Factorization of a 768-bit RSA modulus

version 1.0, January 7, 2010

Thorsten Kleinjung[1],
Kazumaro Aoki[2], Jens Franke[3], Arjen K. Lenstra[1], Emmanuel Thomé[4],
Joppe W. Bos[1], Pierrick Gaudry[4], Alexander Kruppa[4], Peter L. Montgomery[5,6],
Dag Arne Osvik[1], Herman te Riele[6], Andrey Timofeev[6], and Paul Zimmermann[4]

[1] EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland
[2] NTT, 3-9-11 Midori-cho, Musashino-shi, Tokyo, 180-8585 Japan
[3] University of Bonn, Department of Mathematics, Beringstraße 1, D-53115 Bonn, Germany
[4] INRIA CNRS LORIA, Équipe CARAMEL - bâtiment A, 615 rue du jardin botanique,
F-54602 Villers-lès-Nancy Cedex, France
[5] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
[6] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract. This paper reports on the factorization of the 768-bit number RSA-768 by the number field sieve factoring method and discusses some implications for RSA.
Keywords: RSA, number field sieve

1 Introduction

On December 12, 2009, we factored the 768-bit, 232-digit number RSA-768 by the number field sieve (NFS, [19]). The number RSA-768 was taken from the now obsolete RSA Challenge list [37] as a representative 768-bit RSA modulus (cf. [36]). This result is a record for factoring general integers. Factoring a 1024-bit RSA modulus would be about a thousand times harder, and a 768-bit RSA modulus is several thousands times harder to factor than a 512-bit one. Because the first factorization of a 512-bit RSA modulus was reported only a decade ago (cf. [7]) it is not unreasonable to expect that 1024-bit RSA moduli can be factored well within the next decade by an academic effort such as ours or the one in [7]. Thus, it would be prudent to phase out usage of 1024-bit RSA within the next three to four years.

The previous record NFS factorization was that of the 663-bit, 200-digit number RSA-200 (cf. [4]), established on May 9, 2005. That 663-bit and the current 768-bit record NFS factorizations should not be confused with record special NFS (SNFS) factorizations, the most recent of which was the complete factorization of the 1039-bit number $2^{1039} - 1$, obtained in the spring of 2007 (cf. [2]). Although $2^{1039} - 1$ is much bigger than RSA-768, its special form made it an order of magnitude easier to factor than RSA-768.

The following effort was involved. We spent half a year on 80 processors on polynomial selection. This was about 3% of the main task, the *sieving*, which was done on many hundreds of machines and took almost two years. On a single core 2.2 GHz AMD Opteron processor with 2 GB RAM per core, sieving would have taken about fifteen hundred years. This included a generous amount of *oversieving*, to make the most cumbersome step, the *matrix step*, more manageable. Preparing the sieving data for the matrix step took a couple of weeks on a few

processors, the final step after the matrix step took less than half a day of computing, but took about four days of intensive labor because a few bugs had to be fixed.

It turned out that we had done about twice the sieving strictly necessary to obtain a usable matrix, and that the extra data allowed generation of a matrix that was quite a bit easier than anticipated at the outset of the project. Although we spent more computer time on the sieving than required, sieving is a rather laid back process that, once running, does not require much care beyond occasionally restarting a machine. The matrix step, on the other hand, is a more subtle affair where a slight disturbance easily causes major trouble, in particular if the problem is by its sheer size stretching the available resources. Thus, our approach to overspend on an easygoing part of the computation led to a matrix that could be handled relatively smoothly, thereby saving us considerable headaches. More importantly, and another reason behind the oversieving, the extra sieving data allow us to conduct various experiments aimed at getting a better understanding about the relation between sieving and matrix efforts and the effect on NFS feasibility and overall performance. This is ongoing research, the results of which will be reported elsewhere. All in all, the extra sieving cycles were well spent.

For $2^{1039}-1$ the matrix step was, for the first time, performed on a number of different clusters: as reported in [2], the main part of the calculation was run in parallel at two locations on four clusters. This is possible due to our usage of the block Wiedemann algorithm for the matrix step (cf. [9]), where the main calculation consists of two consecutive sequences of matrix times vector multiplications. As discussed in [2], a greater degree of independent parallelization leads to a variety of challenges that need to be overcome before larger problems can be tackled (such as, for instance, 1024-bit moduli). Here we set a first step toward a scalable solution by showing how greater flexibility can be achieved in the number and contributions of the various independent clusters. As a result a nine times harder (than for $2^{1039}-1$) matrix step was solved in less than four months running up to eight independent jobs on clusters located in France, Japan, and Switzerland. During one of the substeps about a terabyte of memory was needed. These figures imply that much larger matrices are already within reach, leaving preciously little doubt about the feasibility by the year 2020 of a matrix required for a 1024-bit NFS factorization. As part of the experiments mentioned above we also intend to study if a single large cluster would be able to handle such matrices using the block Lanczos algorithm (cf. [8]). Compared to block Wiedemann this has advantages (a shorter, single sequence of iterations and no tedious and memory-hungry central Berlekamp-Massey step [40]) but disadvantages as well (it cannot be run on separate clusters and each iteration consists of a multiplication by both a matrix and its transpose).

The factorization problem and the steps taken to solve it using NFS are described in Section 2. The factorization is presented in Section 2.5. Implications for moduli larger than RSA-768 are briefly discussed in the concluding Section 3. Appendix A presents some of the details of the sieving approach that we used, and Appendix B describes a new twist of the block Wiedemann algorithm that makes it easier to share large calculations among different parties.

There are many aspects to an effort as described in this paper that are not interesting from a scientific or research point of view, but that are nevertheless crucial for its success. Although NFS can be run as a BOINC project (cf. [30]), we chose not to do so. Instead, we gathered a small but knowledgeable group of contributors who upfront committed themselves to finish the project, while dedicating at least a known minimum of computational resources for a substantial period of time. This allowed us to target a reasonable completion date, a date

that we managed to meet easily despite our ample oversieving. Although different sieving-clients do not need to communicate, each client needs to communicate a fair amount of data to the central storage location. This required a bit more organizational efforts than expected, occasional recovery from mishaps such as unplugged network cables, switched off servers, or faulty raids, and a constantly growing farm of backup drives. We do not further comment on these more managerial issues in this article, but note that larger efforts of this sort would benefit from full-time professional supervision.

2 Factoring RSA-768

2.1 Factoring using the Morrison-Brillhart approach

A composite integer n can be factored by finding integer solutions x, y of the *congruence of squares* $x^2 \equiv y^2 \bmod n$, and by hoping that n is factored by writing it as $\gcd(x - y, n) \cdot \gcd(x + y, n)$. For a random such pair the probability is at least $\frac{1}{2}$ that a non-trivial factor of n is found in this way. The Morrison-Brillhart approach [25] solves the equation $x^2 \equiv y^2 \bmod n$ by combining congruences of smooth squares. In this section this is explained and, very rudimentarily, how NFS applies the same basic idea.

A non-zero integer u is b-smooth if the prime factors of $|u|$ are at most b. Each b-smooth integer u corresponds to the $(\pi(b) + 1)$-dimensional integer vector $\mathfrak{v}(u)$ of exponents of the primes $\leq b$ in its factorization, where $\pi(b)$ is the number of primes $\leq b$ and the "+1" accounts for inclusion of the exponent of -1. The *factor base* consists of the primes at most equal to the smoothness bound.

Let n be a composite integer, b a smoothness bound, and t a positive integer. Suppose that $\pi(b) + 1 + t$ different integers v have been found such that the least absolute remainders $r(v) = v^2 \bmod n$ are b-smooth. Because the corresponding $(\pi(b) + 1)$-dimensional vectors $\mathfrak{v}(r(v))$ are linearly dependent, at least t independent subsets S of the v's can be found using linear algebra such that, for each of these subsets, $\sum_{v \in S} \mathfrak{v}(r(v))$ is an all-even vector. Each subset S then leads to a solution $x = \prod_{v \in S} v$ and $y = \sqrt{\prod_{v \in S} r(v)}$ to $x^2 \equiv y^2 \bmod n$ and thus, overall, t chances of at least $\frac{1}{2}$ to factor n.

If v's as above are generated by picking random integers and hoping that the least absolute remainder modulo n of their square is smooth, the search for the proper v's depends on the smoothness probability of integers of the same order of magnitude as n. That is Dixon's random squares method (cf. [11]), which has the advantage that its expected runtime can rigorously be proved. It was preceded by Morrison and Brillhart (cf. [25]) who had already obtained a much higher smoothness probability, and thus a much faster – but heuristic – factoring method, by using continued fractions to generate quadratic residues modulo n of roughly half the size of n, i.e., of order $n^{1/2}$. Richard Schroeppel with his *linear sieve* was the first, in about 1976, to combine essentially the same high smoothness probability with a fast *sieving*-based method to recognize the smooth numbers (cf. [33, Section 6]). He was also the first to analyze the resulting expected runtime (cf. [17, Section 4.2.6]). Later this culminated in Carl Pomerance's *quadratic sieve* (cf. [33,34]) which, like Schroeppel's method, relies on fast sieving based recognition of smooth quadratic residues of rough order $n^{1/2}$.

Factoring methods of this sort that rely on smoothness of residues of order $n^{\theta(1)}$ can be shown to be stuck at expected runtimes of the form

$$e^{(c+o(1))(\ln n)^{1/2}(\ln \ln n)^{1/2}},$$

for positive constants c and asymptotically for $n \to \infty$. The number field sieve [19] was the first, and so far the only, practical factoring method to break through the barrier of the $\ln n$-exponent of $\frac{1}{2}$. This is achieved by looking at more contrived congruences that involve smoothness of numbers of order $n^{o(1)}$, for $n \to \infty$, that can, as usual, be combined into a congruence of squares $x^2 \equiv y^2 \bmod n$. As a result, the number field sieve factors a composite integer n in heuristic expected time

$$e^{((64/9)^{1/3}+o(1))(\ln n)^{1/3}(\ln \ln n)^{2/3}},$$

asymptotically for $n \to \infty$. It is currently the algorithm of choice to factor numbers without special properties, such as those on the RSA Challenge List [37]. Of this list, we concentrate on the number RSA-768, a 768-bit RSA modulus with 232-digit decimal representation

> 12301866845301177551304949583849627207728535695953347921973224521517264005072636
> 75187452021997864693899564749427740638459251925573263034537315482685079170261
> 22142913461670429214311602221240479274737794080665351419597459856902143413.

Similar to Schroeppel's linear sieve, the most important steps of NFS are *sieving* and the *matrix step.* In the former *relations* are collected, congruences involving smooth values similar to the smooth $r(v)$-values above. In the latter linear dependencies are found among the exponent vectors of the smooth values. Unlike Schroeppel's method, however, NFS requires two non-trivial additional steps: a pre-processing *polynomial selection* step before the sieving can start, and a post-processing *square root step* to convert the linear dependencies into congruences of squares. A rough operational description of these steps as applied to RSA-768 is given in the remainder of this section. For an explanation why these steps work, the reader may consult one of the many expositions on NFS in the literature (cf. [19,20,35]).

2.2 Polynomial selection

Let n be a composite integer to be factored. In NFS relations are given by coprime pairs of integers (a, b) with $b > 0$ such that two integer values $v_1(a, b)$ and $v_2(a, b)$ that depend on a, b are simultaneously smooth, $v_1(a, b)$ with respect to some bound b_1 and $v_2(a, b)$ with respect to some bound b_2. The values $v_1(a, b)$ and $v_2(a, b)$ are defined as follows. Let $f_1(X), f_2(X) \in \mathbf{Z}[X]$ be two irreducible integer polynomials of degrees d_1 and d_2, respectively, with a common root m modulo n, i.e., $f_1(m) \equiv f_2(m) \equiv 0 \bmod n$. Then $v_k(a, b) = b^{d_k} f_k(a/b) \in \mathbf{Z}$. Sufficiently many more than $\pi(b_1) + \pi(b_2) + 2$ relations lead to enough chances to factor n (cf. [19]), as sketched in the next two paragraphs.

Let $\mathbf{Q}(\alpha_k) = \mathbf{Q}[X]/(f_k(X))$ for $k = 1, 2$ be two algebraic number fields. The elements $a - b\alpha_k \in \mathbf{Z}[\alpha_k]$ have *norm* $v_k(a, b)$ and belong to the first degree prime ideals in $\mathbf{Q}(\alpha_k)$ of (prime) norms equal to the prime factors of $v_k(a, b)$. For monic f_k's, these prime ideals in $\mathbf{Q}(\alpha_k)$ correspond bijectively to the pairs $(p, r \bmod p)$ where p is prime and $f_k(r) \equiv 0 \bmod p$: excluding factors

4

of f_k's discriminant, a first degree prime ideal corresponding to such a pair $(p, r \bmod p)$ has norm p and is generated by p and $r - \alpha_k$.

Because $f_k(m) \equiv 0 \bmod n$, there are two natural ring homomorphisms $\phi_k : \mathbf{Z}[\alpha_k] \to \mathbf{Z}/n\mathbf{Z}$ for $k = 1, 2$ that map $\sum_{i=0}^{d_k-1} a_i \alpha_k^i$ (with $a_i \in \mathbf{Z}$) to $\sum_{i=0}^{d_k-1} a_i m^i \bmod n$ and for which $\phi_1(a - b\alpha_1) \equiv \phi_2(a - b\alpha_2) \bmod n$. Finding a linear dependency modulo 2 among the vectors consisting of the exponents of the primes in the b_1-smooth $v_1(a, b)$, b_2-smooth $v_2(a, b)$ pairs (and after, at some stage, slightly extending the vectors to make sure that the square norms lead to squares of algebraic numbers, cf. [1]), subsets S of the set of relations are constructed such that $\prod_{(a,b) \in S}(a - b\alpha_k)$ is a square σ_k in $\mathbf{Q}(\alpha_k)$, both for $k = 1$ and for $k = 2$. With $\phi_1(\sigma_1) \equiv \phi_2(\sigma_2) \bmod n$ it then remains to compute square roots $\tau_k = \sigma_k^{1/2} \in \mathbf{Q}(\alpha_k)$ for $k = 1, 2$ to find a solution $x = \phi_1(\tau_1)$ and $y = \phi_2(\tau_2)$ to $x^2 \equiv y^2 \bmod n$.

It is easy to find polynomials f_1 and f_2 that lead to smoothness of numbers of order $n^{o(1)}$, for $n \to \infty$. Let $d_1 \in \mathbf{N}$ be of order $\left(\frac{3 \ln n}{\ln \ln n}\right)^{1/3}$, let $d_2 = 1$, let m be an integer of order n^{1/d_1}, and let $n = \sum_{i=0}^{d_1} n_i m^i$ with $0 \le n_i < m$ be the radix m representation of n. Then $f_1(X) = \sum_{i=0}^{d_1} n_i X^i$ and $f_2(X) = X - m$ have coefficients that are $n^{o(1)}$ for $n \to \infty$, they have m as common root modulo n, and for the particular choice of d_1 the values a, b that suffice to generate enough relations are small enough to keep $b^{d_1} f_1(a/b)$ and $b^{d_2} f_2(a/b)$ of order $n^{o(1)}$ as well. Finally, if f_1 is not irreducible, it can be used to directly factor n or, if that fails, one of its factors can be used instead of f_1. If $d_1 > 1$ and $d_2 = 1$ we refer to "$k = 1$" as the *algebraic side* and "$k = 2$" as the *rational side*. With $d_2 = 1$ the algebraic number field $\mathbf{Q}(\alpha_2)$ is simply $\mathbf{Q}$, the first degree prime ideals in $\mathbf{Q}$ are the regular primes and, with $f_2(X) = X - m$, the element $a - b\alpha_2$ of $\mathbf{Z}[\alpha_2]$ is $a - bm = v_2(a, b) \in \mathbf{Z}$ with $\phi_2(a - b\alpha_2) = a - bm \bmod n$.

Although with the above polynomials NFS achieves the asymptotic runtime given in the previous section, the amount of freedom in the choice of m, f_1, and f_2 means that from many different choices the best may be selected. Here we keep interpretation of the term "best" intuitive and make no attempt to define it precisely. When comparing different possibilities, however, it is usually obvious which one is better. Most often this is strongly correlated with the smoothness probability over the relevant range of a, b pairs, and thus with the size of the coefficients and the number of roots modulo small primes, smoothness properties of the leading coefficients, and the number of real roots. Sieving experiments may be carried out to break a tie.

Given this somewhat inadequate explanation of what we are looking for, the way one should be looking for good candidates is the subject of active research. Current best methods involve extensive searches, are guided by experience, helped by luck, and profit from patience. Only one method is known that produces two good polynomials of degrees greater than one (namely, twice degree two, cf. [5]), but its results are not competitive with the current best $d_1 > 1$, $d_2 = 1$ methods which are all based on refinements described in [15] of the approach from [24] and [26] as summarized in [7, Section 3.1]. A search of three months on a cluster of 80 Opteron cores (i.e., $\frac{3}{12} \cdot 80 = 20$ core years), conducted in 2005 already and thus not including the idea

from [16], produced three pairs of polynomials of comparable quality. We used the pair

$$
\begin{aligned}
f_1(X) = {} & 265482057982680X^6 \\
& + 1276509360768321888X^5 \\
& - 5006815697800138351796828X^4 \\
& - 46477854471727854271772677450X^3 \\
& + 6525437261935989397109667371894785X^2 \\
& - 18185779352088594356726018862434803054X \\
& - 277565266791543881995216199713801103343120, \\
f_2(X) = {} & 34661003550492501851445829X - 1291187456580021223163547791574810881
\end{aligned}
$$

from which the common root m follows as the negative of the ratio modulo RSA-768 of the constant and leading coefficients of f_2. The leading coefficients factor as $2^3 \cdot 3^2 \cdot 5 \cdot 7^2 \cdot 11 \cdot 17 \cdot 23 \cdot 31 \cdot 112\,877$ and $13 \cdot 37 \cdot 79 \cdot 97 \cdot 103 \cdot 331 \cdot 601 \cdot 619 \cdot 769 \cdot 907 \cdot 1\,063$, respectively. The discriminant of f_1 equals $2^{12} \cdot 3^2 \cdot 5^2 \cdot 13 \cdot 17 \cdot 17\,722\,398\,737 \cdot c273$, where $c273$ denotes a 273-digit composite number that probably does not have a factor of 30 digits or less and that is expected to be square-free. The discriminant of f_2 equals one. A renewed search in the spring of 2007 (also not using the idea from [16]), after we had decided to factor RSA-768, produced a couple of candidates of similar quality, again after spending about 20 core years.

Given the polynomials, smoothness bounds have to be selected. Also, we need to specify a sieving region S of $\mathbf{Z} \times \mathbf{Z}_{>0}$ where the search for relations will be conducted. We conclude this section with a discussion of this final part of the parameter selection process.

During any practical search for smooth numbers, one always encounters near misses: candidates that are smooth with the exception of a small number of prime factors that are somewhat larger than the smoothness bound. Not only can these *large primes*, as these factors are commonly referred to, be recognized at no or little extra cost, they also turn out to be useful in the factorization process. As a result the smoothness bounds can be chosen substantially lower than they would have to be chosen otherwise and relation collection goes faster while requiring less memory. On the negative side, it is somewhat harder to decide whether enough relations have been found – as the simple criterion that more than $\pi(b_1) + \pi(b_2) + 2$ are needed is no longer adequate – but that is an issue of relatively minor concern. In principle the decision requires duplicate removal, repeated singleton removal, and counting, all briefly touched upon at the end of Section 2.3. In practice it is a simple matter of experience.

Let b_ℓ denote the upper bound on the large primes that we want to keep, which we assume to be the same for the algebraic and rational sides. Thus, $b_\ell \geq \max(b_1, b_2)$. We say that a non-zero integer x is (b_k, b_ℓ)-smooth if with the exception of, say, four prime factors between b_k and b_ℓ, all remaining prime factors of $|x|$ are at most b_k. Consequently, we change the definition of a relation into a coprime pair of integers a, b with $b > 0$ such that $b^{d_1} f_1(a/b)$ is (b_1, b_ℓ)-smooth and $b^{d_2} f_2(a/b)$ is (b_2, b_ℓ)-smooth.

Selection of the smoothness bounds b_1, b_2, and b_ℓ, of the search area S for a, b, and how many primes between b_1 or b_2 and b_ℓ will be permitted and looked for is also mostly guided by experience, and supported by sieving experiments to check if the yield is adequate. For RSA-768 we used smoothness bounds $b_1 = 11 \cdot 10^8$, $b_2 = 2 \cdot 10^8$ and $b_\ell = 2^{40}$ on cores with at least 2 GB RAM (which turned out to be the majority). On cores with less memory (but

6

at least a GB RAM, since otherwise it would not be worth our trouble), $b_1 = 4.5 \cdot 10^8$ and $b_2 = 10^8$ were used instead (with the same b_ℓ). For either choice it was expected that it would suffice to use as sieving region the bounded subset S of $\mathbf{Z} \times \mathbf{Z}_{>0}$ containing about $11 \cdot 10^{18}$ coprime pairs a, b with $|a| \leq 3 \cdot 10^9 \cdot \kappa^{1/2} \approx 6.3 \cdot 10^{11}$ and $0 < b \leq 3 \cdot 10^9 / \kappa^{1/2} \approx 1.4 \cdot 10^7$. Here $\kappa = 44\,000$ approximates the *skewness* of the polynomial f_1, and serves to approximately minimize the largest norm $v_1(a, b)$ that can be encountered in the sieving region. Although prime ideal norms up to $b_\ell = 2^{40}$ were accepted, the parameters were optimized for norms up to 2^{37}. Most jobs attempted to split after the sieving algebraic and rational cofactors up to 2^{140} and 2^{110}, respectively, only considering the most promising candidates (cf. [14]). As far as we know, this was the first NFS factorization where more than three large primes were allowed on the algebraic side.

2.3   Sieving

With the parameters as chosen above, we need to find relations: coprime pairs of integers $(a, b) \in S$ such that $b^{d_1} f_1(a/b)$ is (b_1, b_ℓ)-smooth and $b^{d_2} f_2(a/b)$ is (b_2, b_ℓ)-smooth. In this section we describe this process.

A prime p dividing $f_k(r)$ is equivalent to $(r \bmod p)$ being a root of f_k modulo p (as already noted above with the first degree prime ideals, and disregarding prime factors of f_k's discriminant). Because $d_2 = 1$, the polynomial f_2 has one root modulo p for each prime p not dividing the leading coefficient of f_2, and each such prime p divides $f_2(j)$ once every p consecutive values of j. For f_1 the number of roots modulo p may vary from 0 (if f_1 has no roots modulo p) to d_1 (if f_1 has d_1 linear factors modulo p). Thus, some primes do not divide $f_1(j)$ for all j, whereas other primes p may divide $f_1(j)$ a total of d_1 times for every p consecutive j-values. All (prime,root) pairs with prime at most b_1 for f_1 and at most b_2 for f_2 can be precomputed.

Given a (prime,root) pair (p, r) for f_k, the prime p divides $b^{d_k} f_k(a/b)$ if $a/b \equiv r \bmod p$, i.e., for a fixed b whenever $a = rb \bmod p$. This leads to the *line sieving* approach where per b-value for each (p, r) pair for f_1 (or f_2) one marks with "p" all relevant a-values (of the form $rb + ip$ for $i \in \mathbf{Z}$). After this sieving step, the good locations are those that have been marked for many different p's. These locations are remembered, after which the process is repeated for f_2 (or f_1), and the intersection of the new good locations with the old ones is further studied, since those are the locations where relations may be found. With $1.4 \cdot 10^7$ b-values to be processed (the lines) and about $2 \cdot \kappa$ times that many a-values per b, the lines can be partitioned among almost any number of different processors a factoring effort of this limited scale can get access to. This is how the earliest implementations of distributed NFS sieving worked.

A more efficient but also more complicated approach has gained popularity since the mid 1990s: the *lattice sieve* as described in [32]. For a (prime,root) pair $\mathfrak{q} = (q, s)$ define $L_{\mathfrak{q}}$ as the lattice of integer linear combinations of the 2-dimensional integer (row-)vectors $(q, 0), (s, 1) \in \mathbf{Z}^2$, and let $S_{\mathfrak{q}}$ be the intersection of S and $L_{\mathfrak{q}}$. Fix a (prime,root) pair $\mathfrak{q} = (q, s)$ for, say, f_1. The *special prime* q (as this lattice defining large prime was referred to in [32]) is typically chosen of the same order of magnitude as b_1, possibly bigger – but smaller than b_ℓ. It follows that q divides $b^{d_1} f_1(a/b)$ for each $(a, b) \in S_{\mathfrak{q}}$. Lattice sieving consists of marking, for each other (prime,root) pair $\mathfrak{p}$ for f_1 for which the prime is bounded by b_1, the points in the intersection $L_{\mathfrak{p}} \cap S_{\mathfrak{q}}$. The good locations are remembered, after which the process is repeated for all the

(prime,root) pairs for f_2 with prime bounded by b_2, and the intersection of the new good locations with the old ones is further studied. In this way relations in $S_{\mathfrak{q}}$ are found. Thus, for each of these relations it is the case that q divides $v_1(a, b)$, and the lattice sieve is repeated for other (prime,root) pairs $\mathfrak{q}$ until enough relations have been found.

A relation thus found and involving some prime $q' \neq q$ dividing $v_1(a, b)$ may also be found when lattice sieving with $\mathfrak{q}' = (q', s')$. Thus, unlike line sieving, when lattice sieving duplicates will be found: these have to be removed (while, obviously, keeping one copy).

In practice one fixes bounds I and J independent of $\mathfrak{q}$ and defines $S_{\mathfrak{q}} = \{iu + jv : i, j \in \mathbf{Z}, -I/2 \leq i < I/2, 0 < j < J\}$, where u, v form a basis for $L_{\mathfrak{q}}$ that minimizes the norms $v_1(a, b)$ for $(a, b) \in S_{\mathfrak{q}}$. Such a basis is found by partially reducing the initial basis $(q, 0), (s, 1)$ for $L_{\mathfrak{q}}$ such that, roughly speaking, the first coordinate is about κ times bigger than the second, thereby minimizing the norms $v_1(a, b)$ for $(a, b) \in S_{\mathfrak{q}}$, cf. skewness of the sieving region S. Actual sieving is carried out over the set $\{(i, j) \in \mathbf{Z} \times \mathbf{Z}_{>0} : -I/2 \leq i < I/2, 0 < j < J\}$, interpreted as $S_{\mathfrak{q}}$ in the above manner.

For RSA-768 we used $I = 2^{16}$ and $J = 2^{15}$ which implies that we did lattice sieving over an area of size roughly $2^{31} \approx 2 \cdot 10^9$ (we did not use any line sieving). With $b_1 = 11 \cdot 10^8$ and $b_2 = 2 \cdot 10^8$, the majority of the primes that are being sieved with can be expected to hit $S_{\mathfrak{q}}$ only a few times. Thus, for any $\mathfrak{p}$ that one will be sieving with, only a few of the j-values (the lines) will be hit, unlike the situation in plain line sieving where the lines are so long that each line will be hit several times by each prime. It implies that, when lattice sieving, one cannot afford to look at all lines $0 < j < J$ per $\mathfrak{p}$, and that a sieving method more sophisticated than line sieving must be used. This *sieving by vectors*, as it was referred to in [32], was first implemented in [13] and used for many factorizations in the 1990s, such as those reported in [10] and [7]. We used a particularly efficient implementation of sieving by vectors, the details of which have never appeared in print (cf. [12]). They are summarized in Appendix A.

Lattice sieving with sieving by vectors as above was done for most of the about 480 million (prime,root) pairs (q, s) for special qs between 450 million and 11.1 billion (and some special qs below 450 million, with a smaller b_1-value) by eight contributing parties during the period August 2007 until April 2009. Table 1 lists the percentages contributed by the eight participants. Scaled to a 2.2 GHz Opteron core with 2 GB RAM, a single (prime,root) pair (q, s) was processed in less than a hundred seconds on average and produced about 134 relations, for an average of about four relations every three seconds. This average rate varies by a factor of about two between both ends of the special q range that we considered.

In total $64\,334\,489\,730$ relations were collected, each requiring about 150 bytes. Compressed they occupied about 5 terabytes of disk space, backed up at various locations. Of these relations, 27.4% were duplicates. Duplicate removal can simply be done using bucket sorting (on b-values, for instance) followed by application of UNIX commands `sort` and `uniq`, but may be complicated by odd data formats, and may be done in any way one deems convenient. We used hashing. Most uniqueing was done during the sieving. Overall it took less than 10 days on a 2.66 GHz Core2 processor with ten 1TB hard disks. After inclusion of $57\,223\,462$ *free relations* (cf. [19]) the uniqueing resulted in $47\,762\,243\,404$ relations involving $35\,288\,334\,017$ prime ideals.

name of contributor	relations contribution	matrix			
		stage 1	stage 2	stage 3	total
Bonn (University and BSI)	8.14%				
CWI	3.44%				
EPFL	29.78%	34.3%	100%	78.2%	51.9%
INRIA LORIA (ALADDIN-G5K)	37.97%	46.8%		17.3%	35.0%
NTT	15.01%	18.9%		4.5%	13.1%
Scott Contini (Australia)	0.43%				
Paul Leyland (UK)	0.69%				
Heinz Stockinger (Enabling Grids for E-sciencE)	4.54%				

Table 1: Percentages contributed (with matrix stages 1, 2, and 3 contributing $\frac{3}{5}$th, 0, and $\frac{2}{5}$th to total).

Given the set of unique relations, those that involve a prime ideal that does not occur in any other relation, the singletons, cannot be part of a dependency and can thus be removed. Recognizing singletons can be done using a simple hash table with a count (counting "0", "1", "many"). Doing this once reduced the set of relations to $28\,984\,986\,047$ elements, with $14\,498\,007\,183$ prime ideals. However, removal of singletons usually leads to new singletons, and singleton removal must be repeated until no more relations are removed. Note that the first (usually high volume) runs can be made to accept different prime ideals with the same hash to lower memory requirements for the hash table. Once singletons have been removed, we have to make sure that the number of remaining relations is larger than the total number of prime ideals occurring in them, so that there will indeed be dependencies among the exponent vectors. After a few more singleton removals $24\,615\,168\,385$ relations involving at most $9\,976\,671\,468$ prime ideals were left.

At this point further singleton removal iterations were combined with *clique removal*, i.e., search of combinations with matching first degree prime ideals of norms larger than b_k. Removal of all singletons together with clique removal (cf. [6]) led to $2\,458\,248\,361$ relations involving a total of $1\,697\,618\,199$ prime ideals, and still containing an almost insignificant number $604\,423$ of free relations. This not only clearly indicated that we had done enough sieving, but also gave us lots of flexibility while creating the matrix. In the next section it is described how dependencies were found. Overall singleton removal and clique finding took less than 10 days on the same platform as used for the uniqueing.

2.4   The matrix step

Finding dependencies among the rows of a sparse matrix using block Wiedemann or block Lanczos, currently the two favorite methods, takes time proportional to the product of the dimension and the weight (i.e., number of non-zero entries) of the matrix. Merging is a generic term for the set of strategies developed to build a matrix for which close to optimal performance can be expected for the dependency search of one's choice. It is described in [6]. We ran about 10 merging jobs, aiming for various optimizations (low dimension, low weight, best-of-both, etc.), which each took a couple of days on a single core per node of a 37 node 2.66 GHz Core2 cluster with 16 GB RAM per node, and a not particularly fast interconnection network. The matrix that we ended up using was generated by a 5-day process running on two to three cores on the same 37-node cluster, where the long duration was probably due to the larger communication overhead. It produced a $192\,796\,550 \times 192\,795\,550$-matrix of total weight

27 797 115 920 (thus, on average 144 non-zeros per row). Storage of the matrix required about 105 gigabytes. When we started the project, we expected dimension about a quarter billion and density about 150, which would have been about $\frac{7}{4}$ times harder than what we managed to find (thanks to our large amount of oversieving).

We obtained various other matrices that would all lead to worse performance for the next step, determining dependencies. For instance, for the full collection of relations we found a matrix with about 181 million rows and average weight about 171 per row, which is only slightly worse. Restriction to the set of relations satisfying the much smaller b_ℓ-value of 2^{34} still led to a matrix with more rows than columns (as required). It turned out not to be competitive. Similarly, the matrix we found when using the cutoff $b_\ell = 2^{35}$ was not competitive. More details will be provided at a later stage.

We used the block Wiedemann algorithm [9] with block width $8 \cdot 64$ to find dependencies modulo 2 among the approximately 193 million rows resulting from the filtering. The details of this algorithm can be found in [9,40] and [2, Section 5.1]. Below we give a higher level explanation of the three basic steps of the Wiedemann algorithm (cf. [18, Section 2.19]).

Given a non-singular $d \times d$ matrix M over the finite field $\mathbf{F}_2$ of two elements and $b \in \mathbf{F}_2^d$, we wish to solve the system $Mx = b$. The minimal polynomial F of M on the vector space spanned by b, Mb, M^2b, ... has degree at most d, so that

$$F(M)b = \sum_{i=0}^{d} F_i M^i b = 0.$$

From $F_0 = 1$ it follows that $x = \sum_{i=1}^{d} F_i M^{i-1}b$, so it suffices to find the F_i's.

Denoting by $m_{i,j}$ the jth coordinate of the vector $M^i b$, it follows from the above equation that for each j with $1 \leq j \leq d$ the sequence $(m_{i,j})_{i=0}^{\infty}$ satisfies a linear recurrence relation of order at most d defined by the coefficients F_i: for any $t \geq 0$ and $1 \leq j \leq d$ we have that

$$\sum_{i=0}^{d} F_i m_{i+t,j} = 0.$$

It is well known that $2d + 1$ consecutive terms of an order d linear recurrence suffice to compute the coefficients generating the recurrence, and that this can be done efficiently using the Berlekamp-Massey method (cf. [22,40]). Each particular j may lead to a polynomial of smaller degree than F, but taking, if necessary, the least common multiple of the polynomials found for a small number of different indices j, the correct minimal polynomial will be found.

Summarizing the above, there are three major stages: a first iteration consisting of $2d$ matrix times vector multiplication steps to generate $2d + 1$ consecutive terms of the linear recurrence, the Berlekamp-Massey stage to calculate the F_i's, and finally the second iteration consisting of d matrix times vector multiplication steps to calculate the solution using the F_i's. For large matrices the two iterations, i.e., the first and the final stage, are the most time consuming.

In practice it is common to use blocking, taking advantage of the fact that on 64-bit machines 64 different vectors b over $\mathbf{F}_2$ can be processed simultaneously, at little or no extra cost compared to a single vector (cf. [9]), while using the same three main stages. If the vector $\bar{b}$ is 64 bits wide and in the first stage, the first 64 coordinates of each of the generated 64 bits

wide vectors $M^i\bar{b}$ are kept, the number of matrix (M) times vector $(\bar{b})$ multiplication steps in both the first and the final stage is reduced by a factor of 64 compared to the number of M times b multiplication steps, while making the central Berlekamp-Massey stage a bit more cumbersome. It is less common to take the blocking a step further and run both iteration stages spread over a small number n' of different sequences, possibly run on different clusters at different locations; in [2] this was done with $n' = 4$ sequences run on three clusters at two locations. If for each sequence one keeps the first $64 \cdot n'$ coordinates of each of the 64 bits wide vectors they generate during the first stage, the number of steps to be carried out (per sequence) is further reduced by a factor of n', while allowing independent and simultaneous execution on possibly n' different clusters. After the first stage the data generated for the n' sequences have to be gathered at a central location where the Berlekamp-Massey stage will be carried out.

While keeping the first $64 \cdot n'$ coordinates per step for each sequence results in a reduction of the number of steps per sequence by a factor of $64 \cdot n'$, keeping a different number of coordinates while using n' sequences results in another reduction in the number of steps for the first stage. Following [2, Section 5.1], if the first $64 \cdot m'$ coordinates are kept of the 64 bits wide vectors for n' sequences, the numbers of steps become $\frac{d}{64 \cdot m'} + \frac{d}{64 \cdot n'} = (\frac{n'}{m'} + 1)\frac{d}{64 \cdot n'}$ and $\frac{d}{64 \cdot n'}$ for the first and third stage, respectively and for each of the n' sequences. The choices of m' and n' should be weighed off against the cost of the Berlekamp-Massey step with time and space complexities proportional to $\frac{(m'+n')^3}{n'}d^{1+o(1)}$ and $\frac{(m'+n')^2}{n'}d$, respectively and for $d \to \infty$, and where the exponent "3" may be replaced by the matrix multiplication exponent (our implementation uses "3").

When actually running the first stage of block Wiedemann in this way using n' different sequences, the effect of non-identical resources used for the different sequences quickly becomes apparent: some locations finish their allotted work faster than others (illustrated in Fig. 1). To keep the fast contributors busy and to reduce the work of the slower ones (thereby reducing the wall-clock time), it is possible to let a quickly processed first stage sequence continue for s steps beyond $(\frac{n'}{m'} + 1)\frac{d}{64 \cdot n'}$ while reducing the number of steps in another first stage sequence by the same s. As described in Appendix B, this can be done in a very flexible way, as long as the overall number of steps over all first stage sequences adds up to $n' \cdot (\frac{n'}{m'} + 1)\frac{d}{64 \cdot n'}$. The termination points of the sequences in the third stage need to be adapted accordingly. This is easily arranged for, since the third stage allows much easier and much wider parallelization anyhow (assuming checkpoints from the first stage are kept). Another way to keep all parties busy is swapping jobs, thus requiring data exchanges, synchronization, and more human interaction, making it a less attractive option altogether.

For our matrix with $d \approx 193 \cdot 10^6$ we used, as in [2], $m' = 2n'$. But where $n' = 4$ was used in [2], we used $n' = 8$, thereby quadrupling the Berlekamp-Massey runtime and doubling its memory compared to the matrix from [2], on top of the increased runtime and memory demands caused by the larger dimension of the matrix. On the other hand, the compute intensive first and third stages could be split up into twice as many independent jobs as before. For the first stage on average $(\frac{8}{16} + 1)\frac{193 \cdot 10^6}{64 \cdot 8} \approx 565\,000$ steps needed to be taken per sequence (for 8 sequences), for the second stage the average was about $\frac{193 \cdot 10^6}{64 \cdot 8} \approx 380\,000$ steps. The actual numbers of steps varied, approximately, between $490\,000$ and $650\,000$ for the first stage and between $300\,000$ and $430\,000$ for the third stage. The calculation of these stages was carried out on a wide variety of clusters accessed from three locations: a 56-node cluster of 2.2GHz dual hex-core
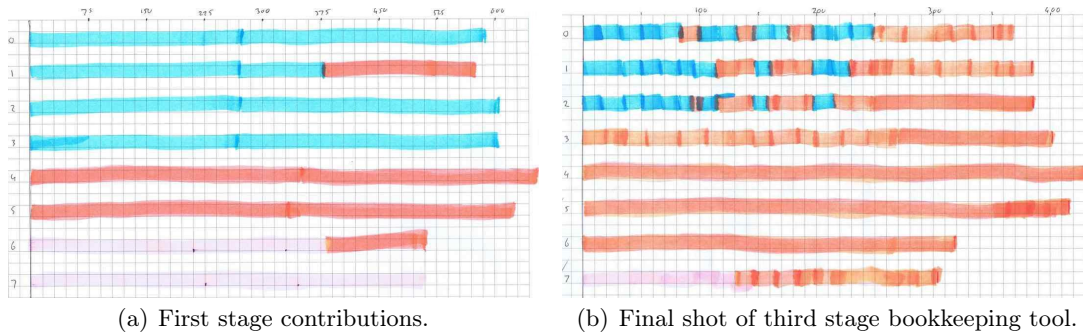
| (a) First stage contributions. | (b) Final shot of third stage bookkeeping tool. |

Fig. 1: Contributions to sequences 0-7 with blue indicating INRIA, orange EPFL, and pink NTT.

AMD processors with Infiniband at EPFL (installed while the first stage was in progress), a variety of ALADDIN-G5K clusters in France accessed from INRIA LORIA, and a cluster of 110 3.0GHz Pentium-D processors on a Gb Ethernet at NTT. A comprehensive overview of clusters and timings is given in Appendix D. It follows from those timings that doing the entire first and third stage would have taken 98 days on 48 nodes (576 cores) of the 56-node EPFL cluster.

The first stage was split up into eight independent jobs run in parallel on those clusters, with each of the eight sequences check-pointing once every 2^{14} steps. Running a first (or third) stage sequence required 180 GB RAM, a single 64 bits wide $\bar{b}$ took 1.5 gigabytes, and a single m_i matrix 8 kilobytes, of which $565\,000$ were kept, on average, per first stage sequence. Each partial sum during the third stage evaluation required 12 gigabytes.

The central Berlekamp-Massey stage was done in 17 hours and 20 minutes on the 56-node EPFL cluster (with 16 GB RAM per node) mentioned above, while using just 4 of the 12 available cores per node. Most of the time the available 896 GB RAM sufficed, but during a central part of the calculation more memory was needed (up to about 1 TB) and some swapping occurred. The third stage started right after completion of the second stage, running as many jobs in parallel as possible. The actual bookkeeping sheet used is pictured in Fig. 1b. Fig. 1a pictures the first stage contributions apocryphally but accurately. Calendar time for the entire block Wiedemann step was 119 days, finishing on December 8, 2009. The percentages of the contributions to the three stages are given more precisely in Table 1.

2.5 That's a bingo[7]

There is nothing new to be reported for the square root step, except for the resulting factorization of RSA-768. Nevertheless, and for the record, we present some of the details.

As expected the matrix step resulted in $512 = 64 \cdot 8$ linear dependencies modulo 2 among the exponent vectors. This was more than enough to include the quadratic characters, which were not included in the matrix, at this stage (cf. [1]). As as result, the solution space was reduced from 512 to 460 elements, giving us 460 independent chances of about $\frac{1}{2}$ to factor RSA-768.

---

[7] "Is that the way you say it?"
  "You just say bingo." (cf. [39])

The square roots of the algebraic numbers were calculated by means of the method from [23] (see also [31]), which uses the known factorization of the algebraic numbers into small prime ideals of known norms. The implementation based on [3] turned out to have a bug when computing the valuations for the free relations of the prime ideals lying above the divisor $17\,722\,398\,737 > 2^{32}$ of the discriminant of f_1. Along with a bug in the quadratic character calculation, this delayed completion of the square root step by a few (harrowing) days. Another approach would be to write down the algebraic number as a product over the $a - b\alpha_1$ in a dependency and to compute the square root using a mathematics software package. Although the latter method is attractively simple to program and has been used for small scale NFS factorizations, it was not considered for the present calculation due to the less appealing size of the algebraic number involved: we estimate that writing it down would have required about 36 gigabytes, and that the calculation would fit in 50 to 100 GB RAM.

Once the bugs were located and fixed, it took two hours using the hard disk and one core on each of twelve dual hex-core 2.2GHz AMD processors to compute the exponents of all prime ideals for eight solutions simultaneously. Computing a square root using the implementation from [3] took one hour and forty minutes on such a dual hex-core processor. The first one (and four of the other seven) led to the factorization, found at 20:16 GMT on December 12, 2009:

$$
\begin{aligned}
\text{RSA-768} = \ & 33478071698956898786044169848212690817704794983713768568 91 \\
& 2413889828883793878002287614711652531743087737814467999489 \cdot \\
& 3674604366679959042824463379962795263227915816434308764267 \\
& 6032283815739666511279233373417143396810270092798736308917.
\end{aligned}
$$

Both factors have 384 bits and 116 decimal digits. The factorizations of the factors ± 1 can be found in Appendix C.

3   Concluding remarks

It is customary to conclude a paper reporting a new factoring record with a preview of coming attractions. For the present case it suffices to remark that our main conclusion has already been reported in [2, Section 7], and was summarized in the first paragraph of the introduction to this paper: at this point factoring a 1024-bit RSA modulus looks more than five times easier than a 768-bit RSA modulus looked back in 1999, when we achieved the first public factorization of a 512-bit RSA modulus. Nevertheless, a 1024-bit RSA modulus is still about one thousand times harder to factor than a 768-bit one. If we are optimistic, it may be possible to factor a 1024-bit RSA modulus within the next decade by means of an academic effort on the same limited scale as the effort presented here. From a practical security point of view this is not a big deal, given that standards recommend phasing out such moduli by the end of the year 2010 (cf. [27,28]). See also [21].

Another conclusion from our work is that we can quite confidently say that if we restrict ourselves to an open community, academic effort as ours and unless something dramatic happens in factoring, we will not be able to factor a 1024-bit RSA modulus within the next five years (cf. [29]). After that, all bets are off.

The ratio between sieving and matrix time was almost 10. This is probably not optimal if one wants to minimize the overall runtime. But, as already alluded to in the introduction,

minimization of runtime may not be the most important criterion. Sieving is easy, and doing more sieving may be a good investment if it leads to a less painful matrix step. We expect that the relations collected for RSA-768 will enable us to get a better insight in the trade-off between sieving and matrix efforts, where also the choice of the bound b_ℓ may play a role. This is a subject for further study that may be expected to lead, ultimately, to a recommendation for close to optimal parameter choices – depending on what one wants to optimize – for NFS factorizations of numbers of 700 to 800 bits.

Our computation required more than 10^{20} operations. With the equivalent of almost 2000 years of computing on a single core 2.2GHz AMD Opteron, on the order of 2^{67} instructions were carried out. The overall effort is sufficiently low that even for short-term protection of data of little value, 768-bit RSA moduli can no longer be recommended. This conclusion is the opposite of the one arrived at on [38], which is based on a hypothetical factoring effort of six months on $100\,000$ workstations, i.e., about two orders of magnitude more than we spent.

Acknowledgements

References

1. L.M. Adleman, Factoring numbers using singular integers, Proceedings 23rd Annual ACM Symposium on Theory of Computing (STOC) (1991) 64–71.
2. K. Aoki, J. Franke, T. Kleinjung, A.K. Lenstra, D.A. Osvik, A kilobit special number field sieve factorization, Proceedings Asiacrypt 2007, Springer-Verlag, LNCS 4833 (2007) 1–12.
3. F. Bahr, Liniensieben und Quadratwurzelberechnung für das Zahlkörpersieb, Diplomarbeit, University of Bonn, 2005.
4. F. Bahr, M. Böhm, J. Franke, T. Kleinjung, Factorization of RSA-200, May 2005, `http://www.loria.fr/~zimmerma/records/rsa200`.
5. J. Buhler, P.L. Montgomery, R. Robson, R. Ruby, Technical report implementing the number field sieve, Oregon State University, Corvallis, OR, 1994.
6. S. Cavallar, Strategies for filtering in the number field sieve, Proceedings ANTS IV, Springer-Verlag, LNCS 1838 (2000) 209–231.
7. 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 (2000) 1–18.
8. D. Coppersmith, Solving linear equations over GF(2): block Lanczos algorithm, Linear algebra and its applications 192 (1993) 33–60.
9. D. Coppersmith, Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm, Math. of Comp. 62 (1994) 333–350.

10. J. Cowie, B. Dodson. R.M. Elkenbracht-Huizing, A.K. Lenstra, P.L. Montgomery, J. Zayer, A world wide number field sieve factoring record: on to 512 bits, Proceedings Asiacrypt'96, Springer-Verlag, LNCS 1163 (1996), 382–394.

11. J.D. Dixon, *Asymptotically fast factorization of integers*, Math. Comp. 36 (1981) 255–260.

12. J. Franke, T. Kleinjung, Continued fractions and lattice sieving; Proceedings SHARCS 2005; `http://www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/talks/FrankeKleinjung.pdf`.

13. R. Golliver, A.K. Lenstra, K. McCurley, Lattice sieving and trial division, Proceedings ANTS'94, Springer-Verlag, LNCS 877 (1994) 18–27.

14. T. Kleinjung, Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers, `http://www.hyperelliptic.org/tanja/SHARCS/talks06/thorsten.pdf`, 2005.

15. T. Kleinjung, On polynomial selection for the general number field sieve. Math. Comp. 75 (2006) 2037–2047.

16. T. Kleinjung, Polynomial selection, talk presented at the CADO workshop, `http://cado.gforge.inria.fr/workshop/slides/kleinjung.pdf`, 2008.

17. A.K. Lenstra, Computational methods in public key cryptology, pp. 175–238 in: H. Niederreiter, Coding theory and cryptology, Singapore University Press, 2002.

18. A.K. Lenstra, H.W. Lenstra, Jr., *Algorithms in number theory*, chapter 12 in *Handbook of theoretical computer science, Volume A, algorithms and complexity* (J. van Leeuwen, ed.), Elsevier, Amsterdam (1990).

19. A.K. Lenstra, H.W. Lenstra, Jr. (editors), The development of the number field sieve, Springer-Verlag, LNM 1554, August 1993.

20. A.K. Lenstra, H.W. Lenstra, Jr., M.S. Manasse, J. Pollard, The factorization of the ninth Fermat number, Math. Comp. 61 (1994) 319–349.

21. A.K. Lenstra, E. Tromer, A. Shamir, W. Kortsmit, B. Dodson, J. Hughes, P. Leyland, Factoring estimates for a 1024-bit RSA modulus, Proceedings Asiacrypt 2003, Springer-Verlag, LNCS 2894 (2003) 55–74.

22. J. Massey, Shift-register synthesis and BCH decoding, IEEE Trans. Inf. Th., 15 (1969) 122–127.

23. P.L. Montgomery, Square roots of products of algebraic numbers, `http://ftp.cwi.nl/pub/pmontgom/sqrt.ps.gz`.

24. P.L. Montgomery, B. Murphy, *Improved polynomial selection for the number field sieve*, extended abstract for the conference on the mathematics of public-key cryptography, June 13-17, 1999, the Fields institute, Toronto, Ontario, Canada.

25. M.A. Morrison, J. Brillhart, *A method of factorization and the factorization of F_7*, Math. Comp. 29 (1975) 183–205.

26. B.A. Murphy, Modelling the Yield of Number Field Sieve Polynomials, Proceedings ANTS III, Springer-Verlag, LNCS 1423 (1998) 137–147.

27. National Institute of Standards and Technology, Suite B Cryptography: `http://csrc.nist.gov/groups/SMA/ispab/documents/minutes/2006-03/E_Barker-March2006-ISPAB.pdf`.

28. National Institute of Standards and Technology, Special Publication 800-57: Recommendation for Key Management Part 1: General (Revised),
`http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf`.

29. National Institute of Standards and Technology, Discussion paper: the transitioning of cryptographic algorithms and key sizes,
`http://csrc.nist.gov/groups/ST/key_mgmt/documents/Transitioning_CryptoAlgos_070209.pdf`.

30. NFS@home, `http://escatter11.fullerton.edu/nfs`.

31. P. Nguyen, A Montgomery-like square root for the number field sieve, Proceedings ANTS III, Springer-Verlag, LNCS 1423 (1998) 151–168.

32. J.M. Pollard, The lattice sieve, 43–49 in [19].

33. C. Pomerance, *Analysis and comparison of some integer factoring algorithms*, in *Computational methods in number theory* (H.W. Lenstra, Jr., R. Tijdeman, eds.), Math. Centre Tracts **154**, **155**, Mathematisch Centrum, Amsterdam (1983) 89-139.

34. C. Pomerance, *The quadratic sieve factoring algorithm*, Proceedings Eurocrypt'84, Springer-Verlag, LNCS 209 (1985) 169-182.

35. C. Pomerance, A tale of two sieves, `http://www.ams.org/notices/199612/pomerance.pdf`.

36. R. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public key cryptosystems, Commun. of the ACM, 21 (1978) 120–126.

37. The RSA challenge numbers, formerly on `http://www.rsa.com/rsalabs/node.asp?id=2093`, now on for instance `http://en.wikipedia.org/wiki/RSA_numbers`.

38. The RSA factoring challenge FAQ, `http://www.rsa.com/rsalabs/node.asp?id=2094`.

39. Q. Tarantino, `http://www.youtube.com/watch?v=EzgVY7lxSlA`.
40. E. Thomé, Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm, Journal of symbolic computation 33 (2002) 757–775.

A   Sieving by vectors

The aim of this section is to give a short description of the implementation of lattice sieving described in [12] and used for most of the number field sieve factorization records of the previous decade.

Let $v_k(a,b) = b^{d_k} f_k(a/b)$. Recall that the idea of the method of lattice sieving, which was introduced by Pollard [32], is to increase the probability of smoothness of $v_1(a,b)$ by only looking at (a,b)-pairs for which $v_1(a,b)$ is divisible by some large prime q, called the special q. Let $s \bmod q$ be a residue class such that this is the case for $a \equiv sb \bmod q$. One constructs a reduced base (u,v) of the lattice of all $(a,b) \in \mathbf{Z}^2$ with $a \equiv sb \bmod q$. A scalar product adapted to the skewness of the polynomial pair is used for this reduction. The problem is then to find all pairs (i,j), $-I/2 \le i < I/2$, $0 < j < J$, without common divisor such that $v_1(a,b)/q$ and $v_2(a,b)$ are both smooth, with $(a,b) = iu + jv$. For the sake of simplicity we assume I to be even. As mentioned in Section 2.3, for practical values of the parameters, I tends to be much smaller than the smoothness bounds b_1 and b_2, and it is non-trivial to efficiently sieve such regions.

Pollard proposed to do this by using, for each (prime,root) pair $\mathfrak{p}$ with prime p bounded by the relevant smoothness bound b_k, a reduced base of the lattice $\Gamma_{\mathfrak{p}}$ of pairs (i,j) for which $v_k(a,b)$ for the corresponding (a,b)-pair is divisible by p. In [13] that approach was used for p larger than a small multiple of I, while avoiding storage of "even, even" sieve locations (and using line sieving for the other primes). Our approach is similar, but instead uses a truncated continued fraction expansion to determine a basis $B = \big((\alpha, \beta), (\gamma, \delta)\big)$ of $\Gamma_{\mathfrak{p}}$ with the following properties:

a The numbers β and δ are positive.
b We have $-I < \alpha \le 0 \le \gamma < I$ and $\gamma - \alpha \ge I$.

Let us assume that $\Gamma_{\mathfrak{p}}$ consists of all (i,j) for which $i \equiv \rho j \bmod p$, where $0 < \rho < p$. The case $\rho = 0$ is not treated because it only gives (i,j) which fail to be coprime, as is the case when $\Gamma_{\mathfrak{p}}$ consists of all (i,j) for which p divides j. We also assume $p \ge I$, as smaller primes are better treated by line sieving. To construct a basis with the above properties, one takes $(i_0, j_0) = (-p, 0)$, $(i_1, j_1) = (\rho, 1)$ and puts $(i_{\ell+1}, j_{\ell+1}) = (i_{\ell-1}, j_{\ell-1}) + r(i_\ell, j_\ell)$ with $r = \lfloor -\frac{i_{\ell-1}}{i_\ell} \rfloor$. Note that $(-1)^{\ell+1} i_\ell \ge 0$, that r is positive and that the j_ℓ thus form an increasing sequence of non-negative numbers. The process is stopped at the first ℓ with $\mid i_\ell \mid < I$. If this number ℓ is odd, we put $(\alpha, \beta) = (i_{\ell-1}, j_{\ell-1}) + r(i_\ell, j_\ell)$, where r is the smallest integer for which $\alpha > -I$. If ℓ is even, we put $(\gamma, \delta) = (i_{\ell-1}, j_{\ell-1}) + r(i_\ell, j_\ell)$, where r is the smallest integer such that $\gamma < I$. In both cases, the element of $B = \big((\alpha, \beta), (\gamma, \delta)\big)$ not yet described is given by (i_ℓ, j_ℓ).

To explain how to efficiently sieve using a basis with these properties, let $(i,j) \in \Gamma_{\mathfrak{p}}$ such that $-I/2 \le i < I/2$. We want to find the (uniquely determined) $(i', j') \in \Gamma_{\mathfrak{p}}$ such that

$-I/2 \leq i' < I/2$, $j' > j$, and j' is as small as possible. As B is a basis of $\Gamma_{\mathfrak{p}}$, there are integers d and e with

$$(i', j') - (i, j) = d(\alpha, \beta) + e(\gamma, \delta).$$

If the numbers d and e were both different from zero, with opposite signs, then condition **b** on B would force the first component of the right hand side to have absolute value $\geq I$, whereas our constraints on i and i' force it to have absolute value $< I$. Since $j' - j$, β, and δ are all positive, we have $d \geq 0$ and $e \geq 0$. It is now easy to see that the solution to our problem is:

$$(d, e) = \begin{cases} (0, 1) & \text{if } i < I/2 - \gamma \\ (1, 1) & \text{if } I/2 - \gamma \leq i < -I/2 - \alpha \\ (1, 0) & \text{if } i \geq I/2 - \alpha. \end{cases}$$

To convince oneself of the minimality of j', one notes that $d = 0$ leads to a violation of $i' < I/2$ unless $i < I/2 - \gamma$ (i.e., save for the first of the above cases) and that $e = 0$ leads to $i' < -I/2$ unless $i \geq I/2 - \alpha$ (i.e., save for the third of the above cases).

To implement this process on a modern CPU, it seems best to take $I = 2^{\iota}$ for some natural number ι. It is possible to identify pairs (i, j) of integers with $-I/2 \leq i < I/2$ with integers x by putting $x = j \cdot I + i + I/2$. If $x' = j' \cdot I + i' + I/2$ with (i', j') as above, then $x' = x + C$, $x' = x + A + C$ and $x' = x + A$ in the three cases above, with $A = \alpha + I \cdot \beta$ and $C = \gamma + I \cdot \delta$. The first component of a pair (i, j), (α, β) or (γ, δ) is extracted from these numbers by using a bitwise logical operation, and the selection of the appropriate one of the above three cases is best done using conditional move instructions.

For cache efficiency, the sieving region $S_{\mathfrak{q}}$ was split into areas A_t, $0 \leq t < T$, of size equal to the L1-cache size. For primes p larger than that size (or a small multiple thereof), sieving is not done directly. Instead, the numbers x corresponding to elements of $S_{\mathfrak{q}} \cap \Gamma_{\mathfrak{p}}$ were calculated ahead of the sieving process, and their offsets into the appropriate region A_t stored in the corresponding element of an array $\mathcal{S}$ of T stacks. To implement the trial division sieve efficiently, the corresponding factor base index was also stored. Of course, this approach may also be used for line sieving, and in fact was used in [3]. An approach which seems to be somewhat similar has been described by T. Oliveira e Silva in connection with his implementation of the Odlyzko-Lagarias-Lehmer-Meissel method.

Parallelization is possible in several different ways. A topology for splitting the sieving region among several nodes connected by a network is described in [12]. If one wants to split the task among several cores sharing their main memory, it seems best to distribute the regions A_t and also the large factor base primes among them. Each core first calculates its part of $\mathcal{S}$, for its assigned part of the large factor base elements, and then uses the information generated by all cores to treat its share of regions A_t. A lattice siever parallelized that way was used for a small part of the RSA-576 sieving tasks, but the code fell out of use and was not used for the current project. The approach may be more useful today, with many cores per processor being a standard.

B   Unbalanced sequences in block Wiedemann

Before describing the modification for unbalanced sequence lengths we give an overview of Coppersmith's block version of the Berlekamp-Massey algorithm. To avoid a too technical description we simplify the presentation of Coppersmith's algorithm and refer to [9] for details. The modification can also be applied to Thomé's subquadratic algorithm (cf. [40]) which is what we did and used. In the following the variables m and n do not denote the common root and a number to be factored, but have the same meaning as in Coppersmith's article. As in that article, the terms $+O(1)$ are constants depending on m and n. We assume that m and n, which play the role of $64 \cdot m'$ and $64 \cdot n'$ in Section 2.4, are much smaller than d.

Let M be a $d \times d$ matrix over $\mathbf{F}_2$, $m \geq n$, $x_k \in \mathbf{F}_2^d$, $1 \leq k \leq m$ and $y_j \in \mathbf{F}_2^d$, $1 \leq j \leq n$ satisfying certain conditions. Set $a_{j,k}^{(i)} = x_k^T M^i y_j$ and

$$A = \sum_i (a_{j,k}^{(i)}) X^i \quad \in \mathrm{Mat}_{n,m}[X].$$

In the first step of block Wiedemann we calculate the coefficients of A up to degree $\frac{d}{m} + \frac{d}{n} + O(1)$.

The goal of the Berlekamp-Massey algorithm is to find polynomials of matrices $F \in \mathrm{Mat}_{n,n}[X]$ and $G \in \mathrm{Mat}_{n,m}[X]$ satisfying $\deg(F) < \frac{d}{n} + O(1)$, $\deg(G) < \frac{d}{n} + O(1)$ and

$$FA \equiv G \pmod{X^{\frac{d}{m} + \frac{d}{n} + O(1)}}.$$

Intuitively, we want to produce at least d zero rows in the higher coefficients of FA up to degree $\frac{d}{m} + \frac{d}{n} + O(1)$. Writing $F = \sum_{i=0}^{d_F} f_{j,k}^{(i)} X^i$, $d_F = \deg(F)$ the j-th row of coefficient $d_F + b$ of G being zero corresponds to

$$(M^b x_h)^T v_j = 0 \quad \text{for } 1 \leq h \leq m, 0 \leq b < \frac{d}{m} + O(1) \text{ where}$$

$$v_j = \sum_{k=1}^{n} \sum_{i=0}^{d_F} f_{j,k}^{(d_F - i)} \cdot M^i y_k.$$

Coppersmith's algorithm produces a sequence of matrices (of $m + n$ rows) $F_t \in \mathrm{Mat}_{m+n,n}[X]$ and $G_t \in \mathrm{Mat}_{m+n,m}[X]$ for $t = t_0, \ldots, \frac{d}{m} + \frac{d}{n} + O(1)$ (where $t_0 = O(1)$) such that

$$F_t A \equiv G_t \pmod{X^t}$$

holds and the degrees of F_t and G_t are roughly $\frac{m}{m+n} t$. In a first step t_0 and F_{t_0} are chosen such that certain conditions are satisfied, in particular we have $\deg(F_{t_0}) = O(1)$ and $\deg(G_{t_0}) = O(1)$. To go from t to $t+1$ a polynomial of degree 1 of matrices $P_t \in \mathrm{Mat}_{m+n,m+n}[X]$ is constructed and we set $F_{t+1} = P_t F_t$ and $G_{t+1} = P_t G_t$. This construction is done as follows. We have $F_t A \equiv G_t + E_t X^t \pmod{X^{t+1}}$ for some matrix E_t. Respecting a restriction involving the degrees of the rows of G_t (essentially we avoid destroying previously constructed zero rows in the G's) we perform a Gaussian elimination on E_t, i.e., we obtain $\tilde{P}_t$ such that

$$\tilde{P}_t E_t = \begin{pmatrix} 0 \\ \mathbf{1}_m \end{pmatrix}.$$

Then we set

$$P_t = \begin{pmatrix} \mathbf{1}_n & 0 \\ 0 & \mathbf{1}_m X \end{pmatrix} \cdot \tilde{P}_t.$$

In this way the degrees of at most m rows are increased when passing from G_t to G_{t+1} (due to the restriction mentioned above $\tilde{P}_t$ does not increase the degrees), so the total number of zero rows in the coefficients is increased by n. Due to the restriction mentioned above the degrees of the rows of F_t and of G_t grow almost uniformly, i.e., they grow on average by $\frac{m}{m+n}$ when going from t to $t+1$.

After $t = \frac{d}{m} + \frac{d}{n} + O(1)$ steps the total number of zero rows in the coefficients of G_t is $\frac{m+n}{m}d + O(1)$ such that we can select m rows that produce at least d zero rows in the coefficients. These m rows form F and G.

We now consider unbalanced sequence lengths. Let ℓ_j be the length of sequence j, i.e., $a_{j,k}^{(i)}$ has been computed for all k and $0 \leq i \leq \ell_j$. Without loss of generality we can assume $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n = \ell$. The sum of the lengths of all sequences has to satisfy again $\sum_j \ell_j \geq d \cdot (1 + \frac{n}{m}) + O(1)$. Moreover we can assume that $\ell_1 \geq \frac{d}{m}$, otherwise we could drop sequence 1 completely, thus facilitating our task.

In this setting our goal is to achieve

$$FA \equiv G \pmod{X^{\ell + O(1)}}$$

with $d_F = \deg(F) < \ell - \frac{d}{m}$, $\deg(G) < \ell - \frac{d}{m}$ and

$$X^{\ell - \ell_k} \mid F_{\cdot,k} \qquad \text{(this denotes the k-th column of F)}.$$

The latter condition will compensate our ignorance of some rows of the higher coefficients of A. Indeed, setting for simplicity $d_F = \ell - \frac{d}{m}$, the vectors

$$v_j = \sum_{k=1}^{n} \sum_{i=0}^{\ell_k - \frac{d}{m}} f_{j,k}^{(d_F - i)} \cdot M^i y_k$$

satisfy for $1 \leq h \leq m, 0 \leq b < \frac{d}{m}$

$$(M^b x_h)^T v_j = \sum_{k=1}^{n} \sum_{i=0}^{\ell_k - \frac{d}{m}} f_{j,k}^{(d_F - i)} a_{k,h}^{(i+b)} = g_{j,h}^{(d_F + b)} = 0.$$

If $i + b > \ell_k$ (thus $a_{k,h}^{(i+b)}$ not being computed), we have $d_F - i < d_F + b - \ell_k \leq \ell - \ell_k$, so $f_{j,k}^{(d_F - i)} = 0$ and the sum computes $g_{j,h}^{(d_F + b)}$.

Our new goal is achieved as before, but we will need ℓ steps and the construction of P_t has to be modified as follows. In step t we have $F_t A \equiv G_t + E_t X^t \pmod{X^{t+1}}$. Let $a \leq n$ be maximal such that

$$\sum_{i=1}^{a-1} (m+i)(\ell_{n-i+1} - \ell_{n-i}) \leq mt$$

(a will increase during the computation). In the Gaussian elimination of E_t we do not use the first $n - a$ rows for elimination. As a consequence, $\tilde{P}_t$ has the form

$$\tilde{P}_t = \begin{pmatrix} \mathbf{1}_{n-a} & * \\ 0 & * \end{pmatrix}.$$

Then we set

$$P_t = \begin{pmatrix} \mathbf{1}_{n-a}X & 0 & 0 \\ 0 & \mathbf{1}_a & 0 \\ 0 & 0 & \mathbf{1}_mX \end{pmatrix} \cdot \tilde{P}_t.$$

Therefore the sum of the degrees of F_t will be increased by $m + n - a$ and the number of zero rows in G_t will be increased by a when passing from t to $t+1$. For a fixed a, $\frac{(m+a)(\ell_{n-a+1} - \ell_{n-a})}{m}$ steps will increase the average degree of the last $m + a$ rows from $\ell - \ell_{n-a+1}$ to $\ell - \ell_{n-a}$. At this point a will be increased.

To see why $X^{\ell - \ell_k} \mid F_{.,k}$ holds we have to describe the choice of F_{t_0} (and t_0). Let c be the number of maximal ℓ_j, i.e., $\ell_{n-c} < \ell_{n-c+1} = \ell_n$. Then F_{t_0} will be of the form

$$F_{t_0} = \begin{pmatrix} \mathbf{1}_{n-c}X^{t_0} & 0 \\ 0 & * \end{pmatrix}.$$

The last $m + c$ rows will be chosen such that they are of degree at most $t_0 - 1$ and such that the conditions in Coppersmith's algorithm are satisfied. This construction will lead to a value of t_0 near $\frac{m}{c}$ instead of the lower value near $\frac{m}{n}$ in the original algorithm.

Let k be such that $\ell_k < \ell$ and consider the k-th column of F_t. As long as $n - a \geq k$ this column will have the only non-zero entry at row k and this will be X^t. Since $n - a \geq k$ holds for $t \leq \ell - \ell_k$ this column will be divisible by $X^{\ell - \ell_k}$ for all $t \geq \ell - \ell_k$.

For RSA-768 we used the algorithm as described above in the subquadratic version of Thomé. However, a variant of the algorithm might be useful in certain situations (e.g., if one of the sequences is much longer than the others) which we will sketch briefly.

Suppose that $\ell_{n-1} < \ell_n$. Then for $t < \frac{(m+1)(\ell_n - \ell_{n-1})}{m}$ we have $a = 1$ and P_t is of the form

$$P_t = \begin{pmatrix} \mathbf{1}_{n-1}X & * \\ 0 & * \end{pmatrix}.$$

A product of several of these P_t will have a similar form, namely an $(n-1) \times (n-1)$ unit matrix times a power of X in the upper left corner and zeros below it.

The basic operations in Thomé's subquadratic version are building a binary product tree of these P_t and doing truncated multiplications of intermediate products with $F_{t_0}A$ or similar polynomials. If we split the computation in this algorithm into two stages, first computing the product of all P_t for $t < \frac{(m+1)(\ell_n - \ell_{n-1})}{m}$ and then the remaining product, the matrix multiplications in the first stage become easier due to the special form of the P_t and its products.

Obviously this can be done in as many stages as there are different values among the ℓ_j.

C Factorizations of $p \pm 1$ and $q \pm 1$

Referring to the smallest factor of RSA-768 as p and to its cofactor as q, the prime factorizations of $p \pm 1$ and $q \pm 1$ are as follows, where "pk" denotes a prime number of k decimal digits:

$$p - 1 = 2^8 \cdot 11^2 \cdot 13 \cdot 7193 \cdot 160378082551 \cdot 7721565388263419219 \cdot$$
$$11110316344948488248471 1393053 \cdot p47,$$
$$p + 1 = 2 \cdot 3 \cdot 5 \cdot 3193212274955337226200549186163 0345183416467 \cdot p71,$$
$$q - 1 = 2^2 \cdot 359 \cdot p113,$$
$$q + 1 = 2 \cdot 3 \cdot 23 \cdot 41 \cdot 47 \cdot 239875144072757917901 \cdot p90.$$

D Clusters used and block Wiedemann timings.

Table 2: Data and first and third stage block Wiedemann timings for all clusters used.

Cluster location	number of nodes	CPU type	clock speed (GHz)	cores per node	GB RAM per node	interconnect	nodes per job	cores per job	seconds per iteration stage 1	stage 3	communication
Lausanne	56	2×AMD 2427	2.2	12	16	ib20g	12	144	4.3-4.5	4.8	40%
Tokyo	110	2×Pentium 4	3.0	2	5	eth1g	110	220	5.8†, 6.4	7.8	..%†, ..%
Grenoble	34	2×Xeon E5420	2.5	8	8	ib20g	24	144	3.7	n/a	30%
Lille	46	2×Xeon E5440	2.8	8	8	mx10g	36	144	3.1	3.3	31%
							32	256	3.8	n/a	38%
							24	144	4.4	n/a	33%
Nancy	92	2×Xeon L5420	2.5	8	16	ib20g	64	256	2.2	2.4	41%
							36	144	3.0	3.2	31%
							24	144	3.5	4.2	30%
							18	144	n/a	5.0	31%
							16	64	n/a	6.5	19%
Orsay	120	2×AMD 250	2.4	2	2	mx10g	98	196	2.8	3.9	32%
Rennes	96	2×Xeon 5148	2.3	4	4	mx10g	64	256	2.5	2.7	37%
							49	196	2.9	3.5	33%
Rennes	64	2×Xeon L5420	2.5	8	32	eth1g	49	196	6.2	n/a	67%
							24	144	8.4	n/a	67%
							18	144	10.0	n/a	68%
							8	64	n/a	18.0	56%

"n/a" means that the job configuration was not used.

†: figure per iteration per sequence when two sequences are processed in parallel, for all other figures just a single sequence is processed.