Analysis of Bernstein's Factorization Circuit

Arjen K. Lenstra[1], Adi Shamir[2], Jim Tomlinson[3], and Eran Tromer[2]

[1] Citibank, N.A. and Technische Universiteit Eindhoven
1 North Gate Road, Mendham, NJ 07945-3104, U.S.A.
arjen.lenstra@citigroup.com

[2] Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot 76100, Israel
{shamir,tromer}@wisdom.weizmann.ac.il

[3] 99 E. 28th St., Bayonne, NJ 07002-4912, U.S.A.
jimtom@optonline.net

Abstract. In [1], Bernstein proposed a circuit-based implementation of the matrix step of the number field sieve factorization algorithm. These circuits offer an asymptotic cost reduction under the measure "construction cost $\times$ run time". We evaluate the cost of these circuits, in agreement with [1], but argue that compared to previously known methods these circuits can factor integers that are 1.17 times larger, rather than 3.01 as claimed (and even this, only under the non-standard cost measure). We also propose an improved circuit design based on a new mesh routing algorithm, and show that for factorization of 1024-bit integers the matrix step can, under an optimistic assumption about the matrix size, be completed within a day by a device that costs a few thousand dollars. We conclude that from a practical standpoint, the security of RSA relies exclusively on the hardness of the relation collection step of the number field sieve.

Keywords: factorization, number field sieve, RSA, mesh routing

1 Introduction

In [1], a new circuit-based approach is proposed for one of the steps of the number field sieve (NFS) integer factorization method, namely finding a linear relation in a large but sparse matrix. Unfortunately, the proposal from [1] has been misinterpreted on a large scale, even to the extent that announcements have been made that the results imply that common RSA key sizes no longer provide an adequate level of security.

In this paper we attempt to give a more balanced interpretation of [1]. In particular, we show that 1024-bit RSA keys are as secure as many believed them to be. Actually, [1] provides compelling new evidence that supports a traditional and widely used method to evaluate the security of RSA moduli. We present a variant of the analysis of [1] that would suggest that, under the metric proposed in [1], the number of digits of factorable integers n has grown by a factor $1.17 + o(1)$, for $n \to \infty$ (in [1] a factor of $3.01 + o(1)$ is mentioned).

We propose an improved circuit design, based on mesh routing rather than mesh sorting. To this end we describe a new routing algorithm whose performance in our setting seems optimal. With some further optimizations, the construction cost is reduced by several orders of magnitude compared to [1]. In the improved design the parallelization is gained essentially for free, since its cost is comparable to the cost of RAM needed just to store the input matrix.

We estimate the cost of breaking 1024-bit RSA with current technology. Using custom-built hardware to implement the improved circuit, the NFS matrix step becomes surprisingly inexpensive. However, the theoretical analysis shows that the cost of the relation collection step cannot be significantly reduced, regardless of the cost of the matrix step. We thus conclude that the practical security of RSA for commonly used modulus sizes is not significantly affected by [1].

Section 2 reviews background on the NFS; it does not contain any new material and simply serves as an explanation and confirmation of the analysis from [1]. Section 3 sketches the circuit approach of [1] and considers its implications. Section 4 discusses various cost-aspects of the NFS. Section 5 focuses on 1024-bit numbers, presenting custom hardware for the NFS matrix step both following [1] and using the newly proposed circuit. Section 6 summarizes our conclusions. Appendices A and B outline the limitations of off-the-shelf parts for the mesh-based approach and the traditional approach, respectively. Throughout this paper, n denotes the composite integer to be factored. Prices are in US dollars.

2 Background on the number field sieve

In theory and in practice the two main steps of the NFS are the *relation collection step* and the *matrix step*. We review their heuristic asymptotic runtime analysis because it enables us to stress several points that are important for a proper understanding of "standard-NFS" and of "circuit-NFS" as proposed in [1].

2.1 Smoothness. An integer is called B-smooth if all its prime factors are at most B. Following [10, 3.16] we denote by $L_x[r;\alpha]$ any function of x that equals

$$e^{(\alpha+o(1))(\log x)^r(\log\log x)^{1-r}},\ \text{for } x\to\infty,$$

where α and r are real numbers with $0\le r\le 1$ and logarithms are natural. Thus, $L_x[r;\alpha]+L_x[r;\beta]=L_x[r;\max(\alpha,\beta)]$, $L_x[r;\alpha]L_x[r;\beta]=L_x[r;\alpha+\beta]$, $L_x[r;\alpha]L_x[s;\beta]=L_x[r;\alpha]$ if $r<s$, $L_x[r,\alpha]^k=L_x[r,k\alpha]$ and if $\alpha>0$ then $(\log x)^kL_x[r;\alpha]=L_x[r;\alpha]$ for any fixed k, and $\pi(L_x[r;\alpha])=L_x[r;\alpha]$ where $\pi(y)$ is the number of primes $\le y$.

Let $\alpha>0$, $\beta>0$, r, and s be fixed real numbers with $0<s<r\le 1$. A random positive integer $\le L_x[r;\alpha]$ is $L_x[s;\beta]$-smooth with probability

$$L_x[r-s;-\alpha(r-s)/\beta],\ \text{for } x\to\infty.$$

We abbreviate L_n to L and $L[1/3,\alpha]$ to $L(\alpha)$. Thus, a random integer $\le L[2/3,\alpha]$ is $L(\beta)$-smooth with probability $L(-\alpha/(3\beta))$. The notation $L^{1.901\cdots+o(1)}$ in [1] corresponds to $L(1.901\cdots)$ here. We write "$\zeta\overset{o}{=}x$" for "$\zeta=x+o(1)$ for $n\to\infty$."

2.2 Ordinary NFS. To factor n using the NFS, more or less following the approach from [11], one selects a positive integer

$$d = \delta\left(\frac{\log n}{\log\log n}\right)^{1/3}$$

for a positive value δ that is yet to be determined, an integer m close to $n^{1/(d+1)}$, a polynomial $f(X) = \sum_{i=0}^{d} f_i X^i \in \mathbf{Z}[X]$ such that $f(m) \equiv 0 \bmod n$ with each f_i of the same order of magnitude as m, a rational smoothness bound B_r, and an algebraic smoothness bound B_a. Other properties of these parameters are not relevant for our purposes.

A pair (a,b) of integers is called a *relation* if a and b are coprime, $b > 0$, $a - bm$ is B_r-smooth, and $b^d f(a/b)$ is B_a-smooth. Each relation corresponds to a sparse D-dimensional bit vector with

$$D \approx \pi(B_r) + \#\{(p,r) : p \text{ prime} \leq B_a,\ f(r) \equiv 0 \bmod p\} \approx \pi(B_r) + \pi(B_a)$$

(cf. [11]). In the relation collection step a set of more than D relations is sought. Given this set, one or more linear dependencies modulo 2 among the corresponding D-dimensional bit vectors are constructed in the matrix step. Per dependency there is a chance of at least 50% (exactly 50% for RSA moduli) that a factor of n is found in the final step, the square root step. We discuss some issues of the relation collection and matrix steps that are relevant for [1].

2.3 Relation collection. We restrict the search for relations to the rectangle $|a| < L(\alpha)$, $0 < b < L(\alpha)$ and use B_r and B_a that are both $L(\beta)$ (which does not imply that $B_r = B_a$), for $\alpha, \beta > 0$ that are yet to be determined. It follows (cf. 2.1) that $D = L(\beta)$. Furthermore,

$$|a - bm| = L[2/3, 1/\delta] \quad \text{and} \quad |b^d f(a/b)| = L[2/3, \alpha\delta + 1/\delta].$$

With 2.1 and under the usual assumption that $a - bm$ and $b^d f(a/b)$ behave, with respect to smoothness probabilities, independently as random integers of comparable sizes, the probability that both are $L(\beta)$-smooth is

$$L\left(\frac{-1/\delta}{3\beta}\right) \cdot L\left(\frac{-\alpha\delta - 1/\delta}{3\beta}\right) = L\left(-\frac{\alpha\delta + 2/\delta}{3\beta}\right).$$

The search space contains $2L(\alpha)^2 = 2L(2\alpha) = L(2\alpha)$ pairs (a,b) and, due to the $o(1)$, as many pairs (a,b) with $\gcd(a,b) = 1$. It follows that α and β must be chosen such that

$$L(2\alpha) \cdot L\left(-\frac{\alpha\delta + 2/\delta}{3\beta}\right) = L(\beta)\ (= D).$$

We find that

$$\alpha \overset{o}{=} \frac{3\beta^2 + 2/\delta}{6\beta - \delta}. \tag{1}$$

2.4 Testing for smoothness. The (a, b) search space can be processed in $L(2\alpha)$ operations. If sufficient memory is available this can be done using sieving. Current PC implementations intended for the factorization of relatively small numbers usually have adequate memory for sieving. For much larger numbers and current programs, sieving would become problematic. In that case, the search space can be processed in the "same" $L(2\alpha)$ operations (with an, admittedly, larger $o(1)$) but at a cost of only $L(0)$ memory using the Elliptic Curve Method (ECM) embellished in any way one sees fit with trial division, Pollard rho, early aborts, etc., and run on any number K of processors in parallel to achieve a K-fold speedup. This was observed many times (see for instance [10, 4.15] and [4]). Thus, despite the fact that current implementations of the relation collection require substantial memory, it is well known that asymptotically this step requires negligible memory without incurring, in theory, a runtime penalty – in practice, however, it is substantially slower than sieving. Intermediate solutions that exchange sieving memory for many tightly coupled processors with small memories could prove valuable too; see [6] for an early example of this approach and [1] for various other interesting proposals that may turn out to be practically relevant. For the asymptotic argument, ECM suffices.

In improved NFS from [4] it was necessary to use a "memory-free" method when searching for B_a-smooth numbers (cf. 2.2), in order to achieve the speedup. It was suggested in [4] that the ECM may be used for this purpose. Since memory usage was no concern for the analysis in [4], regular "memory-wasteful" sieving was suggested to test B_r-smoothness.

2.5 The matrix step. The choices made in 2.3 result in a bit matrix A consisting of $D = L(\beta)$ columns such that each column of A contains only $L(0)$ nonzero entries. Denoting by $w(A)$ the total number of nonzero entries of A (its *weight*), it follows that $w(A) = L(\beta) \cdot L(0) = L(\beta)$. Using a variety of techniques [5, 13], dependencies can be found after, essentially, $O(D)$ multiplications of A times a bit vector. Since one matrix-by-vector multiplication can be done in $O(w(A)) = L(\beta)$ operations, the matrix step can be completed in $L(\beta)^2 = L(2\beta)$ operations. We use "standard-NFS" to refer to NFS that uses a matrix step with $L(2\beta)$ operation count.

We will be concerned with a specific method for finding the dependencies in A, namely the block Wiedemann algorithm [5][18] whose outline is as follows. Let K be the blocking factor, i.e., the amount of parallelism desired. We may assume that either $K = 1$ or $K > 32$. Choose $2K$ binary D-dimensional vectors $\vec{v}_i, \vec{u}_j$ for $1 \le i, j \le K$. For each i, compute the vectors $A^k \vec{v}_i$ for k up to roughly $2D/K$, using repeated matrix-by-vector multiplication. For each such vector $A^k \vec{v}_i$, compute the inner products $\vec{u}_j A^k \vec{v}_i$, for all j. Only these inner products are saved, to conserve storage. From the inner products, compute certain polynomials $f_l(x)$, $l = 1, \ldots, K$ of degree about D/K. Then evaluate $f_l(A)\vec{v}_i$, for all l and i (take one $\vec{v}_i$ at a time and evaluate $f_l(A)\vec{v}_i$ for all l simultaneously using repeated matrix-by-vector multiplications). From the result, K elements from the kernel of A can be computed. The procedure is probabilistic, but succeeds with high probability for $K \gg 1$ [17]. For $K = 1$, the cost roughly doubles [18].

For reasonable blocking factors ($K = 1$ or $32 \le K \ll \sqrt{D}$), the block Wiedemann algorithm involves about $3D$ matrix-by-vector multiplications. These multiplications dominate the cost of the matrix step; accordingly, the circuits of [1], and our variants thereof, aim to reduce their cost. Note that the multiplications are performed in $2K$ separate chains where each chain involves repeated left-multiplication by A. The proposed circuits rely on this for their efficiency. Thus, they appear less suitable for other dependency-finding algorithms, such as block Lanczos [13] which requires just $2D$ multiplications.

2.6 NFS parameter optimization for matrix exponent $2\epsilon > 1$. With the relation collection and matrix steps in $L(2\alpha)$ and $L(2\beta)$ operations, respectively, the values for α, β, and δ that minimize the overall NFS operation count follow using Relation (1). However, we also need the optimal values if the "cost" of the matrix step is different from $L(\beta)^2$: in [1] "cost" is defined using a metric that is not always the same as operation count, so we need to analyse the NFS using alternative cost metrics. This can be done by allowing flexibility in the "cost" of the matrix step: we consider how to optimize the NFS parameters for an $L(\beta)^{2\epsilon}$ matrix step, for some exponent $\epsilon > 1/2$. The corresponding relation collection operation count is fixed at $L(2\alpha)$ (cf. 2.4).

We balance the cost of the relation collection and matrix steps by taking $\alpha \stackrel{o}{=} \epsilon\beta$. With (1) it follows that

$$3(2\epsilon-1)\beta^2 - \epsilon\beta\delta - 2/\delta \stackrel{o}{=} 0, \text{ so that } \beta \stackrel{o}{=} \frac{\epsilon\delta + \sqrt{\epsilon^2\delta^2 + 24(2\epsilon-1)/\delta}}{6(2\epsilon-1)}.$$

Minimizing β given ϵ leads to

$$\delta \stackrel{o}{=} \sqrt[3]{3(2\epsilon-1)/\epsilon^2} \tag{2}$$

and

$$\beta \stackrel{o}{=} 2\sqrt[3]{\epsilon/(3(2\epsilon-1))^2}. \tag{3}$$

Minimizing the resulting

$$\alpha \stackrel{o}{=} 2\epsilon\sqrt[3]{\epsilon/(3(2\epsilon-1))^2} \tag{4}$$

leads to $\epsilon = 1$ and $\alpha \stackrel{o}{=} 2/3^{2/3}$: even though $\epsilon < 1$ would allow more "relaxed" relations (i.e., larger smoothness bounds and thus easier to find), the fact that more of such relations have to be found becomes counterproductive. It follows that an operation count of $L(4/3^{2/3})$ is optimal for relation collection, but that for $2\epsilon > 2$ it is better to use suboptimal relation collection because otherwise the matrix step would dominate. We find the following optimal NFS parameters:

$1 < 2\epsilon \le 2$**:**
$\delta \stackrel{o}{=} 3^{1/3}$, $\alpha \stackrel{o}{=} 2/3^{2/3}$, and $\beta \stackrel{o}{=} 2/3^{2/3}$, with operation counts of relation collection and matrix steps equal to $L(4/3^{2/3})$ and $L(4\epsilon/3^{2/3})$, respectively. For $\epsilon = 1$ the operation counts of the two steps are the same (when expressed in L) and the overall operation count is $L(4/3^{2/3}) = L((64/9)^{1/3}) =$

$L(1.9229994\cdots)$. This corresponds to the heuristic asymptotic runtime of the NFS as given in [11]. We refer to these parameter choices as the *ordinary parameter choices*.

$2\epsilon > 2$**:**
δ, α, and β as given by Relations (2), (4), and (3), respectively, with operation count $L(2\alpha)$ for relation collection and cost $L(2\epsilon\beta)$ for the matrix step, where $L(2\alpha) = L(2\epsilon\beta)$. More in particular, we find the following values.

$2\epsilon = 5/2$**:**
$\delta \stackrel{o}{=} (5/3)^{1/3}(6/5)$, $\alpha \stackrel{o}{=} (5/3)^{1/3}(5/6)$, and $\beta \stackrel{o}{=} (5/3)^{1/3}(2/3)$, for an operation count and cost $L((5/3)^{4/3}) = L(1.9760518\cdots)$ for the relation collection and matrix steps, respectively. These values are familiar from [1, Section 6: Circuits]. With $(1.9229994\cdots/1.9760518\cdots + o(1))^3 \stackrel{o}{=} 0.9216$ and equating operation count and cost, this suggests that factoring $0.9216 \cdot 512 \approx 472$-bit composites using NFS with matrix exponent 5/2 is comparable to factoring 512-bit ones using standard-NFS with ordinary parameter choices (disregarding the effects of the $o(1)$'s).

$2\epsilon = 3$**:**
$\delta \stackrel{o}{=} 2/3^{1/3}$, $\alpha \stackrel{o}{=} 3^{2/3}/2$, and $\beta \stackrel{o}{=} 3^{-1/3}$, for an operation count and cost of $L(3^{2/3}) = L(2.0800838\cdots)$ for the relation collection and matrix steps, respectively.

2.7 Improved NFS. It was shown in [4] that ordinary NFS from [11], and as used in 2.2, can be improved by using more than a single polynomial f. Let α and δ be as in 2.3 and 2.2, respectively, let β indicate the rational smoothness bound B_r (i.e., $B_r = L(\beta)$), and let γ indicate the algebraic smoothness bound B_a (i.e., $B_a = L(\gamma)$). Let G be a set of $B_r/B_a = L(\beta - \gamma)$ different polynomials, each of degree d and common root m modulo n (as in 2.2). A pair (a, b) of integers is a relation if a and b are coprime, $b > 0$, $a - bm$ is B_r-smooth, and $b^d g(a/b)$ is B_a-smooth for at least one $g \in G$. Let ϵ be the matrix exponent. Balancing the cost of the relation collection and matrix steps it follows that $\alpha \stackrel{o}{=} \epsilon\beta$.

Optimization leads to

$$\gamma \stackrel{o}{=} \left(\frac{\epsilon^2 + 5\epsilon + 2 + (\epsilon+1)\sqrt{\epsilon^2 + 8\epsilon + 4}}{9(2\epsilon+1)}\right)^{1/3}$$

and for this γ to

$$\alpha \stackrel{o}{=} \frac{9\gamma^3 + 1 + \sqrt{18\gamma^3(2\epsilon+1) + 1}}{18\gamma^2}, \qquad \beta \stackrel{o}{=} \alpha/\epsilon,$$

and

$$\delta \stackrel{o}{=} \frac{3\gamma(-4\epsilon - 1 + \sqrt{18\gamma^3(2\epsilon+1) + 1})}{9\gamma^3 - 4\epsilon}.$$

It follows that for $2\epsilon = 2$ the method from [4] gives an improvement over the ordinary method, namely $L(1.9018836\cdots)$. The condition $\beta \geq \gamma$ leads to $2\epsilon \leq 7/3$, so that for $2\epsilon > 7/3$ (as in circuit-NFS, cf. 3.1) usage of the method from [4] no longer leads to an improvement over the ordinary method. This explains why in [1] the method from [4] is used to select parameters for standard-NFS and why the ordinary method is used for circuit-NFS.

With 2.1 it follows that the sum of the (rational) sieving and ECM-based (algebraic) smoothness times from [4] (cf. last paragraph of 2.4) is minimized if $\beta = \gamma + 1/(3\beta\delta)$. The above formulas then lead to $2\epsilon = (3+\sqrt{17})/4 = 1.7807764\cdots$. Therefore, unlike the ordinary parameter selection method, optimal relation collection for the improved method from [4] occurs for an ϵ with $2\epsilon < 2$: with $\epsilon = 0.8903882\cdots$ the operation count for relation collection becomes $L(1.8689328\cdots)$. Thus, in principle, and depending on the cost function one is using, the improved method would be able to take advantage of a matrix step with exponent $2\epsilon < 2$. If we disregard the matrix step and minimize the operation count of relation collection, this method yields a cost of $L(1.8689328\cdots)$.

3 The circuits for integer factorization from [1]

3.1 Matrix-by-vector multiplication using mesh sorting. In [1] an interesting new mesh-sorting-based method is described to compute a matrix-by-vector product. Let A be the bit matrix from 2.5 with $D = L(\beta)$ columns and weight $w(A) = L(\beta)$, and let m be the least power of 2 such that $m^2 > w(A)+2D$. Thus $m = L(\beta/2)$. We assume, without loss of generality, that A is square. A mesh of $m \times m$ processors, each with $O(\log D) = L(0)$ memory, initially stores the matrix A and a not necessarily sparse D-dimensional bit vector $\vec{v}$. An elegant method is given that computes the product $A\vec{v}$ using repeated sorting in $O(m)$ steps, where each step involves a small constant number of simultaneous operations on all $m \times m$ mesh processors. At the end of the computation $A\vec{v}$ can easily be extracted from the mesh. Furthermore, the mesh is immediately, without further changes to its state, ready for the computation of the product of A and the vector $A\vec{v}$. We use "circuit-NFS" to refer to NFS that uses the mesh-sorting-based matrix step.

3.2 The throughput cost function from [1]. Judging by operation counts, the mesh-based algorithm is not competitive with the traditional way of computing $A\vec{v}$: as indicated in 2.5 it can be done in $O(w(A)) = L(\beta)$ operations. The mesh-based computation takes $O(m)$ steps on all $m \times m$ mesh processors simultaneously, resulting in an operation count per matrix-by-vector multiplication of $O(m^3) = L(3\beta/2)$. Iterating the matrix-by-vector multiplications $L(\beta)$ times results in a mesh-sorting-based matrix step that requires $L(5\beta/2) = L(\beta)^{5/2}$ operations as opposed to just $L(2\beta)$ for the traditional approach. This explains the non-ordinary relation collection parameters used in [1] corresponding to the analysis given in 2.6 for $2\epsilon = 5/2$, something we comment upon below in 3.3.

However, the standard comparison of operation counts overlooks the following fact. The traditional approach requires memory $O(w(A) + D) = L(\beta)$ for storage of A and the vector; given that amount of memory it takes time $L(2\beta)$. But given the $m \times m$ mesh, with $m \times m = L(\beta)$, the mesh-based approach takes time just $L(3\beta/2)$ because during each unit of time $L(\beta)$ operations are carried out simultaneously on the mesh. To capture the advantage of "active small processors" (as in the mesh) compared to "inactive memory" (as in the traditional approach) and the fact that their price is comparable, it is stipulated in [1] that the cost of factorization is "the product of the time and the cost of the machine." We refer to this cost function as **throughput cost**, since it can be interpreted as measuring the equipment cost per unit problem-solving throughput. It is frequently used in VLSI design (where it's known as "AT cost", for Area$\times$Time), but apparently was not used previously in the context of computational number theory.

It appears that throughput cost is indeed appropriate when a large number of problems must be solved during some long period of time while minimizing total expenses. This does not imply that throughput cost is always appropriate for assessing security, as illustrated by the following example. Suppose Carrol wishes to assess the risk of her encryption key being broken by each of two adversaries, Alice and Bob. Carrol knows that Alice has plans for a device that costs $1M and takes 50 years to break a key, and that Bob's device costs $50M and takes 1 year to break a key. In one scenario, each adversary has a $1M budget — clearly Alice is dangerous and Bob is not. In another scenario, each adversary has a $50M budget. This time both are dangerous, but Bob apparently forms a greater menace because he can break Carrol's key within one year, while Alice still needs 50 years. Thus, the two devices have the same throughput cost, yet either can be more "dangerous" than the other, depending on external settings. The key point is that if Alice and Bob have many keys to break within 50 years then indeed their cost-per-key figures are identical, but the time it will take Bob to break Carrol's key depends on her priority in his list of victims, and arguably Carrol should make the paranoid assumption that she is first.

In Section 4 we comment further on performance measurement for the NFS.

3.3 Application of the throughput cost. The time required for all matrix-by-vector multiplications on the mesh is $L(3\beta/2)$. The equipment cost of the mesh is the cost of m^2 small processors with $L(0)$ memory per processor, and is thus $L(\beta)$. The throughput cost, the product of the time and the cost of the equipment, is therefore $L(5\beta/2)$. The matrix step of standard-NFS requires time $L(2\beta)$ and equipment cost $L(\beta)$ for the memory, resulting in a throughput cost of $L(3\beta)$. Thus, the throughput cost advantage of the mesh-based approach is a factor $L(\beta/2)$ if the two methods would use the same β (cf. Remark 3.4).

The same observation applies if the standard-NFS matrix step is K-fold parallelized, for reasonable K (cf. 2.5): the time drops by a factor K which is cancelled (in the throughput cost) by a K times higher equipment cost because each participating processor needs the same memory $L(\beta)$. In circuit-NFS (i.e., the mesh) a parallelization factor m^2 is used: the time drops by a factor only m (not m^2), but the equipment cost stays the same because memory $L(0)$ suffices for each of the m^2 participating processors. Thus, with the throughput cost circuit-NFS achieves an advantage of $m = L(\beta/2)$. The mesh itself can of course be K-fold parallelized but the resulting K-fold increase in equipment cost and K-fold drop in time cancel each other in the throughput cost [1, Section 4].

Remark 3.4 It can be argued that before evaluating an existing algorithm based on a new cost function, the algorithm first should be tuned to the new cost function. This is further commented upon below in 3.5.

3.5 Implication of the throughput cost. We consider the implication of the matrix step throughput cost of $L(5\beta/2)$ for circuit-NFS compared to $L(3\beta)$ for standard-NFS. In [1] the well known fact is used that the throughput cost of relation collection is $L(2\alpha)$ (cf. 2.4): an operation count of $L(2\alpha)$ on a single processor with $L(0)$ memory results in time $L(2\alpha)$, equipment cost $L(0)$, and throughput cost $L(2\alpha)$. This can be time-sliced in any way that is convenient, i.e., for any K use K processors of $L(0)$ memory each and spend time $L(2\alpha)/K$ on all K processors simultaneously, resulting in the same throughput cost $L(2\alpha)$. Thus, for relation collection the throughput cost is proportional to the operation count. The analysis of 2.6 applies with $2\epsilon = 5/2$ and leads to an optimal overall circuit-NFS throughput cost of $L(1.9760518\cdots)$. As mentioned above and in 3.2, the throughput cost and the operation count are equivalent for both relation collection and the matrix step of circuit-NFS. Thus, as calculated in 2.6, circuit-NFS is from an operation count point of view less powerful than standard-NFS, losing already 40 bits in the 500-bit range (disregarding the $o(1)$'s) when compared to standard-NFS with ordinary parameter choices. This conclusion applies to any NFS implementation, such as many existing ones, where memory requirements are not multiplicatively included in the cost function.

But operation count is not the point of view taken in [1]. There standard-NFS is compared to circuit-NFS in the following way. The parameters for standard-NFS are chosen under the assumption that the throughput cost of relation collection is $L(3\alpha)$: operation count $L(2\alpha)$ and memory cost $L(\alpha)$ for the sieving result in time $L(2\alpha)/K$ and equipment cost $K \cdot L(\alpha)$ (for any K-fold parallelization) and thus throughput cost $L(3\alpha)$. This disregards the fact that long before [1] appeared is was known that the use of $L(\alpha)$ memory per processor may be convenient, in practice and for relatively small numbers, but is by no means required (cf. 2.4). In any case, combined with $L(3\beta)$ for the throughput cost of the matrix step this leads to $\alpha \triangleq \beta$, implying that the analysis from 2.6 with $2\epsilon = 2$ applies, but that the resulting operation count must be raised to the 3/2-th power. In [1] the improvement from [4] mentioned in 2.7 is used, leading to a throughput cost for standard-NFS of $L(2.8528254\cdots)$ (where $2.8528254\cdots$ is 1.5 times the $1.9018836\cdots$ referred to in 2.7). Since $(2.8528254\cdots/1.9760518\cdots)^3 = 3.0090581\cdots$, it is suggested in [1] that the number of digits of factorable composites grows by a factor 3 if circuit-NFS is used instead of standard-NFS.

3.6 Alternative interpretation. How does the comparison between circuit-NFS and standard-NFS with respect to their throughput costs turn out if standard-NFS is first properly tuned (Remark 3.4) to the throughput cost function, given the state of the art in, say, 1990 (cf. [10, 4.15]; also the year that [4] originally appeared)? With throughput cost $L(2\alpha)$ for relation collection (cf. above and 2.4), the analysis from 2.6 with $2\epsilon = 3$ applies, resulting in a throughput cost of just $L(2.0800838\cdots)$ for standard-NFS. Since $(2.0800838\cdots/1.9760518\cdots)^3 < 1.17$, this would suggest that $1.17D$-digit composites can be factored using circuit-NFS for the throughput cost of D-digit integers using standard-NFS. The significance of this comparison depends on whether or not the throughput cost is an acceptable way of measuring the cost of standard-NFS. If not, then the conclusion based on the operation count (as mentioned above) would be that circuit-NFS is slower than standard-NFS; but see Section 4 for a more complete picture. Other examples where it is recognized that the memory cost of relation collection is asymptotically not a concern can be found in [12] and [9], and are implied by [14].

Remark 3.7 It can be argued that the approach in 3.6 of replacing the ordinary standard-NFS parameters by smaller smoothness bounds in order to make the matrix step easier corresponds to what happens in many actual NFS factorizations. There it is done not only to make the matrix step less cumbersome at the cost of somewhat more sieving, but also to make do with available PC memories. Each contributing PC uses the largest smoothness bounds and sieving range that fit conveniently and that cause minimal interference with the PC-owner's real work. Thus, parameters may vary from machine to machine. This is combined with other memory saving methods such as "special-q's." In any case, if insufficient memory is available for sieving with optimal ordinary parameters, one does not run out to buy more memory but settles for slight suboptimality, with the added benefit of an easier matrix step. See also 4.1.

Remark 3.8 In [19], Wiener outlines a three-dimensional circuit for the matrix step, with structure that is optimal in a certain sense (when considering the cost of internal wiring). This design leads to a matrix step exponent of $2\epsilon = 7/3$, compared to $5/2$ in the designs of [1] and this paper. However, adaptation of that design to two dimensions yields a matrix step exponent that is asymptotically identical to ours, and vice versa. Thus the approach of [19] is asymptotically equivalent to ours, while its practical cost remains to be evaluated. We note that in either approach, there are sound technological reasons to prefer the 2D variant. Interestingly, $2\epsilon = 7/3$ is the point where improved and standard NFS become the same (cf. 2.7).

4 Operation count, equipment cost, and real time

The asymptotic characteristics of standard-NFS and circuit-NFS with respect to their operation count, equipment, and real time spent are summarized in Table 1. For non-$L(0)$ equipment requirements it is specified if the main cost goes to memory ("RAM"), processing elements ("PEs") with $L(0)$ memory, or a square mesh as in 3.1, and "tuned" refers to the alternative analysis in 3.6.

The underlined operation counts are the same as the corresponding throughput costs. For the other operation count the throughput cost (not optimized if no sieving is used) follows by taking the maximum of the products of the figures in the "equipment" and "real time" columns. Relation collection, whether using sieving or not, allows almost arbitrary parallelization (as used in the last two rows of Table 1). The amount of parallelization allowed in the matrix step of standard-NFS is much more limited (cf. 2.5); it is not used in Table 1.

Table 1. NFS costs: operation count, equipment, and real time.

		relation collection		matrix step	
	overall operation count	equipment	real time	equipment	real time
standard-NFS:					
sieving	$L(1.90)$	$L(0.95)$ RAM	$L(1.90)$	$L(0.95)$ RAM	$L(1.90)$
no sieving		$L(0)$			
tuned no sieving	$\underline{L(2.08)}$	sequential: $L(0)$ parallel: $L(0.69)$ PEs	$L(2.08)$ $L(1.39)$	$L(0.69)$ RAM	$L(1.39)$
circuit-NFS:	$\underline{L(1.98)}$	sequential: $L(0)$ parallel: $L(0.79)$ PEs	$L(1.98)$ $L(1.19)$	$L(0.79)$ mesh	$L(1.19)$

4.1 Lowering the cost of the standard-NFS matrix step. We show at what cost the asymptotic advantages of the circuit-NFS matrix step (low throughput cost and short real time) can be matched, asymptotically, using the traditional approach to the matrix step. This requires a smaller matrix, i.e., lower smoothness bounds, and results therefore in slower relation collection. We illustrate this with two examples. To get matching throughput costs for the matrix steps of standard-NFS and circuit-NFS, β must be chosen such that $L(3\beta) = L((5/3)^{4/3}) = L(1.9760\cdots)$, so that the matrix step of standard-NFS requires $L(\beta) = L(0.6586\cdots)$ RAM and real time $L(2\beta) = L(1.3173\cdots)$. Substituting this β in Relation (1) and minimizing α with respect to δ we find

$$\delta \hat{=} \frac{\sqrt{4+36\beta^3}-2}{3\beta^2}, \tag{5}$$

i.e., $\delta \hat{=} 1.3675\cdots$ and $\alpha \hat{=} 1.0694\cdots$, resulting in relation collection operation count $L(2.1389\cdots)$. Or, one could match the real time of the matrix steps: with $L(2\beta) = L((5/3)^{1/3}) = L(1.1856\cdots)$ the matrix step of standard-NFS requires $L(0.5928\cdots)$ RAM and real time $L(1.1856\cdots)$. With Relation (5) we find that $\delta \hat{=} 1.3195\cdots$, $\alpha \hat{=} 1.1486\cdots$, and relation collection operation count $L(2.2973\cdots)$.

4.2 Operation count based estimates. Operation count is the traditional way of measuring the cost of the NFS. It corresponds to the standard complexity measure of "runtime" and neglects the cost of memory or other equipment that is needed to actually "run" the algorithm. It was used, for instance, in [11] and [4] and was analysed in 2.6 and 2.7.

It can be seen in Table 1, and was indicated in 3.5, that the operation count for circuit-NFS is higher than for standard-NFS (assuming both methods are optimized with respect to the operation count): $L(1.9760518\cdots)$ as opposed to just $L(1.9018836\cdots)$ when using the improved version (cf. 2.7) as in Table 1, or as opposed to $L(1.9229994\cdots)$ when using the ordinary version (cf. 2.6) as in 3.5. Thus, RSA moduli that are deemed sufficiently secure based on standard-NFS operation count security estimates, are even more secure when circuit-NFS is considered instead. Such estimates are common; see for instance [14] and the "computationally equivalent" estimates in [9, 12]. Security estimates based on the recommendations from [14] or the main ones (i.e., the conservative "computationally equivalent" ones) from [9, 12] are therefore not affected by the result from [1]. Nevertheless, we agree with [2] that the PC-based realization suggested in [12], meant to present an at the time possibly realistic approach that users can relate to, may not be the best way to realize a certain operation count; see also the last paragraph of [12, 2.4.7]. The estimates from [15] are affected by [1].

Remark 4.3 Historically, in past factorization experiments the matrix step was always solved using a fraction of the effort required by relation collection. Moreover, the memory requirements of sieving-based relation collection have never turned out to be a serious problem (it was not even necessary to fall back to the memory-efficient ECM and its variations). Thus, despite the asymptotic analysis, extrapolation from past experience would predict that the bottleneck of the NFS method is relation collection, and that simple operation count is a better practical cost measure for NFS than other measures that are presumably more realistic. The choice of cost function in [9, 12] was done accordingly.

The findings of [1] further support this conservative approach, by going a long way towards closing the gap between the two measures of cost when applied to the NFS: 93% of the gap according to 3.5, and 61% according to 3.6.

5 Hardware for the matrix step for 1024-bit moduli

In this section we extrapolate current factoring knowledge to come up with reasonable estimates for the sizes of the matrix A that would have to be processed for the factorization of a 1024-bit composite when using ordinary relation collection (cf. 2.6), and using slower relation collection according to matrix exponent 5/2 as used in circuit-NFS. For the latter (smaller sized) matrix we consider how expensive it would be to build the mesh-sorting-based matrix-by-vector multiplication circuit proposed in [1] using custom-built hardware and we estimate how much time the matrix step would take on the resulting device. We then propose an alternative mesh-based matrix-by-vector multiplication circuit and estimate its performance for both matrices, for custom-built and off-the-shelf hardware.

Throughout this section we are interested mainly in assessing feasibility, for the purpose of evaluating the security implications. Our assumptions will be somewhat optimistic, but we believe that the designs are fundamentally sound and give realistic indications of feasibility using technology that is available in the present or in the near future.

5.1 Matrix sizes. For the factorization of RSA-512 the matrix had about 6.7 million columns and average column density about 63 [3]. There is no doubt that this matrix is considerably smaller than a matrix that would have resulted from ordinary relation collection as defined in 2.6, cf. Remark 3.7. Nevertheless, we make the optimistic assumption that this is the size that would result from ordinary relation collection.

Combining this figure with the $L(2/3^{2/3})$ matrix size growth rate (cf. 2.6) we find

$$6\,700\,000 \cdot \frac{L_{2^{1024}}[1/3, 2/3^{2/3}]}{L_{2^{512}}[1/3, 2/3^{2/3}]} \approx 1.8 \cdot 10^{10}$$

(cf. 2.1). Including the effect of the $o(1)$ it is estimated that an optimal 1024-bit matrix would contain about 10^{10} columns. We optimistically assume an average column density of about 100. We refer to this matrix as the "large" matrix.

Correcting this matrix size for the $L((5/3)^{1/3}(2/3))$ matrix size growth rate for matrix exponent $5/2$ (cf. 2.6) we find

$$1.8 \cdot 10^{10} \cdot \frac{L_{2^{1024}}[1/3, (5/3)^{1/3}(2/3)]}{L_{2^{1024}}[1/3, 2/3^{2/3}]} \approx 8.7 \cdot 10^{7}.$$

We arrive at an estimate of about $4 \cdot 10^7$ columns for the circuit-NFS 1024-bit matrix. We again, optimistically, assume that the average column density is about 100. We refer to this matrix as the "small" matrix.

5.2 Estimated relation collection cost. Relation collection for RSA-512 could have been done in about 8 years on a 1GHz PC [3]. Since

$$8 \cdot \frac{L_{2^{1024}}[1/3, 4/3^{2/3}]}{L_{2^{512}}[1/3, 4/3^{2/3}]} \approx 6 \cdot 10^{7}$$

we estimate that generating the large matrix would require about a year on about 30 million 1GHz PCs with large memories (or more PC-time but less memory when using alternative smoothness tests – keep in mind, though, that it may be possible to achieve the same operation count using different hardware, as rightly noted in [1] and speculated in [12, 2.4.7]). With

$$\frac{L_{2^{1024}}[1/3, (5/3)^{4/3}]}{L_{2^{1024}}[1/3, 4/3^{2/3}]} \approx 5$$

it follows that generating the smaller matrix would require about 5 times the above effort. Neither computation is infeasible. But, it can be argued that 1024-bit RSA moduli provide a reasonable level of security just based on the operation count of the relation collection step.

5.3 Processing the "small" matrix using Bernstein's circuits. We estimate the size of the circuit required to implement the mesh circuit of [1] when the NFS parameters are optimized for the throughput cost function and 1024-bit composites. We then derive a rough prediction of the associated costs when the mesh is implemented by custom hardware using current VLSI technology. In this subsection we use the circuit exactly as described in [1]; the next subsections will make several improvements, including those listed as future plans in [1].

In [1], the algorithm used for finding dependencies among the columns of A is Wiedemann's original algorithm [18], which is a special case of block Wiedemann with blocking factor K=1 (cf. 2.5). In the first stage (inner product computation), we are given the sparse $D \times D$ matrix A and some pair of vectors $\vec{u}, \vec{v}$ and wish to calculate $\vec{u}A^k\vec{v}$ for $k = 1, \ldots, 2D$. The polynomial evaluation stage is slightly different, but the designs given below can be easily adapted so we will not discuss it explicitly.

The mesh consists of $m \times m$ nodes, where $m^2 > w(A) + 2D$ (cf. 3.1). By assumption, $w(A) \approx 4 \cdot 10^9$ and $D \approx 4 \cdot 10^7$ so we may choose $m = 63256$. To execute the sorting-based algorithm, each node consists mainly of 3 registers of $\lceil \log_2(4 \cdot 10^7) \rceil = 26$ bits each, a 26-bit compare-exchange element (in at least half of the nodes), and some logic for tracking the current stage of the algorithm. Input, namely the nonzero elements of A and the initial vector $\vec{v}$, is loaded just once so this can be done serially. The mesh computes the vectors $A^k\vec{v}$ by repeated matrix-by-vector multiplication, and following each such multiplication it calculates the inner product $\vec{u}(A^k\vec{v})$ and outputs this single bit.

In standard CMOS VLSI design, a single-bit register (i.e., a D-type edge-triggered flip-flop) requires about 8 transistors, which amounts to 624 transistors per node. To account for the logic and additional overheads such as a clock distribution network, we shall assume an average of 2000 transistors per node for a total of $8.0 \cdot 10^{12}$ transistors in the mesh.

As a representative of current technology available on large scale we consider Intel's latest Pentium processor, the Pentium 4 "Northwood" (0.13μm^2 feature size process). A single Northwood chip (inclusive of its on-board L2 cache) contains $5.5 \cdot 10^7$ transistors, and can be manufactured in dies of size 131mm^2 on wafers of diameter 300mm, i.e., about 530 chips per wafer when disregarding defects. The 1.6GHz variant is currently sold at \$140 in retail channels. By transistor count, the complete mesh would require about $\left(8.0 \cdot 10^{12}\right) / \left(5.5 \cdot 10^7\right) \approx$ 145 500 Northwood-sized dies or about 273 wafers. Using the above per-chip price figure naively, the construction cost is about \$20M. Alternatively, assuming a wafer cost of about \$5,000 we get a construction cost of roughly \$1.4M, and the initial costs (e.g., mask creation) are under \$1M.

The matter of inter-chip communication is problematic. The mesh as a whole needs very few external lines (serial input, 1-bit output, clock, and power). However, a chip consisting of $s \times s$ nodes has $4s - 4$ nodes on its edges, and each of these needs two 26-bit bidirectional links with its neighbor on an adjacent chip, for a total of about $2 \cdot 2 \cdot 26 \cdot 4s = 416s$ connections. Moreover, such connections typically do not support the full 1GHz clock rate, so to achieve the necessary bandwidth we will need about 4 times as many connections: $1664s$. While standard wiring technology cannot provide such enormous density, the following scheme seems plausible. Emerging "flip-chip" technologies allow direct connections between chips that are placed face-to-face, at a density of 277 connections per mm^2 (i.e., 60μs array pitch). We cut each wafer into the shape of a cross, and arrange the wafers in a two-dimensional grid with the arms of adjacent wafers in full overlap. The central square of each cross-shaped wafer contains mesh nodes, and the arms are dedicated to inter-wafer connections. Simple calculation shows that with the above connection density, if 40% of the (uncut) wafer area is used for mesh nodes then there is sufficient room left for the connection pads and associated circuitry. This disregards the issues of delays (mesh edges that cross wafer boundaries are realized by longer wires and are thus slower than the rest), and of the defects which are bound to occur. To address these, adaptation of the algorithm is needed. Assuming the algorithmic issues are surmountable, the inter-wafer communication entails a cost increase by a factor of about 3, to \$4.1M.

According to [1, Section 4], a matrix-by-vector multiplication consists of, essentially, three sort operations on the $m \times m$ mesh. Each sort operation takes $8m$ steps, where each step consists of a compare-exchange operation between 26-bit registers of adjacent nodes. Thus, multiplication requires $3 \cdot 8m \approx 1.52 \cdot 10^6$ steps. Assuming that each step takes a single clock cycle at a 1GHz clock rate, we get a throughput of 659 multiplications per second.

Basically, Wiedemann's algorithm requires $3D$ multiplications. Alas, the use of blocking factor $K = 1$ entails some additional costs. First, the number of multiplications roughly doubles due to the possibility of failure (cf. 2.5). Moreover, the algorithm will yield a single vector from the kernel of A, whereas the Number Field Sieve requires several linearly independent kernel elements: half of these yield a trivial congruence (c.f. 2.2), and moreover certain NFS optimizations necessitate discarding most of the vectors. In RSA-512, a total of about 10 kernel vectors were needed. Fortunately, getting additional vectors is likely to be cheaper than getting the first one (this is implicit in [18, Algorithm 1]). Overall, we expect the number of multiplications to be roughly $2 \cdot \frac{10}{3} \cdot 3D = 20D$. Thus, the expected total running time is roughly $20 \cdot 4 \cdot 10^7/659 \approx 1\,210\,000$ seconds, or 14 days. The throughput cost is thus $5.10 \cdot 10^{12}$ \$ $\times$ sec.

If we increase the blocking factor from 1 to over 32 and handle the multiplication chains sequentially on a single mesh, then only $3D$ multiplications are needed ([1] considers this but claims that it will not change the cost of computation; that is true only up to constant factors). In this case the time decreases to 50 hours, and the throughput cost decreases to $7.4 \cdot 10^{11}$ \$ $\times$ sec.

Heat dissipation (i.e., power consumption) may limit the node density and clock rate of the device, and needs to be analysed. Note however that this limitation is technological rather than theoretical, since in principle the mesh sorting algorithm can be efficiently implemented using reversible gates and arbitrarily low heat dissipation.

5.4 A routing-based circuit. The above analysis refers to the mesh circuit described in [1], which relies on the novel use of mesh sorting for matrix-by-vector multiplication. We now present an alternative design, based on mesh routing. This design performs a single routing operation per multiplication, compared to three sorting operations (where even a single sorting operation is slower than routing). The resulting design has a reduced cost, improved fault tolerance and very simple local control. Moreover, its inherent flexibility allows further improvements, as discussed in the next section. The basic design is as follows.

For simplicity assume that each of the D columns of the matrix has weight exactly h (here $h = 100$), and that the nonzero elements of A are uniformly distributed (both assumptions can be easily relaxed). Let $m = \sqrt{D \cdot h}$. We divide the $m \times m$ mesh into D blocks of size $\sqrt{h} \times \sqrt{h}$. Let S_i denote the i-th block in row-major order ($i \in \{1, \ldots, D\}$), and let t_i denote the node in the upper left corner of S_i. We say that t_i is the *target of the value* i. Each node holds two $\log_2 D$-bit values, $Q[i]$ and $R[i]$. Each target node t_i also contains a single-bit value $P[i]$. For repeated multiplication of A and $\vec{v}$, the mesh is initialized as follows: the i-th entry of $\vec{v}$ is loaded into $P[i]$, and the row indices of the nonzero elements in column $i \in \{1, \ldots, D\}$ of A are stored (in arbitrary order) in the $Q[\cdot]$ of the nodes in S_i. Each multiplication is performed thus:

1. For all i, broadcast the value of $P[i]$ from t_i to the rest of the nodes in S_i (this can be accomplished in $2\sqrt{h} - 2$ steps).
2. For all i and every node j in S_i: if $P[i] = 1$ then $R[j] \leftarrow Q[j]$, else $R[j] \leftarrow$ nil (where nil is some distinguished value outside $\{1, \ldots, D\}$).
3. $P[i] \leftarrow 0$ for all i
4. Invoke a mesh-based packet routing algorithm on the $R[\cdot]$, such that each non-nil value $R[j]$ is routed to its target node $t_{R[j]}$. Each time a value i arrives at its target t_i, discard it and flip $P[i]$.

After these steps, $P[\cdot]$ contain the result of the multiplication, and the mesh is ready for the next multiplication. As before, in the inner product computation stage of the Wiedemann algorithm, we need only compute $\vec{u}A^k\vec{v}$ for some vector $\vec{u}$, so we load the i-th coordinate of $\vec{u}$ into node t_i during initialization, and compute the single-bit result $\vec{u}A^k\vec{v}$ inside the mesh during the next multiplication.

There remains the choice of a routing algorithm. Many candidates exist (see [7] for a survey). To minimize hardware cost, we restrict our attention to algorithms for the "one packet" model, in which at each step every node holds at most one packet (and consequentially each node can send at most one packet and receive at most one packet per step). Note that this rules out most known algorithms, including those for the well-studied "hot-potato" routing model which provides a register for every edge. Since we do binary multiplication, the routing problem has the following unusual property: pairwise packet annihilation is allowed. That is, pairs of packets with identical values may be "cancelled out" without affecting the result of the computation. This relaxation can greatly reduce the congestion caused by multiple packets converging to a common destination. Indeed this seems to render commonly-cited lower bounds inapplicable, and we are not aware of any discussion of this variant in the literature. While known routing and sorting algorithms can be adapted to our task, we suggest a new routing algorithm that seems optimal, based on our empirical tests.

The algorithm, which we call *clockwise transposition routing*, has an exceptionally simple control structure which consists of repeating 4 steps. Each step involves compare-exchange operations on pairs of neighboring nodes, such that the exchange is performed iff it reduces the distance-to-target of the non-nil value (out of at most 2) that is farthest from its target along the relevant direction. This boils down to comparison of the target row indices (for vertically adjacent nodes) or target column indices (for horizontally adjacent nodes). For instance, for horizontally adjacent nodes $i, i+1$ such that $t_{R[i]}$ resides on column c_i and $t_{R[i+1]}$ resides on column c_{i+1}, an exchange of i and $i+1$ will be done iff $c_i > c_{i+1}$. To this we add annihilation: if $R[i] = R[i+1]$ then both are replaced by nil.

The first step of clockwise transposition routing consists of compare-exchange between each node residing on an odd row with the node above it (if any). The second step consists of compare-exchange between each node residing on an odd column with the node to its right (if any). The third and fourth steps are similar to the first and second respectively, except that they involve the neighbors in the opposite direction. It is easily seen that each node simply performs compare-exchanges with its four neighbors in either clockwise or counterclockwise order.

We do not yet have a theoretical analysis of this algorithm. However, we have simulated it on numerous inputs of sizes up to $13\,000 \times 13\,000$ with random inputs drawn from a distribution mimicking that of the above mesh, as well as the simple distribution that puts a random value in every node. In all runs (except for very small meshes), we have not observed even a single case where the running time exceeded $2m$ steps. This is just two steps from the trivial lower bound $2m - 2$.

Our algorithm is a generalization of odd-even transposition sort, with a schedule that is identical to the "2D-bubblesort" algorithm of [8] but with different compare-exchange elements. The change from sorting to routing is indeed quite beneficial, as [8] shows that 2D-bubblesort is considerably slower than the observed performance of our clockwise transposition routing. The new algorithm appears to be much faster than the $8m$ sorting algorithm (due to Schimmler) used in [1], and its local control is very simple compared to the complicated recursive algorithms that achieve the $3m$-step lower bound on mesh sorting (cf. [16]).

A physical realization of the mesh will contain many local faults (especially for devices that are wafer-scale or larger, as discussed below). In the routing-based mesh, we can handle local defects by algorithmic means as follows. Each node shall contain 4 additional state bits, indicating whether each of its 4 neighbors is "disabled". These bits are loaded during device initialization, after mapping out the defects. The compare-exchange logic is augmented such that if node i has a "disabled" neighbor in direction Δ then i never performs an exchange in that direction, but always performs the exchange in the two directions orthogonal to Δ. This allows us to "close off" arbitrary rectangular regions of the mesh, such that values that reach a "closed-off" region from outside are routed along its perimeter. We add a few spare nodes to the mesh, and manipulate the mesh inputs such that the spare effectively replace the nodes of the in closed-off regions. We conjecture that the local disturbance caused by a few small closed-off regions will not have a significant effect on the routing performance.

Going back to the cost evaluation, we see that replacing the sorting-based mesh with a routing-based mesh reduces time by a factor of $3 \cdot 8/2 = 12$. Also, note that the $Q[\cdot]$ values are used just once per multiplication, and can thus be stored in slower DRAM cells in the vicinity of the node. DRAM cells are much smaller than edge-triggered flip-flops, since they require only one transistor and one capacitor per bit. Moreover, the regular structure of DRAM banks allows for very dense packing. Using large banks of embedded DRAM (which are shared by many nodes in their vicinity), the amortized chip area per DRAM bit is about $0.7\mu\text{m}^2$. Our Northwood-based estimates lead to $2.38\mu\text{m}^2$ per transistor, so we surmise that for our purposes a DRAM bit costs 1/3.4 as much as a logic transistor, or about 1/27 as much as a flip-flop. For simplicity, we ignore the circuitry needed to retrieve the values from DRAM — this can be done cheaply by temporarily wiring chains of adjacent $R[\cdot]$ into shift registers. In terms of circuit size, we effectively eliminate two of the three large registers per node, and some associated logic, so the routing-based mesh is about 3 times cheaper to manufacture. Overall, we gain a reduction of a factor $3 \cdot 12 = 36$ in the throughput cost.

5.5 An improved routing-based circuit. We now tweak the routing-based circuit design to gain additional cost reductions. Compared to the sorting-based design (cf. 5.3), these will yield a (constant-factor) improvement by several order of magnitudes. While asymptotically insignificant, this suggests a very practical device for the NFS matrix step of 1024-bit moduli. Moreover, it shows that already for 1024-bit moduli, the cost of parallelization can be negligible compared to the cost of the RAM needed to store the input, and thus the speed advantage is gained essentially for free.

The first improvement follows from increasing the density of targets. Let ρ denote the average number of $P[\cdot]$ registers per node. In the above scheme, $\rho = h^{-1} \approx 1/100$. The total number of $P[\cdot]$ registers is fixed at D, so if we increase ρ the number of mesh nodes decreases by $h\rho$. However, we no longer have enough mesh nodes to route all the hD nonzero entries of A simultaneously. We address this by partially serializing the routing process, as follows. Instead of storing one matrix entry $Q[\cdot]$ per node, we store $h\rho$ such values per node: for $\rho \geq 1$, each node j is "in charge" of a set of ρ matrix columns $C_j = \{c_{j,1}, \ldots, c_{j,\rho}\}$, in the sense that node j contains the registers $P[c_{j,1}], \ldots, P[c_{j,\rho}]$, and the nonzero elements of A in columns $c_{j,1}, \ldots, c_{j,\rho}$. To carry out a multiplication we perform $h\rho$ iterations, where each iteration consists of retrieving the next such nonzero element (or skipping it, depending on the result of the previous multiplication) and then performing clockwise transposition routing as before.

The second improvement follows from using block Wiedemann with a blocking factor $K > 1$ (cf. 2.5). Besides reducing the number of multiplications by a factor of roughly $\frac{20}{3}$ (cf. 5.3), this produces an opportunity for reducing the cost of multiplication, as follows. Recall that in block Wiedemann, we need to perform K multiplication chains of the form $A^k \vec{v}_i$, for $i = 1, \ldots, K$ and $k = 1, \ldots, 2D/K$, and later again, for $k = 1, \ldots, D/K$. The idea is to perform several chains in parallel on a single mesh, reusing most resources (in particular, the storage taken by A). For simplicity, we will consider handling all K chains on one mesh. In the routing-based circuits described so far, each node emitted at most one message per routing operation — a matrix row index, which implies the address of the target cell. The information content of this message (or its absence) is a single bit. Consider attaching K bits of information to this message: $\log_2(D)$ bits for the row index, and K bits of "payload", one bit per multiplication chain.

Combining the two generalizations gives the following algorithm, for $0 < \rho \leq 1$ and integer $K \geq 1$. The case $0 < \rho < 1$ requires distributing the entries of each matrix column among several mesh nodes, as in 5.4, but its cost is similar.

Let $\{C_j\}_{j \in \{1,\ldots,D/\rho\}}$ be a partition of $\{1, \ldots, D\}$, $C_j = \{c : (j-1)\rho \leq c-1 < j\rho\}$. Each node $j \in \{1, \ldots, D/\rho\}$ contains single-bit registers $P_i[c]$ and $P'_i[c]$ for all $i = 1, \ldots, K$ and $c \in C_j$, and a register R_j of size $\log_2(D) + K$. Node j also contains a list $Q_j = \{(r, c) \mid A_{r,c} = 1, c \in C_j\}$ of the nonzero matrix entries in the columns C_j of A, and an index I_j into C_j. Initially, load the vectors $\vec{v}_i$ into the $P_i[\cdot]$ registers. Each multiplication is then performed thus:

1. For all i and c, $P'_i[c] \leftarrow 0$. For all j, $I_j \leftarrow 1$.
2. Repeat $h\rho$ times:
   (a) For all j: $(r, c) \leftarrow Q_j[I_j]$, $I_j \leftarrow I_j + 1$, $R[j] \leftarrow \langle r, P_1[c], \ldots, P_K[c] \rangle$.
   (b) Invoke the clockwise transposition routing algorithm on the $R[\cdot]$, such that each value $R[j] = \langle r, \ldots \rangle$ is routed to the node t_j for which $r \in C_j$. During routing, whenever a node j receives a message $\langle r, p_1, \ldots, p_K \rangle$ such that $r \in C_j$, it sets $P'_i[r] \leftarrow P'_i[r] \oplus p_i$ for $i = 1, \ldots, K$ and discards the message. Moreover, whenever packets $\langle r, p_1, \ldots, p_K \rangle$ and $\langle r, p'_1, \ldots, p'_K \rangle$ in adjacent nodes are compared, they are combined: one is annihilated and the other is replaced by $\langle r, p_1 \oplus p'_1, \ldots, p_K \oplus p'_K \rangle$.
3. $P_i[c] \leftarrow P'_i[c]$ for all i and c.

After these steps, $P_i[\cdot]$ contain the bits of $A^k \vec{v}_i$ and the mesh is ready for the next multiplication. We need to compute and output the inner products $\vec{u}_j(A^k \vec{v}_i)$ for some vectors $\vec{u}_1, \ldots, \vec{u}_K$, and this computation should be completed before the next multiplication is done. In general, this seems to require $\Theta(K^2)$ additional wires between neighboring mesh nodes and additional registers. However, usually the $\vec{u}_j$ are chosen to have weight 1 or 2, so the cost of computing these inner products can be kept very low. Also, note that the number of routed messages is now doubled, because previously only half the nodes sent non-nil messages. However, empirically it appears that the clockwise transposition routing algorithm handles the full load without any slowdown.

It remains to determine the optimal values of K and ρ. This involves implementation details and technological quirks, and obtaining precise figures appears rather hard. We thus derive expressions for the various cost measures, based on parameters which can characterize a wide range of implementations. We then substitute values that reasonably represent today's technology, and optimize for these. The parameters are as follows:

– Let $\mathcal{A}_t$, $\mathcal{A}_f$ and $\mathcal{A}_d$ be the average wafer area occupied by a logic transistor, an edge-triggered flip-flop and a DRAM bit, respectively (including the related wires).
– Let $\mathcal{A}_w$ be the area of a wafer.
– Let $\mathcal{A}_p$ be the wafer area occupied by an inter-wafer connection pad (cf. 5.3).
– Let $\mathcal{C}_w$ be the construction cost of a single wafer (in large quantities).
– Let $\mathcal{C}_d$ be the cost of a DRAM bit that is stored off the wafers (this is relevant only to the FPGA design of Appendix A).
– Let $\mathcal{T}_d$ be the reciprocal of the memory DRAM access bandwidth of a single wafer (relevant only to FPGA).
– Let $\mathcal{T}_l$ be the time it takes for signals to propagate through a length of circuitry (averaged over logic, wires, etc.).
– Let $\mathcal{T}_p$ be the time it takes to transmit one bit through a wafer I/O pad.

We consider three implementation approaches: custom-produced "logic" wafers (as used in 5.3, with which we maintain consistency), custom-produced "DRAM" wafers (which reduce the size of DRAM cells at the expense of size and speed of logic transistors) and an FPGA-based design using off-the-shelf parts (cf. Appendix A). Rough estimates of the respective parameters are given in Table 2.

Table 2. Implementation hardware parameters

	Custom1 ("logic")	Custom2 ("DRAM")	FPGA
$\mathcal{A}_t$	2.38 μm^2	2.80 μm^2	0.05
$\mathcal{A}_f$	19.00 μm^2	22.40 μm^2	1.00
$\mathcal{A}_d$	0.70 μm^2	0.20 μm^2	$\emptyset$
$\mathcal{A}_p$	4 000 μm^2 $\times$ sec	4 000 μm^2 $\times$ sec	$\emptyset$
$\mathcal{A}_w$	$6.36 \cdot 10^{10}$ μm^2	$6.36 \cdot 10^{10}$ μm^2	25 660
$\mathcal{C}_w$	\$5,000	\$5,000	\$150
$\mathcal{C}_d$	$\emptyset$	$\emptyset$	\$$4 \cdot 10^{-8}$
$\mathcal{T}_d$	$\emptyset$	$\emptyset$	$1.1 \cdot 10^{-11}$ sec
$\mathcal{T}_p$	$4 \cdot 10^{-9}$ sec	$4 \cdot 10^{-9}$ sec	$2.5 \cdot 10^{-9}$ sec
$\mathcal{T}_l$	$1.46 \cdot 10^{-11}$ sec/μm	$1.80 \cdot 10^{-11}$ sec/μm	$1.43 \cdot 10^{-9}$ sec

$\emptyset$ marks values that are inapplicable, and taken to be zero.

The cost of the matrix step is derived with some additional approximations:

– The number of mesh nodes is D/ρ.
– The values in $Q_j[\cdot]$ (i.e., the nonzero entries of A) can be stored in DRAM banks in the vicinity of the nodes, where (with an efficient representation) they occupy $h\rho\log_2(D)\mathcal{A}_d$ per node.

– The $P_i[c]$ registers can be moved to DRAM banks, where they occupy $\rho K \mathcal{A}_d$ per node.
– The $P'_j[c]$ registers can also be moved to DRAM. However, to update the DRAM when a message is received we need additional storage. Throughout the D/ρ steps of a routing operation, each node gets 1 message on average (or less, due to annihilation). Thus $\log_2(\rho)+K$ latch bits per node would suffice (if they are still in use when another message arrives, it can be forwarded to another node and handled when it arrives again). This occupies $\rho K \mathcal{A}_f$ per node when $\rho < 2$, and $\rho K \mathcal{A}_d + 2(\log_2(\rho)+K)\mathcal{A}_f$ per node when $\rho \geq 2$.
– The bitwise logic related to the $P_i[c]$ registers, the $P'_i[c]$ and the last K bits of the $R[j]$ registers together occupy $20 \cdot \min(\rho, 2) K \mathcal{A}_t$ per node.
– The $R[j]$ registers occupy $(\log_2(D)+K)\mathcal{A}_f$ per node
– The rest of the mesh circuitry (clock distribution, DRAM access, clockwise transposition routing, I/O handling, inner products, etc.) occupies $(200 + 30\log_2(D))\mathcal{A}_t$ per node.
– Let $\mathcal{A}_n$ be total area of a mesh node, obtained by summing the above (we get different formulas for $\rho < 2$ vs. $\rho \geq 2$).
– Let $\mathcal{A}_m = A_n D/\rho$ be the total area of the mesh nodes (excluding inter-wafer connections).
– Let $\mathcal{N}_w$ be the number of wafers required to implement the matrix step, and let $\mathcal{N}_p$ be the number of inter-wafer connection pads per wafer. For single-wafer designs, $\mathcal{N}_w = 1/\lfloor \mathcal{A}_w/\mathcal{A}_m \rfloor$ and $\mathcal{N}_p = 0$. For multiple-wafer designs, these values are derived from equations for wafer area and bandwidth: $\mathcal{N}_w\mathcal{A}_w = \mathcal{A}_m + \mathcal{N}_w\mathcal{N}_p\mathcal{A}_p$, $\mathcal{N}_p = 4 \cdot 2 \cdot \sqrt{D/(\rho\mathcal{N}_w)} \cdot (log_2 D + K) \cdot \mathcal{T}_p/(\sqrt{\mathcal{A}_n}\mathcal{T}_l)$.
– Let $\mathcal{N}_d$ be total number of DRAM bits (obtained by evaluating $\mathcal{A}_m$ for $\mathcal{A}_f = \mathcal{A}_t = 0, \mathcal{A}_d = 1$).
– Let $\mathcal{N}_a$ be the number of DRAM bit accesses (reads+writes) performed throughout the matrix step. We get: $\mathcal{N}_a = 3D(2hDK + Dh\log_2(D))$, where the first term due to the the $P'_i[c]$ updates and the second term accounts for reading the matrix entries.
– Let $\mathcal{C}_s = \mathcal{N}_w\mathcal{C}_w + \mathcal{N}_d\mathcal{C}_d$ be the total construction cost for the matrix step.
– The full block Wiedemann algorithm consists of $3D/K$ matrix-by-vector multiplications, each of which consists of $h\rho$ routing operations, each of which consists of $2\sqrt{D/\rho}$ clocks. Each clock cycle takes $\mathcal{T}_l\sqrt{A_n}$.
  Let $\mathcal{T}_s$ be the time taken by the full block Wiedemann algorithm. We get: $\mathcal{T}_s = 6D^{3/2}h\mathcal{T}_l\sqrt{\rho\mathcal{A}_n}/K + \mathcal{N}_a\mathcal{T}_d/\mathcal{N}_w$.

Table 3 lists the cost of the improved routing-based circuit for several choices of ρ and K, according to the above. It also lists the cost of the sorting-based circuits (cf. 5.3) and the PC implementation of Appendix B. The lines marked by "(opt)" give the parameter choice that minimize the throughput cost for each type of hardware.

The second line describes a routing-based design whose throughput cost is roughly 45 000 times lower than that of the original sorting-based circuit (or 6 700 times lower than sorting with $K \gg 1$). Notably, this is a single-wafer device, which completely solves the technological problem of connecting multiple wafers with millions of parallel wires, as necessary in the original design of [1]. The third line shows that significant parallelism can be gained essentially for free: here, 88% of the wafer area is occupied simply by the DRAM banks needed to store the input matrix, so further reduction in construction cost seems impossible.

Table 3. Cost of the matrix step for the "small" matrix

Algorithm	Implementation	ρ	K	Wafers/ chips/ PCs	Construction cost $\mathcal{C}_s$	Run time $\mathcal{T}_s$ (sec)	Throughput cost $\mathcal{C}_s\mathcal{T}_s$ (\$ × sec)
Routing	Custom1	0.51	107	19	\$94,600	1440 (24 min)	$1.36{\cdot}10^8$ (opt)
Routing	Custom2	42.10	208	1	\$5,000	21 900 (6.1 hours)	$1.10{\cdot}10^8$ (opt)
Routing	Custom2	216.16	42	0.37	\$2,500	341 000 (4 days)	$8.53{\cdot}10^8$
Routing	Custom1	0.11	532	288	\$1,440,000	180 (3 min)	$2.60{\cdot}10^8$
Routing	FPGA	5473.24	25	64	\$13,800	15 900 000 (184 days)	$2.20{\cdot}10^{11}$(opt)
Routing	FPGA	243.35	60	2500	\$380,000	1 420 000 (17 days)	$5.40{\cdot}10^{11}$
Sorting	Custom1		1	273	\$4,100,000	1 210 000 (14 days)	$4.96{\cdot}10^{12}$
Sorting	Custom1		$\gg 1$	273	\$4,100,000	182 000 (50 hours)	$7.44{\cdot}10^{11}$
Serial	PCs		32	1	\$4,460	125 000 000 (4 years)	$5.59{\cdot}10^{11}$
Tree	PCs		32	66	\$24,000	2 290 000 (27 days)	$5.52{\cdot}10^{10}$

5.6 An improved circuit for the "large" matrix. The large matrix resulting from ordinary relation collection contains 250 times more columns: $D \approx 10^{10}$. We assume that the average column density remains $h = 100$. It is no longer possible to fit the device on a single wafer, so the feasibility of the mesh design now depends critically on the ability to make high bandwidth inter-wafer connections (cf. 5.3).

Using the formulas given in the previous section, we obtain the costs in Table 4 for the custom and FPGA implementations, for various parameter choices. The third line shows that here too, significant parallelism can be attained at very little cost (88% of the wafer area is occupied by DRAM storing the input). As can be seen, the improved mesh is quite feasible also for the large matrix, and its cost is a small fraction of the cost of the alternatives, and of relation collection.

Table 4. Cost of the matrix step for the "large" matrix

Algorithm	Implementation	ρ	K	Wafers/ chips/ PCs	Construction cost $\mathcal{C}_s$	Run time $\mathcal{T}_s$ (sec)	Throughput cost $\mathcal{C}_s\mathcal{T}_s$ (\$ × sec)
Routing	Custom1	0.51	136	6 030	\$30.1M	$5.04 \cdot 10^6$ (58 days)	$1.52{\cdot}10^{14}$(opt)
Routing	Custom2	4112	306	391	\$2.0M	$6.87 \cdot 10^7$ (2.2 years)	$1.34{\cdot}10^{14}$(opt)
Routing	Custom2	261.60	52	120	\$0.6M	$1.49 \cdot 10^9$ (47 years)	$8.95{\cdot}10^{14}$
Routing	Custom1	0.11	663	9000	\$500.0M	$6.40 \cdot 10^5$ (7.4 days)	$2.88{\cdot}10^{14}$
Routing	FPGA	17 757.70	99	13 567	\$3.5M	$3.44 \cdot 10^{10}$ (1088 years)	$1.19{\cdot}10^{17}$(opt)
Routing	FPGA	144 .41	471	$6.6 \cdot 10^6$	\$1000.0M	$1.14 \cdot 10^9$ (36 years)	$1.13{\cdot}10^{18}$
Serial	PCs		32	1	\$1.3M	270 000 years	$1.16{\cdot}10^{19}$
Tree	PCs		3484	813	\$153.0M	$3.17 \cdot 10^8$ (10 years)	$4.84{\cdot}10^{16}$

5.7 Summary of hardware findings. The improved design of 5.5 and 5.6, when implemented using custom hardware, appears feasible for both matrix sizes. Moreover, it is very attractive when compared to the traditional serial implementations (though appropriate parallelization techniques partially close this gap; see Appendix B). However, these conclusions are based on numerous assumptions, some quite optimistic. Much more research, and possibly actual relation collection experiments, would have to be carried out to get a clearer grasp of the actual cost (time and money) of both the relation collection and matrix steps for 1024-bit moduli.

In light of the above, one may try to improve the overall performance of NFS by re-balancing the relation collection step and the matrix step, i.e., by increasing the smoothness bounds (the opposite of the approach sketched in Remark 3.7). For ordinary NFS, asymptotically this is impossible since the parameters used for ordinary relation collection (i.e., the "large" matrix) already minimize the cost of relation collection (cf. 2.6). For improved NFS that is applied to a single factorization (cf. 2.7), if we disregard the cost of the matrix step and optimize just for relation collection then we can expect a cost reduction of about $L_{2^{1024}}[1/3, 1.9018836\cdots]/L_{2^{1024}}[1/3, 1.8689328\cdots] \approx 2.8$.

If many integers in a large range must be factored — a reasonable assumption given our interpretation of the throughput cost (cf. 3.2) — a much faster method exists (cf. [4]). It remains to be studied whether these asymptotic properties indeed hold for 1024-bit moduli and what are the practical implications of the methods from [4].

6 Conclusion

We conclude that methods to evaluate the security of RSA moduli that are based on the traditional operation count are not affected by the circuits proposed in [1]. Although the traditional estimates underestimate the difficulty of factoring, [1] provides yet another reason — other than the mostly historical reasons used so far — not to rely too much on supposedly more accurate cost-based estimates for the NFS.

We have shown that the suggestion made in [1] that the number of digits of factorable numbers has grown by a factor of 3, is based on an argument that may not be to everyone's taste. An alternative interpretation leads to a factor 1.17, under the cost function defined in [1]. The most traditional cost function, however, even leads to a factor 0.92.

Finally, we have presented an improved design for a mesh-based implementation of the linear algebra stage of the NFS. For an optimistically estimated 1024-bit factorization, our analysis suggests that a linear dependency between the columns of the sparse matrix can be found within a few hours by a device that costs about $5,000. At the very least, this is an additional argument not to rely on the alleged difficulty of the matrix step when evaluating the difficulty of factoring. As mentioned in [1] there are many other possibilities to be explored. Further study — and unbiased interpretation of the results — should eventually enable the cryptographic research and users communities to assess the true impact of [1] and the method proposed in 5.5.

Acknowledgments. We thank Daniel J. Bernstein for his constructive criticism [2]; we believe that these concerns are addressed by the present paper. We thank Martijn Stam for his assistance with the formulas in 2.7, Scott Contini for his careful reading and his insightful comments, and Yuliang Zheng for his kind cooperation. The first author thanks John Markoff for bringing [1] to his attention. The third author worked at Citibank when the first version of this paper was written.

References

1. D.J. Bernstein, *Circuits for integer factorization: a proposal*, manuscript, November 2001; available at `cr.yp.to/papers.html#nfscircuit`
2. D.J. Bernstein, *Circuits for integer factorization*, web page, July 2002; `http://cr.yp.to/nfscircuit.html`
3. S. Cavallar, B. Dodson, A.K. Lenstra, W. Lioen, P.L. Montgomery, B. Murphy, H.J.J. te Riele, et al., *Factorization of a 512-bit RSA modulus*, Proceedings Eurocrypt 2000, LNCS 1807, Springer-Verlag 2000, 1-17
4. D. Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology **6** (1993) 169-180
5. D. Coppersmith, *Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm*, Math. Comp. bf 62 (1994) 333-350
6. B. Dixon, A.K. Lenstra, *Factoring integers using SIMD sieves*, Proceedings Eurocrypt 1993, LNCS 765, Springer-Verlag 1994, 28-39
7. M. D. Grammatikakis, D. F. Hsu, M. Kraetzl, J. F. Sibeyn, *Packet routing in fixed-connection networks: a survey*, Journal of Parallel and Distributed Computing, 54(2):77-132, Nov. 1998
8. D. Ierardi, *2d-Bubblesorting in average time* $O(N \lg N)$, Proceedings 6th ACM symposium on Parallel algorithms and architectures, 1994
9. A.K. Lenstra, *Unbelievable security; matching AES security using public key systems*, Proceedings Asiacrypt 2001, LNCS 2248, Springer-Verlag 2001, 67–86
10. 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)
11. A.K. Lenstra, H.W. Lenstra, Jr., (eds.), *The development of the number field sieve*, Lecture Notes in Math. **1554**, Springer-Verlag 1993
12. A.K. Lenstra, E.R. Verheul, *Selecting cryptographic key sizes*, J. of Cryptology, **14** (2001) 255–293; available at www.cryptosavvy.com
13. P.L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2)*, Proceedings Eurocrypt'95, LNCS 925, Springer-Verlag 1995, 106–120
14. NIST, *Key management guideline – workshop document*, Draft, October 2001; available at csrc.nist.gov/encryption/kms
15. R.D. Silverman, *A cost-based security analysis of symmetric and asymmetric key lengths*, Bulletin 13, RSA laboratories, 2000;
available at www.rsasecurity.com/rsalabs/bulletins/index.html
16. C.P. Schnorr, A. Shamir, *An Optimal Sorting Algorithm for Mesh Connected Computers*, Proceedings 16th ACM Symposium on Theory of Computing, 255–263, 1986
17. G. Villard, *Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems (extended abstract)*, Proceedings 1997 International Symposium on Symbolic and Algebraic Computation, ACM Press, 32–39, 1997

18. D. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory, **IT-32** (1986), 54–62
19. M.J. Wiener, *The full cost of cryptanalytic attacks*, accepted for publication in J. of Cryptology

A Using off-the-shelf hardware for the circuit approach

In subsections 5.3–5.5 we were concerned primarily with custom-produced hardware, in accordance with the focus on throughput cost. In practice, however, we are often concerned about solving a small number of factorization problems. In this case, it may be preferable to use off-the-shelf components (especially if they can be dismantled and reused, or if discreteness is desired).

Tables 2–4 in Section 5.5 contain the parameters and cost estimates for off-the-shelf hardware, using the following scheme. FPGA chips are connected in a two-dimensional grid, where each chip holds a block of mesh nodes. The FPGA we consider is the Altera Stratix EP1S25F1020C7, which is expected to cost about \$150 in large quantities in mid-2003. It contains 2Mbit of DRAM and 25 660 "logic elements" that consist each of a single-bit register and some configurable logic. Since on-chip DRAM is scant, we connect each FPGA to several DRAM chips. The FPGA has 706 I/O pins that can provide about 70Gbit/sec of bandwidth to the DRAM chips (we can fully utilize this bandwidth by "swapping" large continuous chunks into the on-FPGA DRAM; the algorithm allows efficient scheduling). These I/O pins can also be used for communicating with neighbouring FPGAs at an aggregate bandwidth of 280Gbit/sec.

The parameters given in Table 2 are normalized, such that one LE is considered to occupy 1 area unit, and thus $\mathcal{A}_f = 1$. We make the crude assumption that each LE provides the equivalent of 20 logic transistors in our custom design, so $\mathcal{A}_t = 0.05$. Every FPGA chip is considered a "wafer" for the purpose of calculation, so $\mathcal{A}_w = 51\,840$. Since DRAM is located outside the FGPA chips, $\mathcal{A}_d = 0$ but $\mathcal{C}_d = 4 \cdot 10^8$, assuming \$320 per gigabyte of DRAM. $\mathcal{T}_d$ and $\mathcal{T}_p$ are set according to available bandwidth. For $\mathcal{T}_l$ we assume that on average an LE switches at 700MHz. $\mathcal{A}_p = 0$, but we need to verify that the derived $\mathcal{N}_p$ is at most 706 (fortunately this holds for all our parameter choices).

As can be seen from the tables, the FPGA-based devices are significantly less efficient than both the custom designs and properly parallelized PC-based implementation. Thus they appear unattractive.

B The traditional approach to the matrix step

We give a rough estimate of the price and performance of a traditional implementation of the matrix step using the block Lanczos method [13] running on standard PC hardware. Let the "small" and "large" matrices be as in 5.1.

B.1 Processing the "small" matrix using PCs. A bare-bones PC with a 2GHz Pentium 4 CPU can be bought for \$300, plus \$320 per gigabyte of RAM. We will use block Lanczos with a blocking factor of 32, to match the processor word size. The $hD = 4 \cdot 10^9$ nonzero entries of the "small" matrix require 13GB of storage, and the auxiliary D-dimensional vectors require under 1GB. The construction cost is thus about \$4,500.

The bandwidth of the fastest PC memory is 4.2GB/sec. In each matrix-by-vector multiplication, all the nonzero matrix entries are read, and each of these causes an update (read and write) of a 32-bit word. Thus, a full multiplication consists of accessing $hD \log_2(D) + 2hD \cdot 32 = 4.8 \cdot 10^{10}$ bits, which takes about 11 seconds. The effect of the memory latency on non-sequential access, typically 40n, raises this to about 50 seconds (some reduction may be possible by optimizing the memory access pattern to the specific DRAM modules used, but this appears nontrivial). Since $2D/32$ matrix-by-vector multiplications have to be carried out [13], we arrive at a total of $1.25 \cdot 10^8$ seconds (disregarding the cheaper inner products), i.e., about 4 years.

The throughput cost is $5.6 \cdot 10^{11}$, which is somewhat better than the sorting-based mesh design (despite the asymptotic advantage of the latter), but over 5000 times worse than the the single-wafer improved mesh design (cf. 5.5). Parallelization can be achieved by increasing the blocking factor of the Lanczos algorithm — this would allow for different tradeoffs between construction cost and running time, but would not decrease the throughput cost.

B.2 Processing the "large" matrix using PCs. The large matrix contains 250 times more columns at the same (assumed) average density. Thus, it requires 250 times more memory and $250^2 = 62\,500$ times more time than the small matrix. Moreover, all row indices now occupy $\lceil \log_2 10^9 \rceil = 34$ bits instead of just 24. The cost of memory needed to store the matrix is \$1.36M (we ignore the lack of support for this amount of memory in existing memory controllers), and the running time is 270 000 years. This appears quite impractical (we cannot increase the blocking factor by over $\sqrt{D}$, and even if we could, the construction cost would be billions of dollars).

Remark B.3 Once attention is drawn to the cost of memory, it becomes evident that better schemes are available for parallelizing a PC-based implementation. One simple scheme involves distributing the matrix columns among numerous PCs such that each node j is in charge of some set of columns $C_j \subset \{1, \ldots, D\}$, and contains only these matrix entries (rather than the whole matrix). The nodes are networked together with a binary tree topology. Let $\vec{a}_i$ denote the i-th column of A. Each matrix-by-vector multiplication $A\vec{w}$ consists of the root node broadcasting the bits $w_1, \ldots, w_D$ down the tree, each node j computing a partial sum vector $\vec{r}_j = \sum_{i \in C_j, w_i = 1} \vec{a}_i \pmod 2$, and finally performing a converge-cast operation to produce the sum $\sum_j \vec{r}_j = A\vec{w} \pmod 2$ at the root. If the broadcast and converge-cast are done in a pipelined manner on 0.5 gigabit links, this is easily seen to reduce the throughput cost to roughly $5.6 \cdot 10^{10}$ for the small matrix and $4.8 \cdot 10^{16}$ for the large matrix (see Tables 3,4).

For constant-bandwidth links, this scheme is asymptotically inefficient since its throughput cost is $L(3\beta)$. However, for the parameters considered it is outperformed only by the custom-built improved mesh.