# A kilobit special number field sieve factorization

Kazumaro Aoki[1], Jens Franke[2], Thorsten Kleinjung[2],
Arjen K. Lenstra[3,4], and Dag Arne Osvik[3]

[1] NTT, 3-9-11 Midori-cho, Musashino-shi, Tokyo, 180-8585 Japan
[2] University of Bonn, Department of Mathematics, Beringstraße 1, D-53115 Bonn, Germany
[3] EPFL IC LACAL, INJ 330, Station 14, 1015-Lausanne, Switzerland
[4] Alcatel-Lucent Bell Laboratories, Murray Hill, NJ, U.S.A.

**Abstract.** We describe how we reached a new factoring milestone by completing the first special number field sieve factorization of a number having more than 1024 bits, namely the Mersenne number $2^{1039} - 1$. Although this factorization is orders of magnitude 'easier' than a factorization of a 1024-bit RSA modulus is believed to be, the methods we used to obtain our result shed new light on the feasibility of the latter computation.

## 1 Introduction

Proper RSA security evaluation is one of the key tasks of practitioning cryptologists. This evaluation includes tracking progress in integer factorization. In this note we present a long awaited factoring milestone. More importantly, we consider to what extent the methods we have developed to obtain our result, and which are under constant refinement, may be expected to enable us or others to push factoring capabilities even further.

We have determined the complete factorization of the Mersenne number $2^{1039} - 1$ using the special number field sieve integer factorization method (SNFS). The factor 5080711 was already known, so we obtained the new factorization of the composite 1017-bit number $(2^{1039} - 1)/5080711$. The SNFS, however, cannot take advantage of the factor 5080711. Therefore, the difficulty of our SNFS factoring effort is equivalent to the difficulty of the effort that would be required for a 1039-bit number that is very close to a power of two. This makes our factorization the first SNFS factorization that reaches the 1024-bit milestone. The previous SNFS record was the complete factorization of the 913-bit number $6^{353} - 1$ (cf. [1]).

Factoring an RSA modulus of comparable size would be several orders of magnitude harder. Simply put, this is because RSA moduli require usage of the general number field sieve algorithm (NFS), which runs much slower than the SNFS on numbers of comparable size. It is even the case that factoring a 768-bit RSA modulus would be substantially harder than a 1024-bit 'special' one. For that reason we chose to first attempt a 1024-bit SNFS factorization, as presented in this paper, before embarking on a much harder 768-bit RSA modulus using NFS. We point out that a 768-bit NFS factorization will prove to be more helpful than our present 1039-bit SNFS factorization to assess the difficulty of factoring a 1024-bit RSA modulus.

The aspects of our effort where we made most progress, and where our effort distinguishes itself most from previous factoring work such as the previous (913-bit) SNFS record, apply equally well to NFS as they apply to SNFS. They will therefore also have an effect on the assessment of feasibility of NFS-based factorizations such as those of RSA moduli. This need for re-assessment is the main reason that we feel that our result should be reported in the cryptologic literature. For more information on this point see below under '**Matrix**'.

Descriptions of the SNFS and NFS catering to almost all levels of understanding are scattered all over the literature and the web (cf. [16]). There is no need to duplicate any of these previous efforts for the purposes of the present paper. Although familiarity with sieving methods is helpful to fully appreciate all details, for an adequate understanding of the main points it suffices to know that both SNFS and NFS consist of the following major steps (cf. [10]).

**Polynomial selection.** Decide on polynomials to sieve with. For SNFS this does not require any computational effort, for NFS it pays off to spend a considerable effort to find 'good' polynomials. Since we factored $2^{1039} - 1$ using the SNFS our choice was easy and is reported in Section 3.

**Sieving.** For appropriately chosen parameters, perform the sieving step to find sufficiently many *relations*. Though finding enough relations is the major computational task, it can be done in embarrassingly parallel fashion. All relevant data for our effort are reported in Section 3.

**Filtering.** Filter the relations to produce a matrix. See Section 4 for the effort involved in our case.

**Matrix.** Find linear dependencies modulo 2 among the rows of the matrix. In theory, and asymptotically, this requires an effort comparable to the sieving step. For numbers in our current range of interest, however, the amount of computing time required for the matrix step is a fraction of the time required for the sieving step. Nevertheless, and to some possibly surprisingly, the matrix step normally constitutes the bottleneck of large factorization efforts. This is caused by the fact that it does not seem to allow the same level of parallelization as the sieving step. So far, the matrix step has, by necessity, been carried out at a single location and requires many weeks, if not months, of dedicated computing time on a tightly coupled full cluster (typically consisting of on the order of a hundred compute nodes). Consequently, our matrix-handling capabilities were limited by accessibility and availability of large single clusters.

The major point where our effort distinguishes itself from previous work is that we did the matrix step in parallel as four *independent* jobs on different clusters at various locations. This was made possible by using Coppersmith's block Wiedemann algorithm [7] instead of the block Lanczos method [6]. Further work and fine-tuning in this area can have a major impact on what can realistically be achieved, matrix-wise, and therefore factoring-wise: as implied by what was mentioned before, the effort required for the sieving step is not what practically limited our factoring capabilities, it was limited by the matrix step. The details of the new matrix step are reported in Section 5.

**Square root.** For each dependency in turn a square root calculation in a certain number field is performed, until the factorization is found (which happens for each dependency with probability $\geq 1/2$, independent of the other dependencies). The details, and the resulting factorization, are reported in Section 6.

Sections 3 through 6, with contents related to our factorization of $2^{1039}-1$ as indicated above, are followed by a discussion of the wider consequences of our approach in Section 7. Furthermore, in Section 2 we describe how the number $2^{1039}-1$ was selected as the target number for our kilobit SNFS attempt.

Throughout this paper $M$ and $G$ denote $10^6$ and $10^9$, respectively, and logarithms are natural.

## 2 Selecting a kilobit SNFS target number

Once the decision had been reached to attempt a kilobit SNFS factorization by a joint effort, it remained to find a suitable target number to factor. In this section we describe the process that led to our choice of $2^{1039}-1$.

Regular RSA moduli were ruled out, since in general they will not have the special form required for SNFS. Special form numbers, however, are not especially concocted to have two factors of approximately the same size, and have factors of a priori unknown sizes. In particular, they may have factors that could relatively easily be found using factoring methods different from SNFS, such as Pollard's $p-1$ or $\rho$ method, or the elliptic curve method (ECM, cf. [12]). Thus, for all kilobit special form numbers under consideration, we first spent a considerable ECM effort to increase our confidence that the number we would eventually settle for would not turn out to have an undesirably small factor, i.e., a factor that could have been found easier using, for instance, ECM.

Of the candidates that we tried, a 304-digit factor of $10^{371}-1$ turned out to have a 50-digit prime factor (found by ECM after 2,652 curves with first phase bound $43M$), for a 306-digit factor of the number known as 2,2062M a 47-digit factor was found (by ECM, after 4,094 curves with the same bound), for a 307-digit factor of 2,2038M a 49-digit factor was found (ECM with 5,490 curves and same bound), and $10^{311} - 1$ was similarly ruled out after ECM found a 64-digit factor (11,214 curves with $850M$ as first phase bound and corresponding GMP-ECM 6.0 default second phase bound $12{,}530G$, cf. [2]).

The 307-digit number $(2^{1039} - 1)/5080711$ withstood all our ECM efforts: 1,472 curves with first and second phase bounds $850M$ and $12{,}530G$, respectively, and 256,599 curves with bounds $1{,}100M$ and $2{,}480G$, failed to turn up a factor. This calculation was carried out on idle PCs at NTT. It would have required more than 125 years on a single Opteron 2.2GHz with 4GB RAM. Based on the number of curves and the bounds used, it is estimated that a 65-digit factor would be missed with probability about 3.4%, a 70-digit one with probability 53.2%, and an 80-digit factor with probability 98.2%. Given the ECM failure and the substantial effort spent on it, we settled for the 307-digit factor of $2^{1039} - 1$ for our kilobit SNFS factorization attempt.

The software used for the ECM attempt was GMP-ECM 6.0 [19] and Prime95 24.14 [17] on a variety of platforms.

## 3 Parameter selection and sieving

In this section we present the polynomials that we used for the SNFS factorization of $2^{1039} - 1$ and give a superficial description of the sieving step.

With $1039 = 1 + 6 \cdot 173$ it follows that the polynomials $g(X) = X - 2^{173}$ and $f(X) = 2X^6 - 1$ have the root $2^{173}$ in common modulo $2^{1039} - 1$. As customary, everything related to $g(X)$ is referred to as the 'rational side', as opposed to the 'algebraic side' for $f(X)$. In the sieving step we find sufficiently many relations: coprime integers $a$, $b$ with $b \geq 0$ such that both norms $bg(a/b) = a - 2^{173}b$ and $b^6 f(a/b) = 2a^6 - b^6$ have only small prime factors. Here 'sufficiently many' depends on the meaning of 'small'. What we deem to be 'small' depends in the first place on the memory sizes of the machines used for sieving and on the matrix size that we should be aiming for given what matrix size we think we can handle. This means that 'small' cannot be too large. In the second place, the expected time until we have enough relations should be acceptable too, which implies that 'small' cannot be too small either. The choice made always involves this trade-off and is given below. The theoretical justification, and parameter choice, can be found in the NFS literature (cf. [10]).

To find relations we used so-called special $q$'s on the rational side combined with lattice sieving: primes $q$ dividing $bg(a/b)$, such that each $q$ leads to an index $q$ sublattice $L_q$ of $\mathbb{Z}^2$. Most of the $40M$ special $q$'s between $123M$ and $911M$ were used (though the results of some small regions of $q$'s were for organizational reasons not included in the later steps). For most special $q$'s the rectangular region of size $2^{16} \times 2^{15}$ in the upper half plane of $L_q$ was sieved via lattice sieving. For the special $q$'s smaller than $300M$ this was done with factor bases consisting of all (prime, root) pairs for all primes up to $300M$ on the algebraic side and all primes $\leq 0.9q$ on the rational side, but up to $300M$ on both sides for the special $q$'s larger than $300M$. Running our lattice siever with these parameters required approximately 1GB RAM, which was available on most machines we were using. A small fraction of the special $q$'s was used on machines with smaller amounts of memory with factor base bounds of $120M$ on both sides. Large primes (i.e., factors beyond the factor base bounds) up to $2^{38}$ were accepted on both sides, without trying hard to find anything larger than $2^{36}$ and casting aside cofactors larger than $2^{105}$. Also, cofactor pairs were not considered for which the quotient of the probability of obtaining a relation and the time spent on factoring was below a certain threshold, as described in [9].

After a period of about 6 months, at first using PCs and clusters at NTT and the University of Bonn, but later joined by clusters at EPFL, we had collected $16,570,808,010$ relations. Of these relations, 84.1% were found at NTT, 8.3% at EPFL, and 7.6% at the University of Bonn. The total CPU time would be 95 years when scaled to a 3GHz (dual core) Pentium D, or about 100 years on a 2.2GHz Athlon64/Opteron. This boils to 190 Pentium D core years and to about 2.5 relations per seconds per core. The relations required more than a terabyte of diskspace, with copies held at NTT, EPFL, and the University of Bonn.

We used the sieving software from [8].

## 4 Filtering

Because of the special $q$'s the raw data as produced by the sieving step will contain a considerable number of duplicates. Before doing the complete sieving step we had estimated the number of duplicates as follows. We did lattice sieving for a tiny fraction, say $\frac{1}{t}$, of special $q$'s, uniformly distributed over the special $q$ range that we roughly expected to process. For each relation $r$ (corresponding to $(a, b)$) obtained in this way, we computed how often it will be generated in the sieving step. Denote this number by $\mu(r)$. In an ideal situation $\mu(r)$ can be calculated as follows. First, one checks for each prime in the factorization of $bg(\frac{a}{b})$ whether it is in the special $q$ range, i.e., whether it is a potential special $q$ producing this relation. Secondly, for each such potential special $q$ one checks whether the point $(a, b)$ would be in the sieving region for this special $q$, and if it passed this test, whether the cofactor bounds are kept.

Since a lot of approximations are made in the sieving process, the true $\mu(r)$ might be a bit smaller.

The expected number of relations for the complete special $q$ range is $t\sum_r 1$, and the estimated number of unique relations is $t\sum_r \frac{1}{\mu(r)}$. Note that by possibly overestimating $\mu(r)$ we underestimate the number of unique relations. Doing this calculation for 99 of the special $q$'s and the sieving parameters that we actually used, we expected that slightly more than one sixth (16.73%) of the relations found would be duplicates. It turned out that just a little less than one sixth of the relations (namely $2,748,064,961$ for 16.58%) were identified as duplicates. This resulted in a uniqued set of $13,822,743,049$ relations. Identifying and removing the duplicates took less than ten days on two 2GHz Opterons with 4GB RAM each.

Next the singletons were removed: these are relations in which a prime or (prime, root) pair occurs that does not occur in any other relation. This step is combined with the search for cliques, i.e., combinations of the relations where the large primes match up, as fully described in [4]. This took less than 4 days on single cores of 113 3GHz Pentium D processors. Finally, the same hardware was used for 69 hours for a final filtering step that produced a $66,718,354 \times 66,718,154$ matrix of total weight $9,538,688,635$.

Overall the CPU time required to produce the matrix from the raw relations was less than 2 years on a 3GHz Pentium D. It was completed in less than a week, since most of the uniqueing was done during the sieving.

As usual we did some 'over-sieving', i.e., a smaller number of relations sufficed to produce an over-square, but harder to solve, matrix. More specifically, at $14.32G$ relations (of which $12.34G$ were unique) we found an $82,848,491 \times 82,848,291$ matrix of weight $10,003,376,265$, but this matrix was obtained using suboptimal settings and the relations involving 38-bit primes were not used. At $15.61G$ relations ($13.22G$ unique), using better settings and all relations found, we obtained a $71,573,531 \times 71,773,331$ matrix of weight $9,681,804,348$. We do not know at which point precisely we had enough relations to build a matrix. But from our figures it follows that, since $2 * 2^{38}/\log(2^{38}) \approx 20.9G$, finding $0.68 * 2 * \pi(2^{38})$ (non-uniqued) relations sufficed to construct a matrix. This low value 0.68 compared to previous efforts is due to the relatively large bound $2^{38}$ on the large primes.

## 5 The matrix step

In the matrix step linear dependencies modulo 2 among the rows of the $66,718,354 \times 66,718,154$ matrix were sought. This was done using the block Wiedemann algorithm with block length 4 times 64. The details of this algorithm are described in Section 5.1

below. It resulted in 50 dependencies which gave, after quadratic characters tests, 47 useful solutions. A partial explanation of why we got only 50 dependencies as opposed to the expected 200 ones can be found in Section 5.2.

The major part of the calculation (the matrix×vector multiplies, cf. steps 2 and 4 in Section 5.1 below) was carried out in parallel on a cluster of 110 3GHz Pentium D processors (with 2 cores per processor) at NTT and a cluster of 96 2.66 GHz Dual Core2Duo processors (with 4 cores per node) at EPFL. On the latter cluster one or two jobs were run on a varying number of the 96 processors. Scaled to the processors involved, the entire computation would have required 59 days on the Pentium cluster, which is 35 Pentium D core years, or 162 days on 32 nodes of the other cluster, i.e., 56 Dual Core2Duo core years. It should be noted that each of two parallel jobs running on the Pentium D cluster ran about 1.5 times slower than a single job, whereas the load was about 1. This seems to indicate that the same wall-clock time can be achieved on a cluster of 110 single core 3GHz Pentium Prescott processors on a similar network. The relatively poor performance of the cluster at EPFL is probably caused by the fact that the four cores per Dual Core2Duo node share a single network connection. The cluster at NTT has torus topology and the nodes are connected with gigabit ethernet. Transferring intermediate data between NTT and EPFL took about half a day over the Internet.

The computation took place over a period of 69 days, due to several periods of inactivity caused by a variety of circumstances. In principle it could have been done in less than 59 days: if we would have done everything at NTT under ideal conditions (no inactivity), it would take 59 days, but if we would have used both clusters under ideal conditions it should take less time. The software we used for the matrix step was written by the second and third author.

A relatively minor step of the calculation (the Berlekamp-Massey step, cf. step 3 in Section 5.1 below) took 8 hours on 64 cores at the University of Bonn. On 72 cores at EPFL it took a bit less than 7 hours.

### 5.1 The block Wiedemann algorithm

We give a brief description of the block Wiedemann algorithm (see [7], and for the Berlekamp-Massey algorithm [18]). Let $B$ be a $d \times d$ matrix over $\mathbb{F}_2$. The block Wiedemann algorithm depends on two parameters $m, n \in \mathbb{N}$ and heuristically finds $n$ solutions of $Bv = 0$. For our matrix $d = 66,178,354$ and we used $m = 512 = 64 \cdot 8$ and $n = 256 = 64 \cdot 4$. It consists of the following five steps (suppressing some technical details):

1. Random vectors $x_1, \dots, x_m$ and $z_1, \dots, z_n$ are chosen and $y_l = Bz_l$ for $l = 1, \dots, n$ are computed. It is possible to choose $x_i$ as unit vectors to simplify the next step.

2. For $i = 1, \dots, \frac{d}{m} + \frac{d}{n} + O(1)$ the scalar products $a_{lk}^{(i)} = \langle x_k, B^i y_l \rangle$ are computed. We used $i \leq 393,216$. Denote the polynomial
$$\sum_i a_{lk}^{(i)} t^i$$
of $n \times m$ matrices over $\mathbb{F}_2$ by $A$.
3. (Berlekamp-Massey step) In this step a polynomial $F$ of $n \times n$ matrices is constructed such that
$$FA = G + t^c E$$
holds with $\deg(F), \deg(G) \leq \frac{d}{n} + O(1)$ and $c = \frac{d}{m} + \frac{d}{n} + O(1)$. For us the values were approximately $\deg(F) = \deg(G) = 260,600$ and $c = 391,000$. Writing $F = \sum_{j=0}^{\deg(F)} f_{lk}^{(j)} t^j$ this is equivalent to the orthogonality of the $n$ vectors
$$\sum_{j,k} f_{lk}^{(j)} B^{\deg(F)-j} y_k \qquad (1 \leq l \leq n)$$
to the vectors $(B^T)^i x_k$, $0 \leq i \leq \frac{d}{m}$, $1 \leq k \leq m$.
4. For $k, l = 1, \dots, n$ the vectors $v_{lk} = \sum_j f_{lk}^{(j)} B^{\deg(F)-j} z_k$ are computed.
5. With high probability $B \cdot \sum_k v_{lk} = 0$ holds for $l = 1, \dots, n$. The vectors $v_l = \sum_k v_{lk}$ for which this holds are output as solutions.

For the complexity analysis the first and the last step can be neglected. The second and the fourth step require $(1 + \frac{n}{m})d + O(1)$ resp. $d + O(1)$ matrix vector multiplications. If the vectors $x_i$ are chosen as unit vectors the scalar product calculations in the second step become trivial. In the fourth step additional computations are required, equivalent to $n^2 d$ additions in $\mathbb{F}_2$. These can be neglected as long as $n$ is much smaller than the square root of the weight of $B$ (which we can assume). In both steps we have to store the matrix $B$ and two auxiliary vectors for doing the multiplications. Additionally, in step four $n$ vectors need to be stored.

For the Berlekamp-Massey step we used the sub-quadratic algorithm from [18] with FFT for polynomial multiplication. Its complexity is $O(\frac{(m+n)^3}{n} d^{1+o(1)})$ and its space requirement is $O(\frac{(m+n)^2}{n} d)$.

For small $m$ and $n$ most of the time is spent in steps 2 and 4. The total number of matrix vector multiplications, namely $(2 + \frac{n}{m})d$, will be minimal for $m \to \infty$. So, $n$ being chosen, $m$ should be chosen as large as possible such that the Berlekamp-Massey step does not dominate the run time resp. space requirements.

The computations in steps 2 and 4 can be parallelized in several ways. First, the calculation of $B^i y_l$ can be done simultaneously for different $l$. These computations are completely independent. Notice that for current computers there is almost no difference in doing one or, e.g., 64 such computations. So, we might set $n = 64n'$ and do the computations on $n'$ independent computers or clusters thereof. We used $n' = 4$ and ran the 4 computations on two clusters, sometimes 2 jobs in parallel per cluster. This ability to spread the computation across different clusters is the crucial difference between our block Wiedemann approach and many previous factoring efforts that relied on the block Lanczos method [6, 13]. Unlike block Wiedemann, block Lanczos does not allow this type of independent distribution, roughly speaking because it requires the inversion of an $n \times n$ matrix modulo 2 per iteration, which would obviously lead to considerable communication and synchronization issues when run at different locations.

Second, the calculation of $Bv$ for a vector $v$ can be parallelized. As opposed to the above, this requires a lot of communication. More precisely, for a cluster with $n_1 \times n_2$ nodes in a torus topology the communication required for one multiplication is approximately $\frac{d}{n_1} + \frac{d}{n_2}$ per node. When $n_1$ and $n_2$ are chosen approximately equal, the communication costs deteriorate as the square root $\sqrt{n_1 n_2}$ of the number of participating nodes. At NTT we mostly used $n_1 = 11$ and $n_2 = 10$. At EPFL we used $8 \times 8$ on 64 cores (sometimes two simultaneous jobs totalling 128 cores, i.e., 32 processors), $10 \times 8$ on 80 cores, and $12 \times 12$ on 144. Lower numbers of cores were noticeably more efficient per core: when going from 64 to 144 cores we did not get a speed-up of more than 100% (as one would hope for when increasing the number of cores by more than 100%), but only a speed-up of approximately 50%. Roughly, in steps 2 and 4, a third of the time was spend on computation and two-thirds on communication.

A wider collaboration would lead to a larger $n'$ and thus larger $n$ and $m$. Given currently available hardware and the fact that we used a little more than 128GB of memory to run the Berlekamp-Massey step with our parameters, it might be possible to increase $m$ and $n$ by a factor 4. This would increase the run time by a factor 16. Given our 8 hours on 64 cores, this would result in slightly more than 5 days on existing hardware, which is feasible. Unless a much bigger cluster is used, increasing $m$ and $n$ by larger amounts seems to be difficult at the moment.

Finally, we mention a promising idea that we have experimented with. If approximately the same amounts of time are spent on computation and communication, it is possible to run two different jobs simultaneously on a single cluster, in such a way that one job is computing while the other is communicating, and vice versa. If run as independent—but intertwined—jobs (as we did), this approach requires the matrix to be stored twice. Combining the two chunks in a single job in such a way that they have non-overlapping computational and communication needs would require the matrix to be stored just once.

### 5.2 Only 50 dependencies

As mentioned above, we expected to find 200 dependencies but found only 50. Two independent oversights contributed to this phenomenon, but as far as we currently understand still fail to fully explain it.

In the first place an error was uncovered in the selection of the $z_l$ vectors (cf. Step 1 of the algorithm in Section 5.1) that has a large effect on the number of solutions one may expect to find and that depends on the cluster configuration one is using. In our case this led to a reduction of the dimension of the solution space from 200 to about 34.

Secondly, after close inspection of the input matrix it was found that it contains 37 duplicate rows. Due to the peculiar way their arrangement interacts with the other error, this leads to 54 expected dependencies. Both these problems are easily avoided during future computations.

## 6 The square root step

Each independent solution has a chance of at least 50% to lead to a factorization. The main calculation per solution involves the computation of a square root of a huge algebraic number that factors into small prime ideals whose norms are known. To calculate this square root we used Montgomery's square root method [14] as described in [15] and implemented by Friedrich Bahr as part of his diploma thesis (cf. [3]). The first three solutions all led to the trivial factorization, the fourth one produced the following 80-digit prime factor

55853666619936291260749204658315944968646527018488637648010052346319853288374753

with prime 227-digit cofactor

20758181946442382764570481370359469516293970800739520988120838703792729090324679382343143884144834882534053344769112223028158327696525376091410189105241993899334109711624358962065972167481161749004803659735573409253205425523689

thereby completing the factorization of $2^{1039} - 1$.

Preparing the data for 4 solutions simultaneously took 2 hours, and processing thereafter took 1.8 hours per solution, all run times on a 2.2GHz Opteron.

Note that our attempt to select a special number with a large smallest factor was only partially successful: with more luck we would have found the 80-digit factor using

ECM. To some this result is somewhat disappointing, because an 80-digit factor is considered to be 'small' given the size of the 307-digit composite $(2^{1039}-1)/5080711$ that we factored. Note, however, that the factor-size is irrelevant for our result. Also, as may be infered from the figures presented in Section 2, one may expect to spend much more computing time to find this factor using ECM than we spent on SNFS: we estimate it would require about a million curves with first phase bound $8G$, at a cost of several thousand CPU years and ignoring the very substantial memory demands for the second phase (much more than 4GB RAM). If $(2^{1039}-1)/5080711$ would have had a 70-digit factor, we would have been quite unlucky, a 60-digit factor we should have caught with ECM and we would most likely have selected another 'special' number to factor.

## 7 Discussion

As far as we are aware our factorization is the first kilobit factorization achieved using the special number field sieve. It must be stressed, and was already pointed out in the introduction, that our work does not imply that 1024-bit RSA moduli can now be factored by a comparable effort. Quite on the contrary, according to all information available to us, and as far as we know to anyone else in the open community, factoring a 1024-bit RSA modulus is still beyond the capabilities of anyone with resources a few orders of magnitude larger than ours. We estimate that the effort we spent would suffice to factor a 700-bit RSA modulus.

Nevertheless, our work showed that one major hurdle is not as unsurmountable as some thought it would be: unlike previous efforts we managed to distribute the major computation of the matrix step into 4 chunks whose completion did not require any interaction. It required a huge data exchange among our three locations. This was enabled by the advancement of the Internet, allowing relatively efficient, economical, and convenient communication among geographically dispersed locations at speeds up to about 100megabits per second. It remains a subject of further research how the adverse effects of wider parallelization can be addressed and how substantially larger chunks could be handled per location. But, the beginning is there, and without any doubt our work will inspire further work in this area and lead to more and better results.

Until our work there were two major factoring milestones on our way to 1024-bit RSA moduli. One of these milestones, a kilobit SNFS factorization, is now behind us. The next one, and the only remaining major milestone before we would face 1024-bit RSA moduli, is the factorization of a 768-bit RSA modulus. We have no doubt that 768-bit RSA moduli are firmly within our reach, both as far as sieving effort and size of the matrix problem are concerned. If it would indeed be reached, as is now safe to predict, factoring a 1024-bit RSA modulus would begin to dawn on the horizon of what is practically possible for the open community.

It is unclear how long it will take to get there. But given the progress we keep making, and given that we consistently keep reaching our factoring milestones, it would be unwise to have much faith in the security of 1024-bit RSA moduli for more than a few years to come. To illustrate, substantiate, and quantify this remark, note that the first published factorization of a 512-bit RSA modulus is less than a decade ago (cf. [5]) and that

$$\frac{T(1024)}{T(768)} < \frac{1}{5} \times \frac{T(768)}{T(512)},$$

where

$$T(b) = \exp(1.923 \ln(2^b)^{1/3} (\ln(\ln(2^b)))^{2/3})$$

is a rough growth rate estimate for the run time of NFS when applied to a $b$-bit RSA modulus (cf. [11]). A more precise estimate, involving the $o(1)$ which we omitted in $T(b)$, would result in a value that is even smaller than $\frac{1}{5}$. This means that by the time we manage to factor a 768-bit RSA modulus—something we are convinced we are able to pull off—the relative effort of factoring a 1024-bit RSA modulus will look at least 5 times easier than the relative effort of factoring a 768-bit RSA modulus compared to a 512-bit one. As a final remark we note that since 1989 we have seen no major progress in factoring algorithms that can be run on existing hardware, but just a constant stream of refinements. There is every reason to expect that this type of progress will continue.

## References

1. K. Aoki, Y. Kida, T. Shimoyama, H. Ueda, `http://www.crypto-world.com/announcements/SNFS274.txt`.
2. K. Aoki, T. Shimoyama, R311 is factored by ECM, Proceedings of SCIS 2004, no.2E1-1, Hiroshima, Japan, Technical Group on Information Security (IEICE) (in Japanese).
3. F. Bahr, Liniensieben und Quadratwurzelberechnung für das Zahlkörpersieb, University of Bonn, 2005.
4. S. Cavallar, Strategies for filtering in the number field sieve, proceedings ANTS IV, Springer-Verlag, LNCS 1838, 209–231.
5. S. Cavallar, B. Dodson, A. K. Lenstra, P. Leyland, P. L. Montgomery, B. Murphy, H. te Riele, P. Zimmermann, et al., Factoring a 512-bit RSA modulus, Proceedings Eurocrypt 2000, Springer-Verlag, LNCS 1807, 1–18.
6. D. Coppersmith, Solving linear equations over GF(2): block Lanczos algorithm, Linear algebra and its applications 192 (1993), 33–60.
7. D. Coppersmith, Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm, Math. of Comp. 62 (1994), 333–350.

8. J. Franke, T. Kleinjung, Continued fractions and lattice sieving; proceedings SHARCS 2005; http://www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/talks/FrankeKleinjung.pdf.
9. T. Kleinjung, Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024-bit integers; proceedings SHARCS 2006; http://www.hyperelliptic.org/tanja/SHARCS/talks06/thorsten.pdf.
10. A. K. Lenstra, H. W. Lenstra, The development of the number field sieve, Springer-Verlag, LNM 1554, 1993.
11. A. K. Lenstra, E. R. Verheul, Selecting cryptographic key sizes, J. of Cryptology 14 (2001), 255–293.
12. H. W. Lenstra, Factoring integers with elliptic curves, Ann. of Math. 126 (1987), 649–673.
13. P. L. Montgomery, A block Lanczos algorithm for finding dependencies over $GF(2)$, Proceedings Eurocrypt'95, Springer-Verlag, LNCS 921, 106–120.
14. P. L. Montgomery, Square roots of products of algebraic numbers, http://ftp.cwi.nl/pub/pmontgom/sqrt.ps.gz.
15. P. Nguyen, A Montgomery-like square root for the number field sieve, Proceedings ANTS III, Springer-Verlag, LNCS 1423, 151–168.
16. C. Pomerance, A tale of two sieves, http://www.ams.org/notices/199612/pomerance.pdf.
17. Prime95, http://www.mersenne.org/freesoft.htm.
18. E. Thomé, Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm, Journal of symbolic computation 33 (2002), 757–775.
19. P. Zimmermann, http://gforge.inria.fr/projects/ecm/.