double-matrix product introduced in [26] on our system of equations after a partial triangulation.

Finally, in section 6, we provide estimations on the ability to perform the pre-computation step for a 768-bit discrete logarithm before the year 2000.

1.4 Results

We compare our results to the 768-bit discrete logarithm record established in June 2016 by Kleinjung *et al.* [27] and the 1024-bit special discrete logarithm performed in [17]. In order to make reliable comparisons that do not depend on implementation quality or CPU performance, we use the same publicly available code on the same architecture:

- For the sieving phase, we use the lattice sieving code written by Franke and Kleinjung [4].
- For the linear algebra phase, we use the latest version of the implementation of the Block Wiedemann algorithm provided by CADO-NFS [8].

Although it is not an optimal choice, we keep most of the parameters used in [27] to ensure that the optimizations presented here only come from our mathematical improvements.

Result 1: the sieving phase The use of huge composite special-q's presented in section 4 allows to save about 50% of the sieving time compared to the result given in [27] : after only 2000 core·years of computation (instead of 4000 core·years in [27]), we get enough relations to build a similar matrix in terms of size and density.

Result 2: the linear algebra phase By using the multi-matrix product on partially triangularized matrices presented in section 5, we estimate that the linear algebra phase can be performed in 144 core·years instead of 308 core·years if the CADO-NFS code was used to perform the linear-algebra step in [27].

Result 3: optimized results In [27], the authors propose optimized parameters that reduce the overall computing time to 2400 core·years if the best available code were used to perform this computation. We present optimized parameters using our improvements that reduce the overall computing time to 1260 core·years: 900 core·years of sieving and 360 core·years of linear algebra.

Result 4: back to the twentieth century The use of a fixed 768-bit prime modulus was the default level of security implemented in various standards written in the late 1990's. We give parameters that allow performing the pre-computation of the database of logarithms in less than 2.5 years using super computers presented on the Top500 list [35] issued in November 2000. This shows that state-level agencies were technically able to perform massive decryption of communications protected by standards such as IPsec/IKE at the time where these standards have been written and approved.

2 The Number Field Sieve for Discrete Logarithms

We briefly recall here how discrete logarithms can be computed using the number field sieve, with a particular highlight on information that is used later in this paper.

We can divide the computation into two phases:

– The first phase is the pre-computation of a database of discrete logarithms of elements with small norms. For a given prime field, defined by a prime modulus p, this pre-computation has to be done once and for all.
– The second phase is the computation of the discrete logarithm of $y = g^x \bmod p$, using the database of discrete logarithms pre-computed during the first part [17]. The computing power needed to compute one individual logarithm is very small compared to the one used to perform the pre-computation.

This means that when a prime field is widely used, then a public release of a pre-computed database of discrete logarithms will allow anyone, even with a small amount of computing power, to compute individual logarithms in this prime field.

In the following, we describe the different steps of the pre-computation of the database of discrete logarithms of elements with small norms.

2.1 Polynomial selection

The goal of this step is to select a pair of bivariate homogeneous polynomials f and g that share a common root $m : 1$ modulo p. When we are given many such pairs, we select the polynomial pair that has the highest probability that $f(a, b)$ and $g(a, b)$ are both smooth with respect to a given bound and when (a, b) is randomly selected in an area of a given size.

There are two efficient methods to find such polynomials. The first one was given by Kleinjung [24], enhanced in [2] and finds two polynomials (f, g) with $\deg(g) = 1$. This method can also be used for integer factorization. The second one has been given by Joux and Lercier [19] and finds two polynomials (f, g), such that f has very small coefficients and $\deg(g)=\deg(f) - 1$. It requires to compute roots of a polynomial modulo the prime p, and hence cannot be used for integer factorization.

According to the complexity analysis of the number field sieve, the best value for the sum of the degrees of the polynomials mainly depends on the size of the considered modulus. For integer factorization, the default parameters given in [8] show that $\deg(f)+\deg(g) = 6$ is well suited for 370-bit to 680-bit moduli, while $\deg(f)+\deg(g) = 7$ is the best choice for moduli of greater size.

As the Joux-Lercier method gives a polynomial pair with $\deg(f)+\deg(g)$ an odd integer, such a pair is usually worse than a Kleinjung pair when the optimal degree sum is even but gives the best pair when the optimal degree sum is odd.

2.2 The Sieving step

Relations During the sieving step, we collect a huge number of coprime pairs of integers (a, b) such that $f(a, b)$ and $g(a, b)$ are both smooth with respect to the following definition :

$$f(a,b) = \prod_{\substack{i \\ p_i < F_f}} p_i^{e_i} \times \prod_{\substack{i=1 \\ F_f < P_i < L_f}}^{\leq n_f} P_i \qquad g(a,b) = \prod_{\substack{i \\ q_i < F_g}} q_i^{f_i} \times \prod_{\substack{i=1 \\ F_g < Q_i < L_g}}^{\leq n_g} Q_i$$

where :

- F_f (resp. F_g) is called the algebraic (resp. rational) factor base bound, and each prime number p_i (resp. q_i) is less than this factor base bound ;
- L_f (resp. L_g) is called the algebraic (resp. rational) large prime bound, and each prime number P_i (resp. Q_i) is less than this large prime bound. Moreover, the number of these large primes P_i (resp. Q_i) is not greater that n_f (resp. n_g).

For simplicity, we still refer to an algebraic side for quantities relative to the polynomial f and a rational side for g even if g, in some cases, is not of degree equals to 1.

The number of relations we have to find is close to the sum of the number of primes below L_f and the number of primes below L_g.

The relation search process The relation search works as follows:

- *Define a sieving zone* : We restrict ourselves to examine coprime pairs (a, b) which belong to the sieving area $[-A, A] \times [1, B]$. The size of this sieving area is chosen in order to find enough relations.
- *Divide the work in several pieces* : The practical search of relations is divided in several independent jobs. We detail in the following how to divide the work using the lattice sieving.
- *Find properties by sieving* : Each job identifies properties of each point (a, b), e.g. if $f(a, b)$ or $g(a, b)$ is divisible by a given prime p_i, and collects this information by computing the size of the cofactor C_f (resp. C_g) such that $f(a, b)/C_f$ (resp. $g(a, b)/C_g$) is smooth with respect to the factor base bound F_f (resp. F_g).
- *Select candidates* : We select points (a, b) such that the cofactors C_f and C_g are small enough. Usually we choose the pairs for which $C_f < L_f^{n_f}$ and $C_g < L_g^{n_g}$.
- *Final selection* : We keep a candidate (a, b) if it fulfills the smoothness condition defined by the parameters F_f, F_g, L_f, L_g, n_f and n_g.

The lattice sieving In the lattice sieving [31], each job handles pairs that belong to the lattice of points (a, b) such that $a - br \equiv 0 \bmod q$, called the special-q lattice, where r is a root of f modulo q. The pair (q, r) is usually called the special-q. This lattice is defined by a reduced basis of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, and the job handles pairs $(a, b) = i\boldsymbol{u} + j\boldsymbol{v}$.

For efficiency, the considered pairs are not those which belong to the original sieving zone, but those where (i, j) belongs to an area $[-I/2, I/2] \times [1, J]$, where I is a power of two and defined such that the coverage of the lattice points is as close as possible to the original sieving zone. Using the sieving by vectors technique [15], we can identify quickly the pairs (a, b) in the special-q lattice such that p divides $f(a, b)$.

When we use a lattice sieving, we only consider pairs (a, b) such that the considered special-q divides $f(a, b)$, and these particular pairs have a greater probability to be smooth than an average pair in the original sieving zone. But we have to take care of duplicates of relations that appear e.g. when a pair (a, b) is such that two special-q divide $f(a, b)$. Such pairs might be found twice by two independent jobs, and duplicates must be removed during the next step.

Some variations of the lattice sieving We recall here two variations of the lattice sieving that have been used in other works, and that we use in this paper.

- *Composite special-q's*: Usually, the special-q values are taken in a range of prime numbers that begins just above the algebraic factor base bound, and is large enough to generate enough relations. Nevertheless, it is possible to use special-q's that are not prime or are less than the algebraic factor base bound. This feature is included in the version 5 of Franke-Kleinjung implementation of the lattice sieve [4] and has been used for example in [21]. But the use of such special-q's tends to generate many more duplicate relations, and so should be used with caution.
- *One-side sieving*: In a traditional sieving, a pair is selected when both cofactors C_f and C_g are small enough. But as the size of the moduli increases, such selection becomes tighter and selecting by considering only one cofactor can be sufficient. By using only one side, we save the time of sieving on the other side while keeping a reasonable number of candidates. The smoothness test for this second side on can be done efficiently using Bernstein trees [5], as it has been used in [16], [26] and [27].

2.3 From relations to equations

Each pair (a, b) leads to an equation modulo q, where $q = p - 1$. This equation links linearly virtual discrete logarithms of ideals in the maximal order of $\mathbf{Q}(\alpha)$ (where α is a root of the polynomial f), virtual discrete logarithms of ideals in the maximal order of $\mathbf{Q}(\beta)$ (where β is a root of the polynomial g), and virtual discrete logarithms of $N_s \leq \deg(f) + \deg(g)$ Schirokauer maps [33].

The relation (a, b), where $f(a, b)$ and $g(a, b)$ can be written as:

$$f(a, b) = \prod_{i=1}^{N_f} p_i^{e_i}, p_i < L_f \qquad g(a, b) = \prod_{i=1}^{N_g} q_i^{f_i}, q_i < L_g$$

leads to the following linear modular equation :

$$\sum_{i=1}^{N_f} e_i \log\left(I_f(p_i, r_i^{(f)})\right) + \sum_{i=1}^{N_g} f_i \log\left(I_g(q_i, r_i^{(g)})\right) + \sum_{i=1}^{N_s} \lambda_i \log(\Lambda_i) + J \equiv 0 \bmod q$$

where

- $I_f(p,r)$ is a prime ideal of $\mathbf{Z}[\alpha]$ of norm p. When p does not divide the discriminant of f, $I_f(p,r)$ is the ideal generated by p and $\alpha - r$, where r is a root of f modulo p. The cases where p divides the discriminant of f give other types of prime ideals. For more details see for example chapter 4.8 in [11].
- Λ_i is a Schirokauer map, and comes in this equation to deal with units in the maximal orders of $\mathbf{Q}(\alpha)$ and $\mathbf{Q}(\beta)$. The number of Schirokauer maps used for each polynomial is equal to the rank of its group of units.
- J is the sum of the virtual discrete logarithms of the inverse of the ideal generated by 1 and α in the maximal order of $\mathbf{Q}(\alpha)$ and the inverse of the ideal generated by 1 and β in the maximal order of $\mathbf{Q}(\beta)$.

The number U of unknown logarithms that appear in these equations is bounded by U_{max}, which is the sum of the number of prime ideals in the maximal order of $\mathbf{Q}(\alpha)$ of norm less than L_f, the number of prime ideals in the maximal order of $\mathbf{Q}(\beta)$ of norm less than L_g, and the number of Schirokauer maps. As there exists on average one prime ideal for each prime number p, this sum U_{max} is close to $\pi(L_f) + \pi(L_g)$, where $\pi(n)$ is the number of prime numbers less than n.

So, when the number R of unique relations collected during the sieving step is greater than U, we obtain a system of R equations and U unknowns and, hoping that U of these equations are linearly independent, this system admits a non-zero solution, which gives some of the values for the discrete logarithms of small prime ideals in the maximal orders of $\mathbf{Q}(\alpha)$ and $\mathbf{Q}(\beta)$ and of the Schirokauer maps. These values are the entries of the database of discrete logarithms which is later used to compute efficiently individual logarithms.

2.4 The filtering Step

We now have to solve a system of R linear equations in $\mathbf{Z}_q$ with U unknowns. We define d, the density of the system, as the average number of unknowns with non-zero coefficients in the linear equations. As non-zero coefficients only appear when a prime ideal occurs in the factorization of $f(a,b)$ or $g(a,b)$, this system is sparse, meaning that $d \ll U$. Such systems can be solved using the Block Wiedemann algorithm [36], for a computational cost proportional to $(d + \alpha) \cdot R^2$, where α depends on the interconnect quality of the supercomputer used to perform this algorithm.

The filtering step [32, 9] transforms this system into another system which is easier to solve, by reducing the number of relations and the number of unknowns in the system and trying to minimize the quantity $(d + \alpha) \cdot R^2$.

Basic operations We define the weight of an unknown as the number of equations where this unknown has a non-zero coefficient. We can perform these transformations on the system to simplify it:

– Remove singletons: If an unknown has a weight equal to one, we remove the equation where this unknown appears. This always reduces the number of equations R and the total of non-zero coefficients in the system $d \cdot R$, and thus reduces the computational cost.
– Purge equations: while R is greater than U, we can remove an equation without losing the property that the solutions of the system are the wanted discrete logarithms. For example, if we remove an equation with several unknowns of weight two, these unknowns become singletons and we can remove them.
– Merge equations: we can remove one unknown by performing one round of a Gauss-Jordan elimination: if an unknown is of weight w, we carefully choose one equation containing this unknown (called the pivot) and subtract it from the $w - 1$ other equations containing this unknown.

Strategy The filtering step begins with several rounds of singletons removal: one round of singletons removal (i.e. remove equations containing an unknown of weight one) makes new singletons appears. So we run several rounds until no more unknown of weight one remains into the matrix. Then we purge equations until only a little excess $R-U$ remains. A classical strategy is to keep an excess rate $(R-U)/U$ of about 10%. Details and improvements of the purge step have been studied in [6].

Then, merges are performed according to the Markowitz criterion [30], which selects the unknown and the associated pivot that minimizes the increase of the density d of the system. Cavallar [9] has proposed an algorithm that efficiently performs the merge algorithm. Moreover, she optimizes the application of a pivot using a minimum spanning tree that reduces in some cases the increase of the density. In the following of this paper, we ignore this optimization.

We stop the merge phase when the expected running time of the linear algebra step is minimal. If we use the Block Wiedemann algorithm, with an expected running time equal to $(d+\alpha) \cdot R^2$, we stop when the increase of the density d is no longer balanced by the decrease of the matrix size R, thus when $m \approx 2(d+\alpha)$ where m is the weight increase.

The stopping point of the merge phase highly depends on the way the linear algebra phase is done. We explain in the following that the double-matrix product technique introduced by [26] changes the running time of the algorithm and hence changes the location of this stopping point.

The filtering step ends with a final purge step, where the $R - U$ heaviest equations are removed, and outputs a final system with $R = U$.

The double-matrix product We recall here an idea introduced in [23]. We can write the system of linear modular equations in the matrix form $M \cdot x \equiv 0 \bmod q$, where M is an $R \times R$ matrix. Each row represents an equation, each column represents an unknown, and each entry is the coefficient of an unknown in an equation. Let M_R be the matrix of relations, i.e. the matrix representing the system of equations after the singleton removal and the purge phase. The merge phase, which combines equations (i.e. rows of the matrix) linearly, can be seen as a left multiplication by a matrix M_F, i.e. the final matrix M output by the filtering step can be written as $M = M_F \cdot M_R$. The linear combination of rows creates m zero columns in M that can be removed by multiplying M by an extraction matrix M_E, leaving a $(R - m) \times (R - m)$ matrix $\overline{M} = M \cdot M_E$.

The most consuming part of the Block Wiedemann algorithm consists in performing at each iteration the matrix-vector multiplication $V' = \overline{M} \cdot V$. It is suggested in [23, 26] to perform this multiplication by first computing $V_t = \overline{M_R} \cdot V$, where $\overline{M_R} = (M_R \cdot M_E)$, and then computing $V' = M_F \cdot V_t$. The theoretical cost of a matrix-vector multiplication, i.e. the number of basic operations on vector entries, is proportional to the weight $w(\overline{M})$ of the matrix, which is the number of non-zero entries in the matrix M. When the double-matrix product is performed, the cost of one iteration is $w(M_F) + w(\overline{M_R})$ instead of $w(\overline{M})$. As M_F and $\overline{M_R}$ are very sparse matrices, $w(\overline{M})$ is close to $w(M_F) \cdot w(\overline{M_R})$, and as $w(M_F) + w(\overline{M_R}) \ll w(\overline{M})$, the double-matrix product technique is theoretically more efficient than the classical matrix-vector product.

In a practical point of view, the real time of a matrix-vector product consists not only in the time to perform basic operations on vector entries, but also in cache misses, i.e. the time needed to unload a page of memory that handles some matrix or vector data and load another page, and interconnect time to broadcast data to multiple nodes when a distributed memory cluster is used. As M_F and $\overline{M_R}$ are more sparse than M, leading to proportionally more cache misses, and as V_t must also be broadcast during a double-matrix product computation, most of the theoretical advantage of the double-matrix product can be lost in practice.

Nevertheless, the main advantage of the double-matrix product is that it changes the stopping point of the filtering phase: instead of stopping the merge when the density of M becomes too large, we can go further and keep merging unknowns while the density of the matrix M_F remains reasonable. We can also create several successive matrices $M_F^{(i)}$ such that $M = \left(\prod_i M_F^{(i)}\right) \cdot M_R$, as suggested in [26]. As a consequence, the final number of remaining unknowns U is lowered when using the double-matrix product technique and hence reduces the number of iterations needed in the Block Wiedemann algorithm.

2.5 The Linear Algebra Step

We do not pretend here to explain in detail the Block Wiedemann algorithm, for more details, see [34]. We just recall here some aspects of the algorithm.

First, the system we want to solve is modified: the unknowns corresponding to the virtual logarithm of the Schirokauer maps create dense columns in the matrix M, i.e. columns where coefficients are random-like numbers in $\mathbf{Z}_q$, whereas the other columns are small and sparse (most coefficients are zeros, and non-zero coefficients are very small). We cut the matrix M in two parts: M_{sparse} which contains the small sparse columns and M_{dense} which contains the columns related to the Schirokauer maps. Then we solve $M_{sparse} \cdot x + M_{dense} \cdot y = 0$ instead of solving $M \cdot x = 0$. More details can be found in [20, 17].

We first choose two parameters μ and ν. Taking $\mu = \nu = 16$ is a usual parametrization. The most consuming part of the algorithm is to compute iteratively ν independent Krylov sequences $V_{i+1} = M_{sparse} \cdot V_i$ for i up to $\mathrm{rk}(M) \cdot (1/\nu + 1/\mu) + o(1)$, and to store the μ first coefficients of V_i. Then, a linear generator for the ν sequences is computed with an algorithm whose time and memory complexity depends on ν and μ. Finally, a solution to the system can be computed thanks to the previously generated data in small time compared to the Krylov sequence generation, as explained in [17]. The parameters μ and ν are chosen to offer a good compromise between the time needed to compute the Krylov sequence and the time and memory required to perform the linear generator computation.

2.6 Expansion of the database of logarithms

The linear algebra phase outputs a solution to the system of equations, which consists in the virtual logarithms of the Schirokauer maps and the discrete logarithms of ideals which are represented in the final matrix output by the filtering step. But, to this point, discrete logarithms of ideals that have been eliminated from the matrix by the merge, by the purge or by the singleton removal are still unknown.

We now restart from the original equations output by the sieving phase. We notice that, for some of these equations, there is only one unknown that has not been solved by the linear algebra phase. So this unknown can easily be found thanks to the considered equation. This expands the number of unknowns that have been solved, and this process can be iterated to find new equations with only one unsolved unknown. After several such steps, all the unknowns are solved, and we are given a full database of discrete logarithms that appear at least once in the original equations output by the sieving phase.

The previous algorithm only works when particular circumstances that ensure that there is still equations with only one unsolved unknown at each iteration. The next section explains the conditions under which this algorithm works, conditions that are always met in practical cases.

3 Partial Triangulation of Systems of Equations

In this section, we give a graph representation of the merge phase and give a sufficient condition that ensures the success of the algorithm given in section 2.6. Then, we present another way to consider the merging phase of the filtering step that leads us to several optimizations for the filter step and the linear algebra phase.

3.1 Graph representation of the merge phase

Let us first define a graph representation of a merge phase once this phase has been computed using classical methods as described in section 2.4.

Definition 1. *The graph representation G of a merge phase is an oriented graph whose vertices are the original equations and whose edges link the pivot for a given unknown of weight w to the $w-1$ other equations containing this unknown.*

Then, we give a definition of the condition that ensures that the algorithm described in section 2.6 succeeds.

Definition 2. *A graph representation G is said to be cycle-free if the non-oriented graph $\tilde{G}$ whose edges corresponds to the oriented edges of G contains no cycle.*

Before explaining why this condition is sufficient to ensure the success of the algorithm, let us introduce more definitions on the relations, i.e. for the vertices of the graph. Note that these definitions have only a sense if the graph G is cycle-free, i.e. only contains chains that have a starting point and an end point.

Definition 3. *A level-1 pivot is a relation represented by a vertex r in G such that r is never an end point for edges in G. A level-i pivot is a relation represented by a vertex r in G such that r is the end point of a chain of length equal to $i-1$ but is never the end point of a chain of length greater than $i-1$.*

Definition 4. *A base relation is a relation represented by a vertex r in G such that r is never a starting point for any edge in G.*

Figure 1 represents the graph representation of the merge step for a toy example. In this example, unknowns $x_1, \dots, x_5$ have been merged and only unknowns $x_6, \dots, x_9$ lie into the final system solved by the linear algebra step.

When values have been found for these unknowns, the expansion algorithm finds the values for $x_1, \dots, x_5$ in two steps: during the first step, the algorithm uses three equations R_8, R_7 and R_2, that contains only one merged unknown, to compute the value of this unknown. These three equations respectively gives values for the three unknowns x_2, x_3 and x_4. These three equations come from the three relations that are level-1 pivots. During the second step, the two equations R_4 and R_3 that come from the level-2 pivots are respectively used to find the values for the two remaining unknowns x_1 and x_5.

Finally, the four equations R_1, R_5, R_6 and R_9 come from the four base equations and form the final matrix once pivots have been added or subtracted to remove merged unknowns.

To prove the success of the algorithm given in section 2.6 is straightforward once we have divided the set of the pivots into different levels and see that each pivot of level i solves one new unknown during the iteration i of the algorithm.

3.2 Partial triangular systems

We say that a system of R equations and R unknowns can be put into a m-partial triangular form if we can write this system of equations as $M_R \cdot x = 0$, where M_R is a m-partial triangular matrix, i.e. the $m \times m$ submatrix containing

Original set of equations:

$$\begin{array}{rcccccccccccl}
 & & \multicolumn{5}{c}{\overleftrightarrow{\text{merged}}} & \multicolumn{4}{c}{\overleftrightarrow{\text{unmerged}}} & & \\
R_1 & : & x_1 & & & & +\,x_5 & +\,x_6 & +\,x_7 & & +\,x_9 & = & 0\\
R_2 & : & & & & x_4 & & & +\,x_7 & +\,x_8 & & = & 0\\
R_3 & : & & & & x_4 & +\,x_5 & +\,x_6 & & & +\,x_9 & = & 0\\
R_4 & : & x_1 & +\,x_2 & & & & & +\,x_7 & +\,x_8 & & = & 0\\
R_5 & : & & & x_3 & & +\,x_5 & +\,x_6 & +\,x_7 & & +\,x_9 & = & 0\\
R_6 & : & & & & x_4 & & +\,x_6 & +\,x_7 & +\,x_8 & +\,x_9 & = & 0\\
R_7 & : & & & x_3 & & & & & +\,x_8 & +\,x_9 & = & 0\\
R_8 & : & & x_2 & & & & +\,x_6 & +\,x_7 & & & = & 0\\
R_9 & : & & & x_3 & & & +\,x_6 & & & +\,x_9 & = & 0
\end{array}$$

Graph representation of the merge step:

$$\begin{tikzcd}
\text{Level-1 pivots} & \text{Level-2 pivots} & \text{Base equations} \\
\boxed{R_2} \arrow[rr] \arrow[rd] & & \boxed{R_6} \\
 & \boxed{R_3} \arrow[r] \arrow[rd] & \boxed{R_5} \\
\boxed{R_8} \arrow[r] & \boxed{R_4} \arrow[r] & \boxed{R_1} \\
\boxed{R_7} \arrow[rruu] \arrow[rr] & & \boxed{R_9}
\end{tikzcd}$$

Final equations:

$$\begin{array}{lcccccl}
R_6 - R_2 & : & x_6 & & & +\,x_9 & = 0\\
R_5 - R_3 + R_2 - R_7 & : & & 2x_7 & & -\,x_9 & = 0\\
R_1 - R_3 - R_4 + R_2 + R_8 & : & x_6 & +2x_7 & & & = 0\\
R_9 - R_7 & : & x_6 & & -\,x_8 & & = 0
\end{array}$$

Partial triangulation of the original system of equations:

$$\begin{array}{rcccccccccccl}
 & & \multicolumn{5}{c}{\overleftrightarrow{\text{merged}}} & \multicolumn{4}{c}{\overleftrightarrow{\text{unmerged}}} & & \\
R_8 & : & x_2 & & & & & +\,x_6 & +\,x_7 & & & = & 0\\
R_7 & : & & x_3 & & & & & & +\,x_8 & +\,x_9 & = & 0\\
R_2 & : & & & x_4 & & & & +\,x_7 & +\,x_8 & & = & 0\\
R_4 & : & x_2 & & & +\,x_1 & & & +\,x_7 & +\,x_8 & & = & 0\\
R_3 & : & & & x_4 & & +\,x_5 & +\,x_6 & & & +\,x_9 & = & 0\\
R_6 & : & & & x_4 & & & +\,x_6 & +\,x_7 & +\,x_8 & +\,x_9 & = & 0\\
R_5 & : & & x_3 & & & +\,x_5 & +\,x_6 & +\,x_7 & & +\,x_9 & = & 0\\
R_1 & : & & & & x_1 & +\,x_5 & +\,x_6 & +\,x_7 & & +\,x_9 & = & 0\\
R_9 & : & & x_3 & & & & +\,x_6 & & & +\,x_9 & = & 0
\end{array}$$

Fig. 1. Merge of equations and graph representation

the m first rows and the m first columns of M_R is triangular : $M_R(i,j) = 0$ when $1 \leq i < j \leq m$ and $M_R(i,i) \neq 0$ when $1 \leq i \leq m$. For example, figure 1 gives the 5-partial triangular form of the system of equations, once equations and unknowns have been properly sorted.

When a system of equations has been written in a m-partial triangular form, it can be solved using the following steps:

– *Merge :* The m first rows of M_R are used as pivots and are added or subtracted to the $R-m$ last rows so that an $(R-m) \times m$ zero block appears in the lower left part of the matrix. This elimination can be seen as the left multiplication of M_R by an $(R-m) \times R$ matrix M_F, such that $M = M_F \cdot M_R$, and M is a $(R-m) \times U$ matrix whose m first columns are zero.
– *Columns removal :* As the m first columns of M are zero, we rewrite M_R by removing its first m columns, leading to an $R \times (R-m)$ matrix denoted $\overline{M_R}$. When multiplied by the $(R-m) \times R$ matrix M_F, it gives the $(R-m) \times (R-m)$ final matrix $\overline{M}$, i.e. the matrix obtained when the m first columns of M are removed.
– *Linear algebra :* We solve the equation $\overline{M} \cdot x = 0$. If an iterative algorithm is used such as the Block Wiedemann algorithm or the Lanczos algorithm, one can take advantage that $\overline{M}$ can be written as $\overline{M} = M_F \cdot \overline{M_R}$. This gives solutions for the $U-m$ last unknowns.
– *Expansion :* each of the m first unknowns is successively found using the mth equation of the partial triangular system.

Partial triangulation can be seen as another way to perform the merging step: it defines a matrix M_F that combines the original equations in order to remove rows and columns in the final matrix. We present in the next section a very fast algorithm to perform a partial triangularization on a matrix.

3.3 A greedy algorithm to find a triangular form

Let M_R be a matrix representing a system of equations. We assume in this section that we are given a set of m unknowns for which we want to find a m-partial triangular system. We give in the next section an easy way to build such set of unknowns. By analogy with the classical merge step, we can call these m unknowns the merged unknowns. Without loss of generality, we can assume that these m merged unknowns are represented by the m first columns of M_R.

Here is an algorithm to transform the system of equations into a m-partial triangular system.

```
1. Global initializations
    – Let $M_R$ be the set of original equations
    – Let $\mathcal{P} = \{\}$ be the set of pivot equations
    – Let $\mathcal{U} = \{1, \ldots, m\}$ be the unknown indices to be merged
    – Let $\mathcal{Z} = \{\}$ be the unknowns indices already merged
    – Let $W(r) = 1$ be the weight for each equation $r$ in $M_R$
    – Let $P(u) = undef$ be the pivot chosen for each unknown $u$
```

```
   – Let $\ell = 0$
2. Loop initializations
   – Increase the loop counter $\ell$ by one
   – Let $C(u) = \{\}$ be the pivot candidates for each unknown $u$
3. Find candidates: for each $r$ in $M_R$
   – If $r$ has only one unknown $u$ in $\mathcal{U}$ : add $r$ to $C(u)$
4. Choose candidates : for each $u$ in $\mathcal{U}$ such that $C(u)$ is not empty
   – Set $r_0$ as an element of $C(u)$ with minimal weight $W(r_0)$
   – Set $P(u) = r_0$
   – Remove $r_0$ from $M_R$ and add it to $\mathcal{P}$
   – Remove $u$ from $\mathcal{U}$ and add it to $\mathcal{Z}$
5. Recompute weights : for each $r$ in $M_R$
   – Set $W(r) = 1 + \sum_{u \in K} W(P(u))$ where K is the set of unknowns in $r$.
6. Go to 2. while $U$ is empty (success) or while nothing happened in the loop
   (fail)
7. Sort the unknowns in $M_R$ and $\mathcal{P}$ according to the order of the unknowns
   indices in $\mathcal{Z}$
8. Write equations in $\mathcal{P}$ in that order, then the remaining equations in $M_R$
```

Note that in this algorithm, $W(r)$ represents the number of equations that are added or subtracted together when a Gauss-Jordan elimination is performed to remove unknowns in the equation r whose indices lies in $\mathcal{Z}$ and when the pivot used to remove the unknown u is $P(u)$. According to definition 3, a pivot $P(u)$ that has been chosen at the loop iteration ℓ is a level-ℓ pivot. The value $W(r)$ is computed in step 5 for each relation that still belongs to M_R, using values $W(P(u))$ for relations $P(u)$ that have been removed from M_R during previous loops.

The matrix M_F such that the final matrix M is equal to $M_F \cdot M_R$ has exactly $\sum_{r \in M_R} W(r)$ non-zero coefficients, where M_R is the set of equations given by the base relations. The choice of r_0 with minimal weight for each pivot ensures that the weight of M_F is minimal for the set of unknowns $\mathcal{Z}$.

3.4 Selecting the set of unknowns to be merged

In this section, we discuss briefly on the method we can use to find the set $\mathcal{U}$ of m unknowns for which we find a m-partial triangular system thanks to the previous algorithm, trying to find such a set with the greatest possible value m. At first sight, to find the set $\mathcal{U}$ with maximal possible cardinal m seems to be a hard problem. So we restrict ourselves to a non-optimal algorithm that finds a non-optimal set $\mathcal{U}$ which is nevertheless greater than the one found with classical versions of the sieving step.

To find this set $\mathcal{U}$, we first sort all the unknowns with respect to their weight, i.e. the number of equations that hold this unknown. We begin with a relatively small value of m and set $\mathcal{U}$ as the m first sorted unknowns, i.e. the m unknowns with the smallest weight. We run the triangulation algorithm; while this algorithm succeeds, m is progressively increased and the new set $\mathcal{U}$ is tested until the triangulation algorithm fails. We then set m as the greatest value for which the triangulation algorithm succeeds, and $\mathcal{U}$ as the m first sorted unknowns.

4 Sieving with Composite Special-*q*'s

In this section, we use our observations on the graph representation of the merging phase to explain why the special-q lattice sieving usually prevents us from building small systems of equations. We discuss the techniques used in record computations, such as the pre-computation for discrete logarithms for a 1024-bit special prime [17] and for a 768-bit strong prime [27]. Then, we explain that the use of very large composite special-q's gives better results than the techniques previously used and present the results of our experiments that compare these techniques.

4.1 Prime special-*q*'s and partial triangulation

When we want to get a small matrix, a reasonable goal for the filtering step is to eliminate every unknown representing large primes and keep only the unknowns representing primes that belong to the factor base.

When we use the lattice sieving for integer factorization, we choose special-q's as prime numbers that lie into the set of algebraic large primes. With this choice, we generate relations that give equations that always hold an unknown representing a special-q.

The partial triangular algorithm from section 3 needs enough level-1 pivots to succeed. But, if the set of the merge ideals are the large primes, the only level-1 pivots come from equations where no extra large prime is found on both sides. Moreover, no level-1 pivot can be found for large primes that are not used as special-q. So, if we use a traditional parameterization for the lattice sieving and try to perform the partial triangulation on the entire set of unknowns representing large primes, this partial triangulation usually fails because of the lack of enough level-1 pivots.

4.2 Use of special-*q*'s in the factor base

In [27], Kleinjung *et al.* try to circumvent the special-q problem by choosing special-q's that belong to factor base. In this case, the number of choices for the special-q's is reduced and it becomes necessary to generate many relations per special-q. This condition can only be achieved by increasing the lattice sieving area $[-I/2, I/2] \times [1, J]$.

In [27], the size of the lattice sieving zone $I \times J$ is up to 2^{40} instead of a classical size of 2^{31} for 768-bit integer factorization. This reduces the number of relations found per second, as the increase of the lattice sieving zone by a factor of four usually multiplies the sieving time by four while only multiplying the number of relations by a factor close to two.

This method also limits the ability to reduce the size of the final matrix using over-sieving, i.e. generating more relations than needed, because the special-q's used to generate these relations will be outside the factor base. This explains why in table 2 in [27], the increase of the sieving effort from 2400 core·years to 4000 core·years reduces the matrix effort by only 26%.

4.3 Use of a large fraction of large primes as special-q's

A completely different type of parameterization has been used for the 180-digit record [7] and the 1024-bit SNFS experiment in [17]. In these two computations, the special-q ranges cover a large fraction of the large primes: In [7], every algebraic large prime is chosen as a special-q, and in [17], about three-quarters of both algebraic and rational large primes are used as special-q's.

This kind of parameterization creates a lot of duplicates, as most of the large primes found in one relation will be used as special-q elsewhere during the computation. This effect must be taken into account to evaluate the real efficiency of the sieving phase.

Moreover, such parameterization cannot achieve the elimination of every large prime during the filtering phase. For example, in [17], the size of the factor base is about 16.10^6, whereas the final matrix has about 28.10^6 columns.

4.4 Sieving with huge composite special-q's

We propose here another type of parameterization based on the use of huge composite numbers as special primes, previously introduced in [21].

The first idea is to use composite special-q's: the advantage of using composite numbers is to decrease by one the number of large primes in every relation, as the special-q is no longer a large prime but the product of two small primes. Moreover, it removes the bad prime special-q effect that prevents the existence of level-1 pivots, as every relation with exactly one large prime becomes a level-1 pivot.

The use of composite special-q's generates more duplicates. Nevertheless, we can take control over the generation of too many duplicates by noticing that the percentage of unique relations is mostly driven by the ratio between the two bounds of the special-q range. If q is taken among the composite numbers that lie within the range $[q_1, q_2]$ such that its prime factors are all greater than a given bound (equal to 2048 in our experiments), then practical experiments show that when q_2/q_1 is close to two, then the percentage of unique relations is around 60%.

Here comes the second idea: instead of using a small number of special-q's in order to respect the condition $q_2/q_1 \approx 2$, we take very large values for q_1 and q_2, even if these values become greater than the large prime bound. While the parameterization used in [27] takes q_2 close to the factor base bound F_f and a lattice sieving area up to 2^{40}, we prefer to take $q_2 = q_1/2 \approx 500F_f$ and a lattice sieving area equal to 2^{31}. With these two types of parameterizations, we search for relations (a, b) where $|a|$ and $|b|$ are bounded by $2^{20}\sqrt{F_f}$, but when we use a huge composite special-q, we search for relations on points where $f(a, b)$ is divisible by a 500 times greater special-q.

4.5 Practical results

In this section, we present parameters and results for experiments on the precomputation for the 768-bit discrete logarithm [27] and the special 1024-bit discrete logarithm [17] records, summarized in table 2. In order to measure the improvements provided by the use of huge composite special-q's, we keep most of the parameters used in [27] and [17], even if some of these parameters are probably not optimal. We tune our parameters in order to produce a final matrix which has roughly the same size and the same density than the one computed in the original computation.

	Experiment 1		Experiment 2		Experiment 3	
	[27]	This paper	[27]	This paper	[17]	This paper
Type of special-q's	Primes	Compos.	Primes	Compos.	Primes	Compos.
Area size IJ	38-40	31	40	31	31	31
# large primes	(2,2)	(2,2)	(2,2)	(2,2)	(3,2)-(2,3)	(2,2)
Special-q side	f	f	f	f	f and g	f
q_1	190.10^6	75.10^9	190.10^6	35.10^9	150.10^6	950.10^6
q_2	630.10^6	150.10^9	306.10^6	70.10^9	1560.10^6	2450.10^6
# special-q's	$22.2 \cdot 10^6$	$2.56 \cdot 10^9$	$5.8 \cdot 10^6$	$1.17 \cdot 10^9$	$138 \cdot 10^6$	$70 \cdot 10^6$
Time per q (sec)	2550–8750	24.85	8750	24.85	50.87	61.99
Time (core·y)	4000	2000	1625	900	222	138
# relations	*unknown*	$10.51 \cdot 10^9$	*unknown*	$5.97 \cdot 10^9$	$520 \cdot 10^6$	$301 \cdot 10^6$
# unique	$9.08 \cdot 10^9$	$6.25 \cdot 10^9$	$3.24 \cdot 10^9$	$3.74 \cdot 10^9$	$249 \cdot 10^6$	$202 \cdot 10^6$
% unique	*unknown*	59.5%	*unknown*	62.6%	47.9%	67.1%
matrix size	$23.5 \cdot 10^6$	$23.5 \cdot 10^6$	$32.7 \cdot 10^6$	$32.1 \cdot 10^6$	$28.5 \cdot 10^6$	$28.1 \cdot 10^6$
matrix density	134	128	189	180	200	200
Sieving time saved	50 %		45 %		38 %	

Fig. 2. Parameters and estimations for our experiments

Details about the simulation process which evaluates the size and the density of the final matrix, as well as the rules to evaluate the computing power used, are given in Appendix B.

DL768 – Experiment 1 We choose sieving parameters in order to build a matrix with 23.5 million rows and columns with density $d = 134$, i.e. a matrix similar to the one found in [27] after 4000 core·years of sieving. We take the composite special-q's in the range $[75.10^9, 150.10^9]$ and use a lattice sieving area where $I = 2^{16}$ (i.e. $IJ = A = 31$ using [27] notations). We use the composite numbers whose factorization into prime numbers only holds primes greater than 2048. The running time of this sieving step is estimated to 2000 core·years. We conclude that the huge composite technique allows us to save 50% of the sieving step running time. Then, a simulation process (see Appendix B for details) is done in order to evaluate the size and the density of the matrix that can be built from relations found with our new parameters. This process shows that this matrix has roughly the same characteristics than the one found in [27].

DL768 – Experiment 2 In this second experiment, we compare ourselves to the optimal parameters given in [27] by choosing parameters that build a matrix with 32.7 million rows and columns and with density $d = 189$. We take the composite special-q's in the range $[35.10^9, 70.10^9]$ and use a lattice sieving area where $I = 2^{16}$. Our parameters allow us to perform the sieving step in 900 core·years instead of 1625 core·years in [27] to find a similar matrix. The running time of the sieving steps is reduced by roughly 45%.

SNFS1024 – Experiment 3 We now compare our parameterization method to the one used in [17], where the sieving step has been done using most of the large primes as special-q's on both sides. We estimate that this sieving step can be done in 207 core·years with parameters from [17] when code from [4] is used. Once again we adjust our parameters in order to find a similar matrix, i.e. a matrix with 28.5 million rows and columns with density $d = 200$. We take the composite special-q's in the range $[950.10^6, 2450.10^6]$ and use a lattice sieving area where $I = 2^{16}$. We choose factor bases $F_f = F_g = 350 \cdot 10^6$ and allow only two large primes on both sides. To be fair, we do not use the one side sieving technique as it is not used in [17]. This sieving step can be performed in only 132 core·years, thus saving 36 % of the running time.

5 Solving Sparse Systems using Partial Triangulation

We have presented in the section 3 an algorithm to write the original matrix M_R in a m-partial triangular form that allows us to transform the original system into a matrix equation $M \cdot x = 0$, where $M = M_F \cdot M_R$. In this section, we explain how to use this special form to solve efficiently this matrix equation using the Block Wiedemann algorithm by improving the efficiency of the matrix vector multiplication of M by a vector V. As no code using this new technique has been implemented yet, we only give estimates of the efficiency of this method based on some reasonable assumptions.

5.1 A special multi-matrix product for partial triangular matrices

Figure 3 recalls the special form of the original system of R equations and R unknowns once it has been transformed into a m-partial triangular form M_R. In this figure, there are ℓ_1 equations that are level-1 pivots and are dedicated to the elimination of the coefficients of the first ℓ_1 unknowns in the remaining equations. More generally, there are ℓ_i equations that are level-i pivots and dedicated to the elimination of ℓ_i unknowns. Let n be the number of iterations needed to put the matrix into a m-partial triangular form.

Let $S_i = 1 + \sum_{j=1}^{i-1} l_j$ and $T_i = \sum_{j=1}^{i} l_j = S_{i+1} - 1$, and $m = \sum_{j=1}^{n} l_j$. Then:

- the rows from S_i to T_i are the level-i pivots.
- For each i, the submatrix from rows between S_i and T_i and columns between S_i and T_i is an identity matrix.
- For each i, the submatrix from rows between S_i and T_i and columns between S_{i+1} and m is a zero matrix.

For each i, we note C_i the submatrix from rows between S_{i+1} and R and columns between S_i and T_i. We also note $L_i = (-C_i, Id)$ as the $(R - T_i) \times (l_i +$

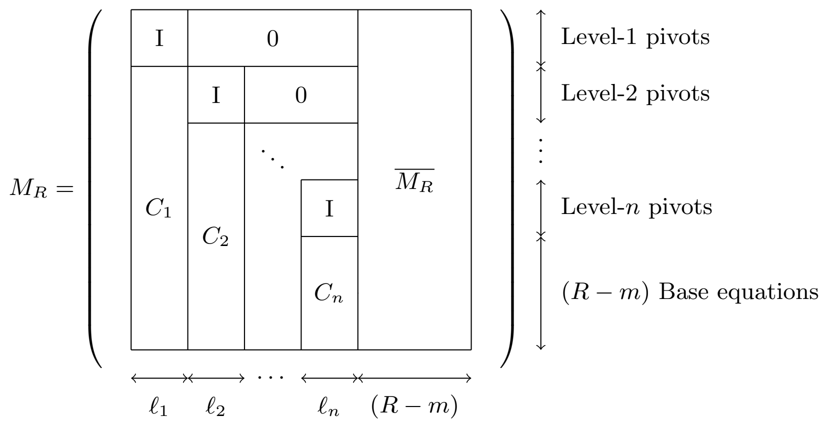

Fig. 3. Matrix in its partial triangular form

$R - T_i)$ matrix whose l_i first columns are the columns of $-C_i$ and the $(R - T_i)$ remaining columns form an identity matrix.

The left multiplication of M_R by L_1 performs the Gauss-Jordan elimination on the l_1 first columns using the l_1 first rows as pivots. This means that the first l_1 columns of the resulting matrix $M_1 = L_1 \cdot M_R$ are zero. Recursively, we see that for all i up to n, the left-multiplication of M_{i-1} by L_i gives a matrix $M_i = L_i \cdot M_{i-1}$ beginning with T_i zero columns. As a consequence, the final matrix M_n has $R-m$ rows and only $R-m$ non-zero columns and the matrix $\overline{M}$ consisting of these $R - m$ non-zero columns of M_n is the expected matrix after merging. This matrix $\overline{M}$ is such that $\overline{M} = M_F \cdot \overline{M_R}$, where M_F is the product of the L_i matrices : $M_F = L_n \cdot \dots \cdot L_2 \cdot L_1$.

This leads to the following algorithm, called the multi-matrix product algorithm, that computes V', the result of the multiplication between the square matrix $\overline{M}$ and a vector V of length $R - m$:

```
Set $V_1 = \overline{M_R} \cdot V$;
**for** $i$ **to** $n$ **do**
  | Set $V_{i+1} = L_i \cdot V_i$
**end**
Set $V' = V_{n+1}$;
```

5.2 Practical efficiency of the multi-matrix product

In this section, we show that the computation of the vector $V' = \overline{M} \cdot V$ using the multi-matrix product algorithm has roughly the same cost than the multiplication of the original matrix M_R by a vector of length R. This means that the use of the multi-matrix product does not change the time of a single of an iteration while reducing the number of iterations needed.

Arithmetic cost We first evaluate the number of basic operations to perform the multiplication. We consider that one basic operation is the addition or the subtraction between two entries of the vector V (where one of them is rarely multiplied by a coefficient of the matrix which is not 1 or -1). When we consider the matrix M_R of size R with $w(M_R)$ non-zero coefficients, the number of basic operations is $w(M_R)$. For the multi-matrix product algorithm, we count $w(\overline{M_R})$ basic operations for step 1, then $w(C_i)$ basic operations for each step of the loop in step 2. As $w(M_R) = w(\overline{M_R}) + \sum w(C_i)$, the number of basic operations to perform the multi-matrix product algorithm is equal to the number of basic operations to multiply a vector V by the original matrix M_R.

Dealing with inter-processor communications When we solve large systems, we frequently use clusters consisting of several processors linked with inter-processor technologies. To compute the Krylov sequence, we have to perform at each iteration the multiplication $V' = M_R \cdot V$. At the end of this computation, each processor holds only a fragment of the vector V' and we have to broadcast these fragments of V' to every processor before starting the next iteration. So we have to broadcast a vector of length R at each iteration.

When we use the multi-matrix product algorithm, only the l_i first entries of the vector V_i must be broadcast to the processors at each step of the loop to let them compute their own fragment of V_{i+1} from their fragment of V_i. And at the end of the computation, the final vector that has to be broadcast has a length equal to $R - m$. So at each iteration, we have to broadcast a total of $R - m + \sum l_i = R$ entries, that is equal to the number of entries broadcast when we perform $V' = M_R \cdot V$.

Conclusion The number of basic operations and the inter-processor communications needed to perform one iteration of the Block Wiedemann algorithm using the multi-matrix product are equal to the operations and communications needed to perform a classical product between the matrix M_R and a vector of length R. Throughout the rest of this paper, we assume that the computing times for these two operations are equal, despite some differences can occur due *e.g.* to differences in cache misses.

5.3 Cost optimization using a light merging phase

The m-partial triangulation and the multi-matrix product allow us to perform the first part of the linear algebra phase, i.e. the generation of the Krylov sequence, such that :

– the number of required iterations is $\mathrm{rk}(M) \cdot (1/\nu + 1/\mu) + o(1)$, where $\mathrm{rk}(M)$ is equal to $R - m$
– each iteration consists in a multi-matrix product whose cost is roughly equal to the multiplication between the $R \times R$ matrix M_R and the vector of length R.

In this section, we transform the matrix M_R into a modified matrix M'_R using the merging algorithm presented in section 2.4 in order to reduce the practical cost of the multi-matrix product.

From a theoretical point of view, the cost of the multi-matrix product is equal to the number of basic operations to multiply a vector by M'_R, i.e. equal to $w(M'_R)$. The merging phase increases the weight of the matrix except when a merge on an unknown of weight 2 is performed. So the merging technique that minimizes the cost of the multi-matrix product is to perform all the merges on unknowns of weight 2.

From a practical point of view, it is beneficial to perform the merging step a little more: the increase of basic operations is balanced by the reduction of the size of the matrix because this reduces the amount of inter-processor communications. The optimal stopping point comes when the practical cost of one iteration is reached.

5.4 Practical experiments

Figure 4 presents practical experiments on various sets of relations: The two first experiments are done on simulated relations generated during the two first experiments in section 4 of this paper. The third experiment is done relations used in [17] to perform the pre-computation phase for a special 1024-bit prime.

We first use a classical implementation of the merging phase in order to produce a matrix for the classical version of Block Wiedemann algorithm. Then, we apply to the same set of relations our new ideas: we first perform a light sieving in order to reduce the cost of a single iteration of the Block Wiedemann algorithm. Then, we perform the partial triangulation to reduce the rank of the matrix M. Finally, we compare the expected running time of the two methods.

Formulas used to estimate the computing time of a single iteration on 140 cores are given in Appendix B. The number of iterations is given when the parameters of the Block Wiedemann algorithm are $\mu = \nu = 2$.

6 Back to the twentieth century

The results presented in this paper show that the pre-computation of a database of discrete logarithms in order to break communications based on the security of 768-bit discrete logarithm is easily reachable today with an academic sized computing power, thus show that such ability has been reached by state-level agencies many years ago. This last experiment tries to evaluate if this pre-computation was feasible by the year 2000, using the parameters presented in experiment 2.

The sieving step running time is estimated to 2000 core·years on a single core at 2.2 Ghz. In the year 2000, the most powerful supercomputer was the ASCI White at the Lawrence Livermore National Laboratory [35], consisting of 8192 POWER3 processors running at 375 MHz. We can estimate that this supercomputer can perform the sieving step in $2000 \times 365 \times 2200/375/8192 = 523$ days.

	Experiment 1	Experiment 2	Experiment 3
Classical merge			
Matrix size	$23.5 \cdot 10^6$	$32.1 \cdot 10^6$	$28.5 \cdot 10^6$
Matrix density	134	189	200
Iteration time on 140 cores	1.97 sec	3.62 sec	0.70 sec
# Iterations	$35.3 \cdot 10^6$	$48.2 \cdot 10^6$	$42.7 \cdot 10^6$
Total time (core·y)	308	730	132
New methods			
Light Merged Matrix size	$26.2 \cdot 10^6$	$41.5 \cdot 10^6$	$36.6 \cdot 10^6$
Light Merged Matrix density	100	100	100
Iteration time on 140 cores	1.73 sec	2.86 sec	0.59 sec
Rank after triangulation	$12.5 \cdot 10^6$	$19 \cdot 10^6$	$18 \cdot 10^6$
# Iterations	$18.8 \cdot 10^6$	$28.5 \cdot 10^6$	$27 \cdot 10^6$
Total time (core·y)	144	362	71

Fig. 4. Practical experiments on partial trianguation

We must take into account that in the year 2000, large interconnected clusters used today to perform the linear algebra step were not yet available. This kind of computation was performed on vectorial supercomputers, and the most powerful vectorial machine available in the year 2000 was the Hitachi SR8000-F1 with 112 vectorial nodes at Leibniz-Rechenzentrum with Rmax equals to 1035 GFlops/s [35].

As a vectorial supercomputer has a direct access to the memory without using different cache levels, the light filtering step is limited to perform merges on unknowns of weight 2, leading to a matrix of size $44.3 \cdot 10^6$ and density $d = 42$. We evaluate the time to perform one matrix-vector product to 422 days, using an extrapolation of the results given in [10]. This extrapolation is given in Appendix B.

Although these estimations can be seen as very crude, we can conclude that the pre-computation of the database of discrete logarithm was reachable by a state-level agency after two years and a half of computation around the year 2000, i.e. right after the IPsec standard was issued and recommended that 768-bit was the default key size that must be implemented in every IPsec product.

7 Conclusion

Nowadays, the ComSec agencies recommend to not use public keys of size below 3072 bits in any way for systems based on the hardness of factorization or discrete logarithms. Nevertheless, such small keys are still in use and even 1024-bit keys are very common if not the majority in many practical applications. For the deepest regret of the academic community, old key size standards remain in use long after their withdrawal was recommended.

This paper, along with other recent works [17, 27], provides reliable estimations on the hardness of the discrete logarithm problem for 768-bit key sizes. Barely two years ago, such problem was evaluated to 36,500 core·years in [1] in a context that pushed the authors to provide conservative estimations. Our paper shows that this problem, thanks to some mathematical improvements, can be solved using only 1250 core·years. Moreover, we show that a state-level agency was able to perform the required computations from the year 2000.

This observation questions ourselves on the ability for a state-level agency to perform the pre-computation of the database of discrete logarithm for the 1024-bit strong prime used in the IPsec standard and thus its ability to perform massive decryptions of IPsec communications. According to the conservative estimations given in [1], this requires 45 million core years of computation. Right after the release of [1], and unlike the authors' views, the main tech companies issued patches for their products to remove the use of 512-bit keys, sometimes also for 768-bit keys, but chose to keep 1024-bit keys, pretending that such estimation shows that this size is still secure in practice.

We believe that providing more reliable estimates on the 1024-bit discrete logarithm problem can convince the tech companies to drop the use of 1024-bit keys in their products in the case where these estimates will show that such computation is feasible at a reasonable cost. But before trying to perform this work, we suggest taking into account the following remarks. First, there is still room for mathematical improvements on the number field sieve. As pointed out by [27], some improvements (e.g. the one-side sieving) only arise when the key length is large enough. We can assume that other improvements will be found when we will consider performing computations for 1024-bit problems. Moreover, the use of simulators instead of the computation of real records can help us to understand what a state-level agency is able to do and limiting ourselves to key sizes that are reachable with academic resources prevents us from studying improvements for larger keys. Although the real computation of a factorization or discrete logarithm can be seen as a tremendous waste of computing time, we believe that these records remain useful, at least for proving the accuracy of our simulators.

One can be surprised that so much effort has to be put to convince someone that 1024-bit keys are insecure. But removing 1024-bit keys everywhere is a difficult and risky task, as some old devices still in use do not support larger keys. This is the reason why the main tech companies will engage in this upgrade as late as possible, hoping that most of these old devices will be then replaced, in order to limit the risks of major failures in their systems.

Therefore, we strongly encourage the academic community to put much more effort on evaluating the security of its main cryptographic primitives even for sizes that are well below the current minimal recommendations.

Acknowledgements

The author would like to thank Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann for their helpful and constructive comments that contributed to improving this paper and for providing the computing power used to perform the simulations presented in this paper. We also thank the people who contributed to earlier versions of this work.

References

1. D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security* , Oct. 2015.
2. S. Bai, C. Bouvier, A. Kruppa, and P. Zimmermann. Better polynomials for GNFS. In *Mathematics of Computation, American Mathematical Society*, 85, pp.12. , 2016.
3. F. Bahr, M. Böhm, J. Franke, T. Kleinjung. Factorization of RSA-200. `http://www.loria.fr/ zimmerma/records/rsa200` . May 2005.
4. F. Bahr, J. Franke, and T. Kleinjung, gnfs4linux, Available at `http://mersenneforum.org/showthread.php?p=169889`.
5. D. J. Bernstein. How to find small factors of integers. `http://cr.yp.to/papers.html` , june 2002.
6. C. Bouvier. The filtering step of discrete logarithm and integer factorization algorithms. `https://hal.inria.fr/hal-00734654`, 2013.
7. C. Bouvier, P. Gaudry, L. Imbert, H. Jeljeli, and E. Thomé. Discrete logarithms in GF(p) — 180 digits. E-mail to the NMBRTHRY mailing list, `http://listserv.nodak.edu/archives/nmbrthry.html`, June 2014.
8. The CADO-NFS Development Team. CADO-NFS, An Implementation of the Number Field Sieve Algorithm, 2016. `http://cado-nfs.gforge.inria.fr/`
9. S. Cavallar, On the number field sieve integer factorization algorithm, Ph.D. thesis, Universiteit Leiden, 2002.
10. S. Cavallar, B. Dodson, A. K. Lenstra, W. Lioen, P. L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, and P. Zimmermann. Factorization of a 512-bit RSA modulus. In B. Preneel, editor, *EUROCRYPT 2000* volume 1807, of *Lecture Notes in Computer Science*, pages 1–18. Springer, Heidelberg, 2000.
11. H. Cohen. A course in computational algebraic number theory. Springer-Verlag, New York, 1993.
12. W. Diffie and M. Hellman. New Directions in Cryptography. In *IEEE Trans. Info. Theor.* 22, 6, pp. 644–654, 1976.
13. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. Blakley and D. Chaum, editors, *Crypto 1984*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, Heidelberg, 1985.
14. FIPS PUB 186-4: Digital Signature Standard (DSS), July 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`
15. J. Franke and T. Kleinjung. Continued fractions and lattice sieving. In *Proceedings of SHARCS 2005.* `www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/talks/FrankeKleinjung.pdf` .

16. J. Franke, T. Kleinjung, F. Morain, and T. Wirth. Proving the primality of very large numbers with fastECPP. I n D. A. Buell, editor, *Algorithmic Number Theory – ANTS-VI* , volume 3076 of *Lecture Notes in Computer Science* , pages 194–207. Springer, Heidelberg, 2004.
17. J. Fried, P. Gaudry, N. Heninger and E. Thomé. A kilobit hidden SNFS discrete logarithm computation. Cryptology ePrint Archive, Report 2016/961, 2016. `http://eprint.iacr.org/`.
18. A. Joux and R. Lercier. Discrete logarithms in GF(p) — 130 digits. E-mail to the NMBRTHRY mailing list,
`http://listserv.nodak.edu/archives/nmbrthry.html`, June 2005.
19. A. Joux and R. Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the Gaussian integer method. *Mathematics of Computation*, 72(242):953–967 , 2003.
20. A. Joux and C. Pierrot. Nearly sparse linear algebra and application to discrete logarithms computations. In A. Canteaut, G. Effinger, S. Huczynska, D. Panario, and L. Storme, eds., *Contemporary Developments in Finite Fields and Applications*, pp. 119–144. World Scientific Publishing Company, 2016.
21. T. Kleinjung. Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024-bit integers. *SCHARCS 2006, Workshop on Special Purpose Hardware for Attacking Cryptographic Systems*, 2006.
22. T. Kleinjung. Discrete logarithms in GF(p) — 160 digits. E-mail to the NMBRTHRY mailing list,
`http://listserv.nodak.edu/archives/nmbrthry.html`, February 2007.
23. T. Kleinjung. Filtering and the matrix step in NFS. *Workshop on Computational Number Theory*, , 2011.
24. T. Kleinjung. Polynomial selection. talk presented at the CADO workshop on integer factorization, `http://cado.gforge.inria.fr/workshop/slides/kleinjung.pdf` , 2008.
25. T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In T. Rabin, editor, *Crypto 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, Heidelberg, 2010.
26. T. Kleinjung, J. W. Bos, and A. K. Lenstra. Mersenne factorization factory. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I* , pages 358-377, 2014.
27. T. Kleinjung, C. Diem, A. K. Lenstra, C. Priplata, and C. Stahlke Computation of a 768-bit prime field discrete logarithm. Cryptology ePrint Archive, Report 2017/067, 2017. `http://eprint.iacr.org/`.
28. IETF. RFC 2409. `https://tools.ietf.org/html/rfc2409` , November 1998.
29. A. K. Lenstra and H. W. Lenstra Jr. The Development of the Number Field Sieve, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, 1993.
30. H. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science* 3 (3):255–269, 1957.
31. J. M. Pollard The lattice sieve. In Lenstra A.K., Lenstra H.W. (eds) The development of the number field sieve. *Lecture Notes in Mathematics*, vol 1554. Springer, Berlin, Heidelberg, 1993.
32. C. Pomerance and J. W. Smith Reduction of huge, sparse matrices over finite fields via created catastrophes *Experiment. Math.*, Volume 1, Issue 2, pp. 89-94, 1992.

33. O. Schirokauer. Virtual logarithms. *J. Algorithms, 57(2):140–147*, 2005.
34. E. Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *J. Symbolic Comput.*, 33(5):757-775, Jul. 2002.
35. Top500: TOP 500 Supercomputer Sites. `http://www.top500.org`.
36. D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32:54–62, 1986.

A A Simulator for the Number Field Sieve

In this paper, we present new types of parameterizations to perform the precomputation of databases of discrete logarithms. The efficiency of these parameterizations depends on the time needed to perform the sieving step and the time to perform the linear algebra step. Making reliable estimations for these two steps requires being able to predict the number of duplicates found during the sieving step, then to predict the size of the final matrix when these parameterizations are used.

We briefly describe here our simulation process and the way we predict the number of duplicates and the size of the final matrix. This simulation process has been tested on various examples and these experiments have shown that the predicted values differ from real computations by less than 5%.

A.1 Generation of a sample of relations

We first randomly select special-q's uniformly in the range $[qmin, qmax]$ of special-q's we have chosen to perform the sieving step. The number of selected special-q's is chosen to generate several thousands of relations so that statistics and extrapolations based on these relations become reliable. Then we use the sieving code on the selected special-q's to generate relations.

The evaluation of the time needed to perform the sieving step can be done by multiplying the number of special-q's found in the range $[qmin, qmax]$ by the average time needed to generate the sample relations for one special-q.

A.2 Simulation of the duplicate removal

Once these relations have been generated, we must take into account that some of them are duplicates: we have to remove a relation from our sample if it may be previously found during a full sieving step, i.e. if this relation contains a different special-q which is lower than the one we have selected to generate our sample.

Note that it is easy to know if a relation will be a duplicate or not: we can check from the decomposition into prime factors of $f(a, b)$ if this relation can be generated by previous special-q's that lie in $[q_1, q[$. If it is the case, we have to check if (a, b) belongs to lattice sieving area or is outside of this area, by computing the reduced base of this lattice and computing the coordinates of (a, b) in this lattice. If at least one factor q' of $f(a, b)$ lies in $[q_1, q[$ and that (a, b)

belongs to the sieving area for this factor, then the relation is discarded as it is a duplicate.

After simulating the duplicate removal, the remaining relations in our sample can be considered as a statistically significant sample of the set of all relations generated by the sieving step.

A.3 Simulation of the original matrix

We create a fake original matrix by generating rows (representing relations) that have properties similar to real relations. The number of rows generated is equal to the mean of unique relations per special-q in our sample multiplied by the number of special-q we want to use in a whole sieve.

Here are the properties we have taken into account to generate each relation, i.e. statistics extracted from our sample and applied to the generation of fake rows of the matrix:

- The number (ℓ_1, ℓ_2) of large primes on each side (not counting the prime special-q).
- The distribution of large primes for each set of relations with exactly (ℓ_1, ℓ_2) large primes.
- The existence of a prime special-q and its expected distribution.
- The number (m_1, m_2) of medium primes lying between M_f (resp. M_g) and F_f (resp. M_g), where M_f is the closest power of two to $F_f/5$, for each set of relations with exactly (ℓ_1, ℓ_2) large primes.
- The distribution of medium primes for each set of relations with exactly (ℓ_1, ℓ_2) large and (m_1, m_2) primes.
- The generation of small primes (below M_f and M_g) uses a random relation from the sample and replaces each small prime by a small prime of the same binary size.

A.4 Simulation of the filtering step and the linear algebra

The fake original matrix is used as input to the filter step using existing code for the traditional filtering step or some new code if we use the partial triangulation. This provides a final matrix whose size and density is very close to the real one.

This fake final matrix can be used to evaluate the overall time of the generation of the Krylov sequence by just running this generation on a few iterations and extrapolating the generation time for the expected number of iterations. As the partial triangulation code for Krylov is not available yet, we use estimations based on assumptions explained in section 5.2.

B Time estimations for the main phases

B.1 Time estimations for the sieving step

The sieving code used in [27] has not been made publicly available yet and we have no access to the clusters at the authors' universities (nor accurate hardware specifications of these clusters) used to perform the sieving step. In this setting, to compare different parameterizations based on core·years on different clusters introduces many biases, as a sieving code depends not only on the processor's frequency but also on cache sizes, RAM speed, NUMA properties, the use of hyper-threading or CPU binding.

Most of the time in the sieving step is spent in two main parts:

- The first one consists in computing the vectors for each member of the factor base used to perform the sieving by vectors. This part does not depend on the size of the sieving area but only on the size of the factor base. As we have chosen to keep the same factor base, the timing for this part is constant.
- The other one consists of scheduling hits, sieving small primes, applying the scheduled hits and finding candidates. This part is linear with respect to the size of the sieving area when the size of the factor base is fixed.

A practical experiment show that for $I = 2^{16}$ (corresponding to $A = 31$ in [27]), about 20% of the time is spent on the first part and 80% on the second part, thus estimates the time spent for one special-q in the sieving step when the sieving area is equal to 2^A as :

$$t(A) = K(20 + 80 \times 2^{A-31})$$

The scaling factor K must be set to $K = 0.2485$ seconds in order to recover the value $t(38) = 2550$ seconds found in [27]. Note that in [27], the factor base bound is reduced for the smaller values of q, leading to smaller times than expected for $A = 39$ and $A = 40$.

B.2 Time estimations for the Krylov phase

Estimations for the Krylov phase, which is the most consuming part of the Block Wiedemann algorithm, can be done by running a few iterations on the simulated matrix. Our estimations are done on 8 nodes of the CATREL cluster at LORIA, where each node holds two Xeon E5-2650 processors with 16 cores at 2.4GHz, linked with Infiniband technology that provides a full-duplex intercommunication at 56 Gb per second, and using the CADO-NFS code (version used in [17]) to perform this phase. Each second spent on that cluster corresponds to $S = 16 \times 8 \times 2.4/2.2 = 140$ seconds on a single core at 2.2 Ghz, used as standard in [27]. We can extract from practical experiments on various matrices that the time to perform a single iteration on a $R \times U$ matrix with row density d can be modeled by $T = S \times (t_c + t_m + t_x)$, where :

- $t_c = K_c \cdot d \cdot R$ (Computing time)
- $t_m = K_m \cdot R \cdot U$ (Cache misses time)
- $t_x = K_x \cdot R$ (Interconnect time)

The three constants $K_c = 4.77e-10$, $K_m = 8.29e-17$, $K_x = 1.78e-8$ have been computed using practical experiments on various matrices of various sizes and densities.

Thanks to this model, we find that the generation of the Krylov sequence for the final matrix of [27], with size $R = 23.5 \cdot 10^6$ and density $d = 134$, using Block Wiedemann parameters μ and ν such that $\mu = 2\nu$ (i.e. the number of iterations is close to $3R/2$), costs 308 core·years using the CADO-NFS code on 8 nodes of the CATREL cluster.

B.3 Time estimations for the Krylov phase on a Y2K supercomputer

We estimate here the time needed to perform the Krylov step on the Hitachi SR8000-F1 with 112 vectorial nodes at Leibniz-Rechenzentrum. According to [35], the performance of this computer is estimated with the parameter Rmax equals to 1035 GFlops/s. This estimation is done considering that the supercomputer Cray Y-MP C916 used in [10], with Rmax equals to 13.7 GFlops/s, has run the Lanczos algorithm on a $6.7 \cdot 10^6$ matrix with density $d = 62$ in 9.5 days. The Lanczos algorithm computes on average 63.24 orthogonal vectors per iteration, where one iteration consists in two multiplications of the matrix by a 64-bit vector (and other computations that are neglected here). As the number of orthogonal vectors one has to compute is equal to the rank of the matrix, the multiplication has been performed $2 \times 6.7 \cdot 10^6/63.24 = 211892$ times in [10], so one 64-bit matrix-vector product is done in 3.87 seconds. Using this result we can estimate that one 768-bit matrix-vector product can be done in $3.87 \times 44.3/6.7 \times 42/62 \times 768/64 \times 13.7/1035 = 2.75$ seconds on the Hitachi SR8000-F1. With Block Wiedemann parameters $\mu = 16$ and $\nu = 1$, the generation of the Krylov sequence on the matrix of rank equal to $12.5 \cdot 10^6$ can be done in 422 days.