Improved Routing-Based Linear Algebra for the Number Field Sieve

Willi Geiselmann, Hubert Köpfer, Rainer Steinwandt
IAKS, Arbeitsgruppe Systemsicherheit, Prof. Beth,
Universität Karlsruhe, 76 131 Karlsruhe, Germany

Eran Tromer
Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot 76100, Israel

January 20, 2005

Abstract

Recently, efficient custom-hardware designs were proposed for the linear algebra step of the Number Field Sieve integer factoring algorithm. These designs make use of a heuristic mesh routing algorithm, whose performance has been analyzed only experimentally. We show that the routing algorithm may in fact encounter livelocks, i.e., may never terminate. We propose a context-specific solution for this problem. We also show how efficiency can be improved by a factor of ≈ 4 through the use of a parallel tori topology and packet injection during routing.

1 Introduction

The best known algorithm for factoring large integers, the Number Field Sieve (NFS), involves two major computational steps: the *relation collection (sieving) step* and the *linear algebra step*. These steps effectively determine the cost of breaking factoring-based cryptosystems such as RSA in the absense of theoretical breakthroughs. Consequentially, several attempts have been made to expedite them through the use of specialized hardware (e.g., [10, 7, 1, 8, 3, 11, 4, 5]).

The most recent, and currently most promising, proposals for dedicated hardware supporting the linear algebra step [4, 8] are based on a two-dimensional mesh routing algorithm known as *clockwise transposition routing*. The reported experimental results suggest that this algorithm is extraordinarily efficient, in terms of both running time and circuit size; however, no theoretical analysis has been given yet, and in this contribution we show that in fact livelocks can occur: for certain inputs the algorithm enters an infinite loop and fails to terminate. During the NFS linear algebra step, the clockwise transposition routing would be executed numerous times, and the failure of any such execution would be devastating.

We propose a context-specific solution, which does not significantly increase the circuit size. While a sound theoretical analysis of clockwise transposition routing remains absent, the proposed solution suffices to restore the confidence in the claimed performance of the aforementioned hardware designs, and should be considered by their potential implementors. In addition, we show how the routing circuit can be made ≈ 4 times more efficient by use of a parallel tori topology (adapted from a previously proposed mesh-based sieving circuit [5]).

Section 2 reviews clockwise transposition routing and its context in the NFS. Section 3 demonstrates the existence of livelocks and describes our solution. Section 4 describes the efficiency improvements, and Section 5 provides some experimental simulation results.

2 Clockwise transposition routing and NFS

2.1 The NFS and the linear algebra step

For a full description of the NFS algorithm, the reader is referred to [6]. We will focus on the linear algebra (LA) step. Its input is a large (w.l.o.g., square) sparse binary matrix $A \in \mathrm{GF}(2)^{r\times r}$, and the task is to find a nonzero vector in the kernel of A, i.e., $\vec{x}$ such that $A\vec{x} = \vec{0}$ over GF(2). The difficulty stems from the matrix size r being subexponential in the length of the number being factored.

The most recent proposals for custom hardware implementation of the linear algebra step are the design of Lenstra, Shamir, Tomlinson, and Tromer [8] and its distributed version by Geiselmann and Steinwandt [4]. These designs make use Coppersmith's block Wiedemann algorithm [12, 2] for finding kernel vectors. With this algorithm, cost is dominated by the computation of long vector chains of the form $A\vec{v}, A^2\vec{v}, \ldots, A^k\vec{v}$, where $\vec{v} \in \mathrm{GF}(2)^r$ is a binary vector (which can be chosen nearly arbitrarily). Note that this can be achieved by k matrix-by-vector multiplications, with a fixed matrix and changing vector. Overall, we need to carry out $3r$ multiplications divided into $\Theta(K)$ chains, where K is the *blocking factor*.[1]

Both the original device by [8] and its distributed version [4] reduce the matrix-vector multiplications to a mesh routing problem. The essential idea is as follows. The circuit consists of a two-dimensional $m \times m$ mesh of cells, where m^2 is (approximately) the number of non-zero entries in A. For every $i, j \in \{1, \ldots, r\}$ with $A_{i,j} = 1$, the pair (i, j) is stored in some distinct cell of A. The storage of pairs is such that for every $j \in \{1, \ldots, r\}$, all pairs (i, j) are stored close together; this cluster corresponds to the j-th column of A. One of the cells in each cluster is designated a *target cell*, denoted T_j. It initially stores the bit u_j, where u is the vector input of the current multiplication. To perform a multiplication, start by performing local communication within each cluster j, identifying the cells containing a pair (i, j) such that $u_j = 1$. Each such cell "emits" a packet with the value i. Then, perform a global mesh routing operation which carries packet i to the target cell T_i. The total number of pairs reaching T_i gives the i-th coordinate of the multiplication result (mod 2). To perform the next multiplication, just repeat the last 2 steps; there is no need to reload the inputs.

[1]For very small K, the number of multiplications is somewhat larger due to the possibility of failure and the need for multiple kernel elements.

pins	K	ρ	m^2	#chips	size (mm^2)	time
1280	69	1982	128^2	308^2	36^2	357 d
2560	69	1982	128^2	308^2	36^2	181 d
2048	31	1551	254^2	100^2	50^2	722 d
4096	31	1551	254^2	100^2	50^2	383 d

Table 1: Estimates for the computational effort of the LA step of the NFS.

Note that each multiplication uses the result of the previous one, so any failure will ruin all subsequent computations. Hence, correctness of the routing algorithm is paramount.[2] Several modifications are made to the above in the sake of efficiency. Here we note just one of these: one can reduce the size of the mesh and store several matrix entries at each cell. Each cell then emits multiple packets (corresponding to the matrix entries it contains), so more routing operations are needed, but each one is faster. The trade-off is captured by the parameter $\rho = r/m^2$.

2.2 Implementation cost estimates

For concreteness, we give an example of estimated implementation costs using a set of parameters discussed in [8, 4].[3] Namely, we assume that the matrix $A \in \mathrm{GF}(2)^{r\times r}$ has $r = 10^{10}$ rows and columns with 100 non-zero entries per column. With these parameters the original design of [8] appears impractical for current technology, as it would necessitate high-bandwidth connections among dozens of wafer-scale circuits. The distributed version of [4] solves this hurdle; its cost is estimated in Table 1 for some parameter choices, assuming 0.13 μm CMOS process technology and a 200 MHz clock rate. These chip sizes, albeit being large in comparison to standard chips, do not appear utopian for a regular systolic design that is inherently fault-tolerant.[4] In the table, "pins" is the number of full-speed single-bit unidirectional I/O lines per chip, K and ρ are as defined in Section 2.1, m^2 is the number of cells in the mesh, "#chips" is the number of (separate but connected) chips needed, "size" the area of a single chip, and "time" estimates the total time needed for the LA step in days.

2.3 Clockwise transposition routing

The main step in the above realization of the block Wiedemann algorithm is a routing problem on the two-dimensional mesh: each cell contains (at most) one packet whose destination is some arbitrary cell, and all packets need to be carried to their destinations using local

[2]Conversely, occasional routing failure is less consequential for the mesh-based relation collection step of [5].

[3]We use the conservative "large matrix" parameter set of [8]. Subsequent work by Lenstra et al. [9] provides refined and revised estimates for the NFS parameters needed for 1024-bit composites; we maintain use of the original parameter set for ease of comparison.

[4]Note that these chip sizes are significantly below the chip sizes suggested in [11] for the sieving step for 1024-bit numbers.

operations. The design goal is to minimize the total routing time as well as the size of the circuit.

The routing algorithm proposed in [8] is *clockwise transposition routing*, defined as follows. At every step, each cell holds at most one packet, and the only operation allowed is a conditional exchange ("compare-exchange"). The compare-exchange rule is defined as follows. Consider adjacent cells at columns j, $j+1$ which hold packets destined to columns j_0, j_1 respectively (either of which may be NIL if the corresponding cell is empty). The exchange is performed if one of the following holds:

- Single packet: $j_1 = \text{NIL}$ and $j_0 > j$, or $j_0 = \text{NIL}$ and $j_1 < j+1$.
- Farthest-first along compared direction: $j_0, j_1 \neq \text{NIL}$ and $j_0 \geq j_1$.

The analogous rule holds for packets destined to vertically adjacent cells. When a packet reaches its destination, it is consumed (removed from the mesh). For the linear algebra step, we use an additional rule justified by the modulo 2 arithmetics: whenever two identical packets are compared, both are annihilated.[5] The schedule of the compare-exchange operations is as follows:

- In step $t \equiv 0 \pmod 4$, for every column j and odd row i, compare the cell at (i, j) to the cell at $(i-1, j)$, i.e., above it.
- In step $t \equiv 1 \pmod 4$, for every row i and odd column j, compare the cell at (i, j) to the cell at $(i, j+1)$, i.e., to its right.
- In step $t \equiv 2 \pmod 4$, for every column j and odd row i, compare the cell at (i, j) to the cell at $(i+1, j)$, i.e., below it.
- In step $t \equiv 3 \pmod 4$, for every row i and odd column j, compare the cell at (i, j) to the cell at $(i, j-1)$, i.e., to its left.

Note that each cell performs compare-exchange with its 4 neighbors in cyclic clockwise or anticlockwise order, where the direction and phase depend on its location.

While no theoretical analysis of this algorithm's performance is known, experimentally its average-case performance appears close to optimal: for randomly filled $m \times m$ meshes, and m sufficiently large, the running time of the algorithm seems to be close to the optimum $2m-2$ with overwhelming probability. Combined with the very simple control circuitry, this renders clockwise transposition routing a very attractive choice for the NFS linear algebra step implementation.

3 Livelocks

3.1 Existence of livelocks

Unfortunately, it turns out that the clockwise transposition routing algorithm may enter a livelock (infinite loop) and not terminate. The simplest example is "rotation" of the 4×4

[5] For $K > 1$, each packet contains a vector in $\mathrm{GF}(2)^K$, and the packets are merged (i.e., replaced by one packet containing their XOR) instead of annihilated.

mesh, defined as follows (the matrix entries give the row and column indices of the packet's target):

$$\begin{array}{cccc} (3,0) & (2,0) & (1,0) & (0,0) \\ (3,1) & (2,1) & (1,1) & (0,1) \\ (3,2) & (2,2) & (1,2) & (0,2) \\ (3,3) & (2,3) & (1,3) & (0,3) \end{array}$$

On this input, the compare-exchange rule maintains the following invariant: in each row all packets contain the same destination column, and vice versa. Hence the exchange will always be performed. We can thus easily track each packet and verify that it never reaches its destination. Indeed, the original state is restored after 16 steps, without any packet having reached its destination.

In general, define $m \times m$ "rotation" as the routing problem in which the packet initially at (i,j) is destined to $(m-1-j,i)$ for all $0 \le i,j < m$. Then for m which is a multiple of 4, clockwise transposition routing enters a livelock loop of period $4m$.

The failure of these livelocks to occur in past experiments may be intuitively explained as follows. The livelock configurations are unstable, in the sense that when they are embedded in a larger mesh, interaction with packets "passing by" is likely to break the symmetries. Moreover, the observed livelock configurations are not translation-invariant — the constituent packets and their destinations must be very close to begin with. Thus, we may expect that in a large mesh there is a low probability that a local livelock configuration will occur and survive.

We remark that in addition to livelocks, there are pathological inputs that do terminate but take significantly more than $2m$ steps; one example is the "matrix transpose" input, where for every i and j, the packet initially at (i,j) is destined to (j,i). Characterization of the pathologies and analysis for random inputs, however, remain open problems.

3.2 Addressing livelocks

If a livelock (or some other pathology) occurs during any of the numerous routing operations performed during the linear algebra step, the whole computation will fail and much effort will have been wasted. As mentioned in Section 2.3, the experimental behavior of the clockwise transposition routing algorithm suggests a heuristic upper bound close to $2m$ on the expected running time of the algorithm. The mesh-based sieving device of [5] relies on this and simply aborts the routing after some allotted time. However, in contrast to the sieving step, for the LA step it is unacceptable to abort (and never complete) a valid computation.

In the absence of a full theoretical analysis of the algorithm and its pathologies, we are not aware of a conceptually satisfying solution. However, below we describe a heuristic solution that dodges the above problem without significantly increasing the complexity of the circuit hardware. Namely, we describe methods for detecting and resolving pathologies.

3.3 Detecting pathologies

The successful completion of the routing process is characterized by the fact that no cell in the mesh holds an undelivered packet. Let a cell at at position (i,j) output an $\textsc{empty}_{i,j}$

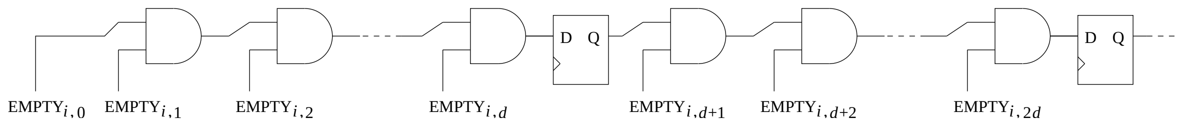

Figure 1: Pipeline for computing the disjunction of all $\text{EMPTY}_{i,\cdot}$ signals on the i-th row.

signal, which is 1 iff this cell does not contain any packet that is still pending (i.e., awaiting injection into the mesh) or in transition (i.e., undergoing clockwise transposition routing). The goal is then to ascertain when the $\text{EMPTY}_{i,j}$ signals are 1 for all $0 \leq i, j < m$.

This can be implemented as follows. First, we compute the AND of all $\text{EMPTY}_{i,j}$ signals along each row i, by means of a chain of binary AND gates that spans the whole row. Since the propagation delay along each row is huge, to ensure a stable and correct output we pipeline each chain by adding a flip-flop gate every d AND gates, where d is the depth of each pipeline stage ($d \gg 1$, but is limited by the technology employed and the clock rate). This is depicted in Figure 1. The outputs of the m chains are again ANDed together, using one additional chain (or alternatively, a more expensive but low-latency m-input AND tree). The worst-case latency from each cell to the final output is $2m/d$, so whenever some $\text{EMPTY}_{i,j}$ is unasserted, the global output will be 0 at least once in the subsequent $2m/d$ clock cycles. Thus, we know the routing is complete whenever the final output is 1 for $2m/d$ consecutive steps (this is trivially tested). If this does not occur within some allocated time $t_{\max}$, we consider the current routing configuration pathological. A reasonable time bound, assuming the refill from Section 4.3 is not used, is $t_{\max} = 2.1m + 2m/d$.

3.4 Completing the computation

Once the device has ascertained that a given routing operation fails to complete in the allocated time, it must recover from this situation. Since we expect this to occur very rarely (with realistic parameter choices in our experiments we never encountered a livelock), the solution can be relatively inefficient in terms of recovery time, but should not significantly increase the circuit size.

The simplest recovery method is to pause all computation, off-load the problematic routing problem to an auxiliary general-purpose computer for computation using some slower (on average) but well-analyzed routing algorithm, and re-load the state from the auxiliary computer.

A more elegant solution is to temporarily change the function and schedule of the compare-exchange elements: the cells are logically considered to lie along one long serpentine ring embedded in the mesh; the exchanges along this ring are always performed, among even pairs on even steps and among odd pairs on odd steps (see Figure 2). This way, every packet travels along the ring at a constant pace until it reaches its destination. After at most m^2 steps, all packets are consumed; for our choice of parameters, m^2 is a negligible fraction of the total running time of the LA step.

A third approach is based on the observation that all known livelocks are very fragile (cf. Section 3.1): permuting the packets locally in some random way, or disturbing the

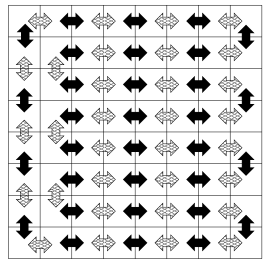

Figure 2: Reusing the compare-exchange elements for m^2-step guaranteed routing. Black (resp., dashed) arrows indicate exchanges done on even (resp., odd) clock cycles.

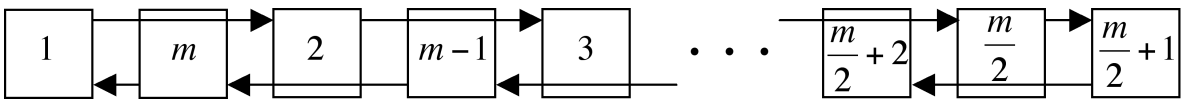

Figure 3: Realizing a torus in a flat array.

compare-exchange schedule for a few cycles, appears likely to break the pathological configuration. If a small amount of entropy suffices, this may be the most efficient solution. However, theoretical analysis seems non-trivial.

4 Efficiency improvements

4.1 Torus topology

One design improvement, proposed earlier in the context of the sieving step [5], is based on replacing the mesh topology with a torus: the two ends of each row (and column) are connected. This modification halves the maximum source-to-target distance, and thus we may hope that it halves the running time. Empirically, this is indeed the case (cf. Section 5).[6] An additional benefit is discussed in Section 4.3.

We cannot naively realize the torus as a mesh augmented by wrap-around wires; these wires will be very long (they span m cells), and thus will have extremely high signal propagation time and necessitate a low clocking rate of the chip. We use the standard technique of interleaving, shown (for a single row) in Figure 3. This amortizes the length of the wrap-around connections across the cells. The interleaving is applied to both rows and columns, to obtain equal distances between all (logically) neighboring mesh cells.

Next, the routing algorithm needs to be adapted to the torus. Only the compare-exchange rule needs to be modified. We wish to maintain the "farthest-first along the compared direction" intuition, but difficulties arise with respect to packets crossing the $m/2$ distance-to-target threshold. For simplicity and ease of implementation, we choose the following variant. For each packet we compute its shortest horizontal and vertical distance-to-target. If the shortest path crosses column $m/2$, then we add $m/2$ to the target column address; if the

[6] This holds even though the *expected* distance for random inputs is reduced only by a factor of 4/3.

shortest path crosses column 0 then we subtract $m/2$ from the target column address. Thus, the range of possible column addresses is between $-m/2$ and $m+m/2-1$. We normalize by adding $m/2$ to all column addresses, to obtain numbers between 0 and $2m$. The same is done for row addresses. All of this is done once, when loading the matrix entries into the mesh at the beginning of the multiplication chain.

With this representation, each packet has a dedicated "preferred direction" along each dimension, and will not change this direction even if the distance in the opposite direction would occasionally become the shorter choice (due to deflection by other packets). The exchange rule is the same as in the mesh, with row/column addresses that are 1 bit longer. The above representation assigns two addresses, which differ by m, to each cell. If m is a power of two then the difference is only in the most significant bit, so detecting that a packet has reached its destination entails the same simple comparison as in the mesh.

4.2 Parallel tori

Another idea from [5] which can be adapted to our case is to speed up the routing by an additional factor of ≈ 2 by using four interleaved but independent sub-tori of size $m/2 \times m/2$, instead of one $m \times m$ torus. One sub-torus consists of the cells at even rows and even columns; another consists of the cells at even rows and odd columns, and so on. In each sub-torus, the expected source-to-destination distance is $m/4$ as opposed to $m/2$ in the single torus (and the maximum distance is similarly halved).

Taking into account the interleaving already done in order to realize the torus topology, we see that each edge now physically spans a distance of 3 cells. In a concrete chip implementation, this would require additional metal interconnect layers. While the entailed cost depends on the specific manufacturing technology, it appears likely to be outweighed by the factor-of-2 improvement in throughput. However, further parallelization (e.g., use of 16 interleaved tori) appears beyond the capabilities of current technology.

Since the sub-tori are independent, packets cannot be injected into arbitrary cells anymore; they must be injected into a cell belonging to the same sub-torus as the packet's destination. To handle this we partition the physical mesh into blocks of 2×2 cells, so that each block contains one cell from each of the 4 sub-tori. The packets are then re-distributed among the cells of each block according to the above criterion. Here the situation is advantageous compared to that of the sieving problem considered by [5]: since the packet destinations are known in advance (cf. Section2.1), the re-distribution can be computed in advance, and only the packet contents (i.e., coordinates of the input vector of the matrix-by-vector multiplication) need to be communicated among the cells of the group.

4.3 Refilling

For the relevant parameter choices (and specifically for large ρ as defined in Section 2.1), each cell has several packets that have to be routed. Previous works [8, 4] handled this by iterating the routing algorithm several times. At each iteration a set of packets is injected into the mesh and all of them are routed to their destination; only then the next set is injected.

	mesh			torus		
	no refill	with refill		no refill	with refill	
Mesh size	steps	w	cycles	cycles	w	cycles
128×128	33 840	4	43 228	17 888	2	17 320
256×256	67 698	7	95 610	34 532	2	31 124
512×512				69 188	3	57 876

Table 2: Number of steps for routing 128 (randomly chosen) packets per cell.

To improve performance, we may try to inject packets before the mesh has been fully cleared. The question arises when to release a new packet into the underlying rectangular mesh: in principle a new packet can be released into a cell whenever the cell does not contain any packet in transit; however, releasing new packets too fast can result in a congestion of the mesh and after all be slower than the original scheme. We choose the following heuristic. Let w be some small positive number. Whenever a cell has not held a packet in transition for the last w clock cycles, it injects one of its pending packets (if any) into the mesh. This is repeated, until there are no packets left, either pending or in transit. Experimental results are reported in Section 5.

In the context of refilling, the torus topology has an interesting advantage. With the plain mesh, many packets attempts to reach their target by traveling through the central area of the mesh. Experimentally, we observe that when refilling is used the central area of the mesh quickly becomes very congested, and forms a routing bottleneck. This problem is completely mitigated by the torus topology (for random inputs), since there is no longer any preferred region. Experimentally, the behavior is indeed more homogeneous and the total routing time is lower.

5 Experimental performance

We simulated the above routing algorithms on meshes of size 128×128 and 256×256, which suffice for the linear algebra step in the factorization of 1024-bit composites using the distributed design of [4] (cf. Table 1). The results are given in Table 2. In these simulation, each cell in the mesh was initially loaded with 128 packets having random target addresses. We then tested four variants: mesh vs. torus (Section 4.1), and with vs. without refill (Section 4.3). For the refill variant, we tried several values of the parameter w; the results shown are for the experimentally optimal values of w. To estimate the effect of the parallel tori of Section 4.2, simply halve the value of m and look at the corresponding row. We performed several runs for each set of parameters, and the table gives the average values for the number of steps until termination. These values can be regarded as very stable; the maximum and minimum values we observed differ by less than 1%.

For the mesh topology, refilling reduces performance (cf. Section 4.3). If the number of "wait cycles" w is reduced (e.g. to 2), then the congestion at the center of the mesh gets even worse, and the running time becomes about twice larger than without refill. Moreover the variance of the running time is much higher (up to 20 %), and the simulation ran into a livelock in one of our examples. Thus, for the mesh it is better to completely route a set of packets before injecting the next set.

Routing with torus is faster by approximately a factor 2, as expected. Refilling the mesh during routing seems to be the better choice here, especially for larger meshes. For a mesh with 128×128 cells the advantage is relatively small (3 %), for a mesh with 512×512 cells it increases to 17 %.

The number of steps in Table 2 might be slightly reduced if packet annihilation or merging are used (cf. Section 2.3 and [8]). In the mesh topology, packet annihilation or merging cause a significant speed up if the mesh is used with refill, but the running time is still longer than without refill (more than 75 000 clock cycles compared to $\approx$ 67 700 clock cycles). In the cases of interest (torus with refill and mesh without refill), the improvement obtained through packet annihilation or merging is less than one percent.

The speed-up of a factor ≈ 4 compared to the original routing circuit [8] does not imply the same speed-up for the whole NFS linear algebra step, since the the distributed design [4] spends some time on inter-chip I/O. However, for a single-wafer design like [8], the factor ≈ 4 improvement indeed carries over to the total running time of the LA step.

6 Conclusion

We have shown that clockwise transposition routing, as originally proposed and used in the custom-hardware designs [4, 8] for the linear algebra step in the Number Field Sieve, may fail to terminate. To avoid a failure of the linear algebra step in this or other pathological cases, we described an efficient modification of the proposed circuits which ensures termination of the routing algorithm in all cases. Furthermore, we have suggested improvements to the circuit which improve the routing efficiency by a factor of ≈ 4.

Acknowledgements. We thank Adi Shamir for insightful discussions. The "matrix transpose" example in Section 3.1 was suggested by Uriel Feige.

References

[1] D. J. Bernstein. Circuits for Integer Factorization: a Proposal. Available at `http://cr.yp.to/papers/nfscircuit.pdf`, 2001.

[2] D. Coppersmith. Solving Homogeneous Linear Equations over GF(2) via Block Wiedemann Algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.

[3] W. Geiselmann and R. Steinwandt. A Dedicated Sieving Hardware. In *Public Key Cryptography — PKC 2003*, volume 2567 of *LNCS*, pages 254–266. Springer, 2003.

[4] W. Geiselmann and R. Steinwandt. Hardware for Solving Sparse Systems of Linear Equations over GF(2). In *Cryptographic Hardware and Embedded Systems — CHES 2003*, volume 2779 of *LNCS*, pages 51–61. Springer, 2003.

[5] W. Geiselmann and R. Steinwandt. Yet Another Sieving Device. In *Topics in Cryptology — CT-RSA 2004*, volume 2964 of *LNCS*, pages 278–291. Springer, 2004.

[6] A. K. Lenstra and H. W. L. Jr., editors. *The development of the number field sieve*, volume 1554 of *LNCS*. Springer, 1993.

[7] A. K. Lenstra and A. Shamir. Analysis and Optimization of the TWINKLE Factoring Device. In *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 35–52. Springer, 2000.

[8] A. K. Lenstra, A. Shamir, J. Tomlinson, and E. Tromer. Analysis of Bernstein's Factorization Circuit. In *Advances in Cryptology — ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 1–26. Springer, 2002.

[9] A. K. Lenstra, E. Tromer, A. Shamir, W. Kortsmit, B. Dodson, J. Hughes, and P. C. Leyland. Factoring Estimates for a 1024-Bit RSA Modulus. In *Advances in Cryptology — ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 55–74. Springer, 2003.

[10] A. Shamir. Factoring Large Numbers with the TWINKLE Device. In *Cryptographic Hardware and Embedded Systems. First International Workshop, CHES'99*, volume 1717 of *LNCS*, pages 2–12. Springer, 1999.

[11] A. Shamir and E. Tromer. Factoring Large Numbers with the TWIRL Device. In *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *LNCS*, pages 1–26. Springer, 2003.

[12] D. H. Wiedemann. Solving Sparse Linear Equations Over Finite Fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986.