Analysis of Bernstein's Factorization Circuit

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

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

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

[3] Citibank, N.A., 850 Third Avenue, 12th floor, New York, NY 10022, U.S.A.
james.tomlinson@citicorp.com

Abstract. In [1], Bernstein proposed a circuit-based implementation of the matrix step of the number field sieve factorization algorithm. We show that under the non-standard cost function used in [1], these circuits indeed offer an asymptotic improvement over other methods but to a lesser degree than previously claimed: for a given cost, the new method can factor integers that are 1.17 times larger (rather than 3.01). 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. 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 [9, 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. Let $\alpha > 0$, $\beta > 0$, r, and s be fixed real numbers with $0 <$

$s < r \leq 1$. A random positive integer $\leq L_x[r; \alpha]$ 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 $\leq L[2/3, \alpha]$ 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 [10], 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, and smoothness bounds B_r and B_a. Other properties that these parameters must satisfy 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

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

(cf. [10]), where $\pi(x)$ is the number of primes $\leq x$. 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% that a factor of n is found in the final step, the square root step. We discuss in more detail 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. Since $\pi(L(\beta)) = L(\beta)$ and $2L(\beta) = L(\beta)$, it follows 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

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

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

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

Remark 2.4 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 [9, 4.15] and [3]). 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. Intermediate solutions that exchange sieving memory for many tightly coupled processors with small memories could prove valuable too; see [5] 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.

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 [4, 12], 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. As long as the parallelization factor K is reasonably small, the techniques referred to allow a K-fold parallelization that results in a K-fold speedup. For large K auxiliary steps cause significant overhead. We do not elaborate.

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 $L(2\alpha)$ (Remark 2.4).

2.6 NFS parameter optimization for general matrix exponent $2\epsilon > 1$. We balance the cost of the relation collection and matrix steps by taking $\alpha \overset{o}{=} \epsilon\beta$. With (1) it follows that

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

Minimizing β given ϵ leads to

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

and

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

Minimizing the resulting

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

leads to $\epsilon = 1$ and $\alpha \overset{o}{=} 2/3^{2/3}$: 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:

$1 < 2\epsilon \leq 2$:

$\delta \doteq 3^{1/3}$, $\alpha \doteq 2/3^{2/3}$, and $\beta \doteq 2/3^{2/3}$, 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 [10]. 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.

$2\epsilon = 5/2$:

$\delta \doteq (5/3)^{1/3}(6/5)$, $\alpha \doteq (5/3)^{1/3}(5/6)$, and $\beta \doteq (5/3)^{1/3}(2/3)$, 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 $(1.9229994\cdots/1.9760518\cdots + o(1))^3 \doteq 0.9216$ 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).

$2\epsilon = 3$:

$\delta \doteq 2/3^{1/3}$, $\alpha \doteq 3^{2/3}/2$, and $\beta \doteq 3^{-1/3}$, 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 [3] that ordinary NFS from [10], and as used in 2.2, can be improved by using more than a single polynomial f. That method is used in [1] to select parameters for standard-NFS but the ordinary method is used in [1] for circuit-NFS (cf. 3.1). The reason is that for $2\epsilon = 2$, the method from [3] gives an improvement over the ordinary method, namely $L(1.9018836\cdots)$, but that for $2\epsilon > 7/3$ (as in circuit-NFS) usage of more polynomials no longer leads to an improvement. This follows from a more elaborate analysis that has no further relevance for this paper. From that same analysis it follows that, unlike the ordinary parameter selection method, optimal relation collection for the improved method occurs for $2\epsilon < 2$: with $2\epsilon = 1.7804291\cdots$ the operation count for relation collection becomes $L(1.8688796\cdots)$. 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 (i.e., $\epsilon = 0$) and minimize the operation count of relation collection, this method yields a cost of $L(1.8688\cdots)$.

3 The circuits for integer factorization from [1]

3.1 Matrix-by-vector multiplication based on 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 v. An elegant method is given that computes the product Av 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 Av 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 Av. We use "circuit-NFS" to refer to NFS that uses the mesh-sorting-based matrix step.

3.2 The multiplicative cost function from [1]. Judging by operation counts, the mesh-based algorithm is not competitive with the traditional way of computing Av: 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 new cost function as **multiplicative cost**.

If we interpret the multiplicative cost as measuring the expenses per unit of throughput, we see that it 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 the multiplicative 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 $10M 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 is apparently a far 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 multiplicative 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 multiplicative 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 multiplicative 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 multiplicative cost of $L(3\beta)$. Thus, the multiplicative 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 multiplicative 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 multiplicative 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 multiplicative 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 multiplicative cost. We consider the implication of the matrix step multiplicative 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 multiplicative cost of relation collection is $L(2\alpha)$ (Remark 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 multiplicative 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 multiplicative cost $L(2\alpha)$. Thus, for relation collection the multiplicative 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 multiplicative cost of $L(1.9760518\cdots)$. As mentioned above and in 3.2, the multiplicative 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 multiplicative 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 multiplicative 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

(Remark 2.4). In any case, combined with $L(3\beta)$ for the multiplicative 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 [3] mentioned in 2.7 is used, leading to a multiplicative 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. Interestingly, in [3], in order to achieve the $L(1.9018836\cdots)$ operation count, the "memory-free" relation collection method is used to search for B_a-smooth numbers (cf. 2.2). Since memory usage was no concern for the analysis in [3], regular "memory-wasteful" sieving was suggested to test B_r-smoothness.

3.6 Alternative interpretation. How does the comparison between circuit-NFS and standard-NFS with respect to their multiplicative costs turn out if standard-NFS is first properly tuned (Remark 3.4) to the multiplicative cost function, given the state of the art in, say, 1990 (cf. [9, 4.15]; also the year that [3] originally appeared)? With multiplicative cost $L(2\alpha)$ for relation collection (cf. above and Remark 2.4), the analysis from 2.6 with $2\epsilon = 3$ applies, resulting in a multiplicative 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 multiplicative cost of D-digit integers using standard-NFS. The significance of this comparison depends on whether or not the multiplicative 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 [11] and [8], and are implied by [13].

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 fits conveniently and that causes 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.

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 multiplicative costs. For the other operation count the multiplicative 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.

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

immediate

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 multiplicative 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 multiplicative costs for the matrix steps of standard-NFS and circuit-NFS, β must be chosen such that $L(3\beta) = L((5/3)^{4/3}) = L(1.9760518\cdots)$, so that the matrix step of standard-NFS would require $L(\beta) = L(0.6586839\cdots)$ RAM and real time $L(2\beta) = L(1.3173678\cdots)$. Substituting this β in Relation (1) and minimizing α with respect to δ we find

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

i.e., $\delta \triangleq 1.3675203\cdots$ and $\alpha \triangleq 1.0694546\cdots$, resulting in relation collection operation count $L(2.1389093\cdots)$. Or, one could match the real time of the matrix steps: with $L(2\beta) = L((5/3)^{1/3}) = L(1.1856311\cdots)$ the matrix step of standard-NFS would require $L(0.5928155\cdots)$ RAM and real time $L(1.1856311\cdots)$. With Relation (5) we find that $\delta \triangleq 1.3195267\cdots$, $\alpha \triangleq 1.1486654\cdots$, and relation collection operation count $L(2.2973309\cdots)$.

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 [10] and [3] 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 [13] and the "computationally equivalent" estimates in [8, 11]. Security estimates based on the recommendations from [13] or the main ones (i.e., the conservative "computationally equivalent" ones) from [8, 11] are therefore not affected by the result from [1]. The estimates from [14], however, 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 [8, 11] 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. In the appendices we give similar estimates using off-the-shelf hardware, for both the traditional and circuit-based approaches.

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 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 [2]. 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 [2]. 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 time but less memory when using alternative smoothness tests). 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 this 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 difficulty of the relation collection step.

5.3 Processing the "small" matrix using circuits. We estimate the size of the circuit required to implement the mesh of [1] when the NFS parameters are optimized for the multiplicative 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 subsection will make several improvements, including those listed as future plans in [1].

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 $m = 2^{16}$. 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 v, is loaded just once so this can be done serially. The output, namely the D-bit vector $A^k v$ for $k \in \{0, \ldots, D-1\}$, is produced once per matrix-by-vector multiplication. Since all that is needed for finding the dependency is the bit sequence $uA^k v$ for some vector u, we add circuitry that computes this inner product given $A^k v$ so that the mesh emits a single bit after each multiplication.

In standard 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 $8.6 \cdot 10^{12}$ transistors in the mesh.

As a representative of current technology available on large scale we take Intel's latest Pentium processor, the Pentium 4 "Northwood" (0.13μm 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 $\left(8.6 \cdot 10^{12}\right) / \left(5.5 \cdot 10^7\right) \approx 157\,000$ Northwood-sized dies or about 295 wafers. Using the above per-chip price figure naively, the construction cost is about \$22M. Alternatively, assuming a wafer cost of about \$5,000 we get a construction cost of roughly \$1.5M, and the initial costs (e.g., mask creation) is under \$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, each matrix-by-vector multiplication requires $3 \cdot 8m \approx 1.57 \cdot 10^6$ steps. Assuming that each step takes a single clock cycle at a 1GHz clock rate, we get a throughput of 635 multiplications per second. The number of required multiplications is expected to be D, so the expected total running time is $4 \cdot 10^7/635 \approx 63\,000$ seconds, or 17.5 hours.

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. Thus the construction cost increases by a factor of 2.5. 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.

Heat dissipation (i.e., power consumption) may limit the node density and clock rate of the device, and needs to be analyzed. 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 An improved circuit. The above analysis refers to the original sorting-based mesh circuit as described in [1]. We now present an alternative design that yields a (constant-factor) improvement of several orders of magnitude. While asymptotically insignificant, this shows that already for 1024-bit moduli the cost of parallelization is negligible compared to the cost of the RAM needed to store the input. Thus the speed advantage of parallelism is gained essentially for free.

One of the novel ideas presented in [1] is the use of mesh sorting for matrix-by-vector multiplication. We improve upon this by replacing the three sorting operations by a single routing operation. Beside the reduction in the number of global operations, we also benefit from routing being easier than sorting (intuitively, because the final position of each element is independent of the rest; this intuition is supported by applicable lower bounds). This change lets us improve all aspects of the circuit: speed, size, complexity of local control, and fault resistance.

For simplicity assume that each of the D columns of the matrix has weight exactly h (here $h = 100$), and 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 ($i \in \{1, \dots D\}$), 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 v, the mesh is initialized as follows: the i-th entry of v is loaded into $P[i]$, and the row indices of the nonzero elements in column $i \in \{1, \dots, D\}$ 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 $\{1, \dots, D\}$).
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. For the purpose of kernel computation we are interested only in $uA^k v$ for some vector u, so we can load each u_i into node t_i during initialization, and compute the single-bit result $uA^k v$ inside the mesh during the next multiplication.

There remains the choice of a routing algorithm. Many candidates exist (see [6] 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 a 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 rows (for vertically adjacent nodes) or target columns (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 the 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-exchange with its four neighbors, either in clockwise or in 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 $10\,000 \times 10\,000$ 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 [7] but with different compare-exchange elements. The change from sorting to routing is indeed quite beneficial, as [7] 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. [15]).

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 multiplicative cost.

Additional reduction can be obtained by 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 becomes $h\rho$ times smaller and the routing time decreases by a factor of $\sqrt{h\rho}$ (in fact we can expect the performance of the routing algorithm to degrade due to fewer opportunities for annihilation, but empirically we have not observed any appreciable effect for $\rho \leq 256$). However, we can no longer route all the hD nonzero entries of A simultaneously because the mesh is too small. We solve 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 i is "in charge" of some ρ matrix columns $c_{i,1}, \ldots, c_{i,\rho}$, in the sense that node i contains the registers $P[c_{i,1}], \ldots, P[c_{i,\rho}]$, and loads the content of matrix columns $c_{i,1}, \ldots, c_{i,\rho}$ into $R[i]$ one by one. The matrix entries are stored in DRAM banks close to the respective nodes. To carry out a multiplication we perform $h\rho$ iterations, where each iteration consists of retrieving the next value from the DRAM (or skipping it, depending on the result of the previous multiplication) and then performing clockwise transposition routing as before. Thus the total number of steps per multiplication is increased by a factor of $\sqrt{h\rho}$. As ρ increases the manufacturing cost becomes dominated by the hD DRAM cells that store the entries of A, so asymptotically the reduced parallelism is indeed unfavorable. A small increase is still beneficial, as follows.

Setting $\rho = 400h^{-1} \approx 4$ (i.e., 4 targets per cell) increases the total number of steps by a factor of $\sqrt{400} = 20$ and decreases the number of mesh nodes by a factor of 400 (i.e., $m = \sqrt{D/4} \approx 3163$). The area occupied by each node increases by a factor of roughly 6, since to the 2000/3 existing transistors per node we add 3 more single-bit target registers, a little logic and $400 \cdot 26 = 10\,400$ DRAM bits which take as much room as 3 058 transistors. Overall, this yields a decrease of a factor of 3.3 in the multiplicative cost. Also, the circuit is now smaller by a factor of $3 \cdot (400/6) \approx 200$ compared to our estimates for the sorting-based mesh, and thus occupies just $295/200 \approx 1.5$ wafers.

For these parameters we can in fact squeeze the mesh into a single wafer, by switching to a manufacturing process that is specialized for DRAM chips. This decreases the amortized area per DRAM bit by a factor of about 3 (to about $0.2\mu\mathrm{m}^2$). The different process entails a 30% reduction in the clock rate and a slight increase in the size of logic transistors. None the less, the full mesh now comfortably fits on a single wafer with just half a dozen external connections.

This completely solves the technological problem of connecting multiple wafers with millions of parallel wires, as necessary in the original design of [1]. Note that most of the silicon real-estate is now occupied simply by the DRAM banks needed to store the input matrix A, and thus further reduction in construction cost seems impossible.

Some speedup follows from noting that at the beginning of each multiplication, only half of the nodes have packets to route (i.e., if the previous multiplication result yielded $P[i] = 0$ then the node t_i will not send any values during this multiplication). At negligible cost, we can do some local smoothing before each invocation of the clockwise transposition routing, such that there are always nearly m^2 values to route. This reduces the number of invocations of clockwise transposition routing by a factor of 2, essentially for free. Also, this allows us to relax the assumption that the weight of all columns is exactly $h \approx 100$.

In current technology, wafers normally have some local defects. When the wafer is sliced into separately packaged dies the defective ones can be discarded, but for a wafer-scale device this is not an option. In our case, 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 DRAM and $P[\cdot]$ registers of the spare nodes replace those of nodes 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.

Combining all of the above, we see that based on our estimation of the relevant parameters, the matrix step for a 1024-bit RSA moduli can be completed by a single-wafer device within time $63\,000 \cdot 20/12/2/0.7 = 75\,500$ seconds, or about 21 hours. In large quantities this device would cost about \$5,000.

5.5 An improved circuit for the "large" matrix. The large matrix resulting from ordinary relation collection contains 250 times more columns. Assuming that the average column density remains 100, the mesh would thus require about 250 full wafers. Inter-wafer connections can be achieved by the means described in 5.3, and moreover the number of connections is much lower in our design (the number of lines per node is halved due to the use of routing, and the number of cells per wafer is decreased due to $\rho \gg h^{-1}$). Our design can handle sporadic failures (in nodes and in inter-wafer connections) by algorithmic means. Adapting the routing algorithm to inter-wafer delays also appears possible, but will not be discussed here.

At \$5,000 per wafer, and after doubling of the number of wafer due to inter-wafer connection overhead, the cost of the large mesh would be \$2.5M. The running time would be $250^{3/2} \cdot 21 \approx 83\,000$ hours, or about 10 years. Restoring $\rho = h^{-1}$ would reduce this to 175 days and increase the cost to \$172M.

5.6 Summary of custom hardware findings. The improved design of 5.4 and 5.5 appears feasible for both matrix sizes, and very attractive when compared to the traditional serial implementations (see Appendix B). However, this is 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 $L_{2^{1024}}[1/3, 1.9018\cdots]/L_{2^{1024}}[1/3, 1.8688\cdots] \approx 2.8$.

If many integers in a large range must be factored — a reasonable assumption given our interpretation of the multiplicative cost (cf. 3.2) — a much faster method exists (cf. [3]). 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 [3].

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 in less than a day 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 5.4.

Acknowledgments. We thank Scott Contini for his careful reading of our manuscript and his insightful comments. The first author thanks John Markoff for bringing [1] to his attention.

References

1. D.J. Bernstein, *Circuits for integer factorization: a proposal*, manuscript, November 2001; available at cr.yp.to/papers.html#nfscircuit
2. 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
3. D. Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology **6** (1993) 169-180
4. D. Coppersmith, *Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm*, Math. Comp. bf 62 (1994) 333-350
5. B. Dixon, A.K. Lenstra, *Factoring integers using SIMD sieves*, Proceedings Eurocrypt 1993, LNCS 765, Springer-Verlag 1994, 28-39
6. 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
7. D. Ierardi, *2d-Bubblesorting in average time* $O(N \lg N)$, Proceedings 6th ACM symposium on Parallel algorithms and architectures, 1994
8. A.K. Lenstra, *Unbelievable security; matching AES security using public key systems*, Proceedings Asiacrypt 2001, LNCS 2248, Springer-Verlag 2001, 67-86
9. 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)
10. A.K. Lenstra, H.W. Lenstra, Jr., (eds.), *The development of the number field sieve*, Lecture Notes in Math. **1554**, Springer-Verlag 1993
11. A.K. Lenstra, E.R. Verheul, *Selecting cryptographic key sizes*, J. of Cryptology **14** (2001) 255-293; available at www.cryptosavvy.com
12. P.L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2)*, Proceedings Eurocrypt'95, LNCS 925, Springer-Verlag 1995, 106-120
13. NIST, *Key management guideline – workshop document*, Draft, October 2001; available at csrc.nist.gov/encryption/kms
14. 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

15. C.P. Schnorr, A. Shamir, *An Optimal Sorting Algorithm for Mesh Connected Computers*, Proceedings 16th ACM Symposium on Theory of Computing, 255–263, 1986

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

We estimate the price and performance of the improved mesh design of 5.4 when off-the-shelf hardware is used to process the "small" matrix defined in 5.1.

We will sketch a design based on a combination of FPGA chips and DRAM memory chips. The FPGA we consider is the Altera EP20K1500E, currently one of the largest on the market. It contains 51 840 "logical elements" that consist each of a single-bit register and some configurable logic. It has 808 I/O pins which can be operated at 250MHz.

Recall that in the improved mesh design for the "small" matrix (cf. 5.4), each node consists of a 26-bit register $R[\cdot]$, ρ single-bit target registers, some control and compare-exchange logic, and about $h\rho$ 26-bit values in seldom-accessed DRAM (in a DRAM bank that is shared with other nodes in the vicinity). A few additional state bits are needed, but these can be shared by several adjacent nodes. We assume that one logical element per state bit suffices, i.e., that the configurable logic is sufficiently powerful. We set $\rho = 196$, so that each chip contains a block of 15×15 mesh nodes. These 225 nodes require $225 \cdot 196 \cdot 100 \cdot 26$ DRAM bits, i.e., 115Mbit. We thus attach one 128Mbit DRAM chip to each FPGA chip (the FPGA has the appropriate interface).

The mesh consists of $4 \cdot 10^7/\rho = 204\,081$ nodes in a 451×451 configuration. Thus each routing operation takes about 900 steps. With $h\rho/2 = 19\,600$ routing operations per multiplication and $4 \cdot 10^7$ multiplications needed, the total number of steps is $7 \cdot 10^{14}$ (disregarding the loading of values from DRAM). Due to the limited number of I/O pins, each step requires 4 clock cycles at 250MHz, hence the total running time is about 4 900 000 seconds or 57 days.

With $451^2/15^2 = 905$ FPGAs at \$5,000 each and 905 DRAM chips at a negligible cost, the total equipment cost is \$4.5M. While time and equipment cost can be traded by changing the parameter ρ, this is considerably more expensive than the custom-built circuit (21 hours operation, \$5,000 per unit and initial costs under \$1M).

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 method described in [3] and Intel's Itanium microprocessor. Let the "small" and "large" matrices be as in 5.1.

B.1 Processing the "small" matrix using PCs. The Itanium is a 64-bit processor that costs \$4,000 and runs at 1GHz. The nonzero entries of the "small" matrix can be stored in $4 \cdot 10^9 \cdot 4$ bytes, i.e., 16GB. With a price tag of \$160 for half a gigabyte, \$25,000 buys us four Itaniums and enough shared memory to store the matrix and several auxiliary D-dimensional vectors. The bandwidth of the fastest PC memory is 3.2GB/sec, or $4 \cdot 10^8$ 64-bit words per second. In the matrix-by-vector multiplication, each nonzero matrix entry causes 2 memory reads and one memory write, so a full multiplication consisting of $4 \cdot 10^9$ operations on 64-bit integers can be done in about 30 seconds. The effect of the memory latency (typically 40ns) raises this to at least 100 seconds. Since $3 \cdot 4 \cdot 10^7/64$ matrix-by-vector multiplications have to be carried out [3], we arrive at a total of $1.9 \cdot 10^8$ seconds (disregarding the cheaper inner products), i.e., about 6 years.

To match the 21 hours run time of the improved circuit (cf. 5.4) we can make the blocking factor 2500 times wider, thereby increasing the cost to over \$62M. By contrast, the circuit would cost about \$5,000, with initial costs under \$1M.

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. Memory cost to store the matrix works out to about \$1.3M (we ignore the lack of support for this amount of memory in existing memory controllers), and the running time would total 375 000 years. Note that the cost is comparable to that of the improved circuit (cf. 5.5), but the time is 37 500 times longer. Matching the 10 years running time of the improved circuit would require \$48B worth of PCs.