On the reduction of composed relations from the number field sieve
(Extended Abstract)

Thomas F. Denny*
Universität des Saarlandes
FB 14 Informatik
Postfach 15 11 50
66041 Saarbrücken
Germany

Volker Müller*
Department of C & O
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

4th December 1995

Abstract

In this paper we will present an algorithm which reduces the weight (the number of non zero elements) of the matrices that arise from the *number field sieve* (*NFS*) for factoring integers [9] and computing discrete logarithm in $\mathbb{F}_p$, where p is a prime ([3], [13]). In the so called *Quadruple Large Prime Variation* of *NFS* a graph algorithm computes sets of *partial relations* (relations with up to 4 large primes) that can each be combined to ordinary relations. The cardinality of these sets is not as low as possible due to time and place requirements. The algorithm presented in this paper reduces the cardinality of these sets up to 30 %. The resulting system of linear equations is therefore more sparse as before, which leads to significant improvements in the running time of the linear algebra step (with either the Lanczos algorithm ([7], [11]) or structured Gaussian elimination [6]). Compared with the total time that is needed to solve the systems (above all in $\mathbb{F}_p$), the time needed by the presented algorithm can be ignored.

1 Introduction

Two of the most important problems in number theory and cryptography are the factorization problem for integers and the problem of computing discrete logarithms (DL) in a finite prime field. These two problems can both be solved with the so called *number field sieve algorithm (NFS)* ([9], [3], [13]). Factoring and solving DL with *NFS* requires solving a large sparse system of linear equations. This can be done with either the Lanczos algorithm ([7], [11]) or structured Gaussian elimination [6]. The complexity of the Lanczos algorithm depends on the dimension and the weight of the system. In contrary to ordinary Gaussian elimination which depends only on the dimension of the system, the reduction effect that can be achieved with structured Gaussian elimination depends strongly on the weight of the system. Therefore we want to generate systems with small weight and small dimension. Recently, a new variation, the so called *Quadruple Large Prime Variation* of the *NFS*, has gained a lot of attention [2]. Both

*Email: denny@cs.uni-sb.de , vmueller@crypto2.uwaterloo.ca

methods that were described to handle the *partial relations* lead to a growth in either the dimension ([2], [5]) or the weight of the system ([8], [14]). Due to place and time requirements we prefer (at least for DL) the second method. In this variation a graph algorithm is used to combine *partial relations* to ordinary relations. This is done by reducing the problem of combining the *partial relations* to the problem of finding a set of fundamental cycles in a graph. Each fundamental cycle of the graph leads to one ordinary relation. The algorithm does not find the shortest fundamental cycles because of the enormous time and place requirements of such an algorithm (see [4], [8]). Practical experiences (mainly with *NFS* for factoring integers) show that up to 200 *partial relations* are involved in composing one ordinary relation. This leads to much heavier systems of linear equations than in the original *NFS* algorithm. These heavier systems are more difficult to solve either with the Lanczos algorithm or the structured Gaussian elimination.

The algorithm described in this paper reduces the length of the fundamental cycles by doing transformations on their basis. These transformations can lead to a set of fundamental cycles where up to 30 % less *partial relations* are involved. Therefore the resulting matrix for the final linear algebra step of the *NFS* is probably of much lower weight as the matrix produced by the standard graph algorithm, and it can be solved significantly faster. Moreover, the running time of our transformation procedure can be ignored compared to the time of the algorithms for solving the linear system.

2 Notations

We will not describe the graph algorithm that produces the set of fundamental cycles. For a detailed description we refer to ([14], [8]). In the following we assume that a set of fundamental cycles has been computed. A cycle is further on viewed as the set of *partial relations* that it consists off. The set of all cycles will be called $\mathcal{C}$ and the set of all *partial relations* that occur in at least one cycle will be called $\mathcal{E}$. The *partial relations* will be marked by the small letters a, b, c, d and e. If it is not necessary to describe each *partial relation* of a cycle, r will be used for a path of at least two edges.

We will number the set of cycles consecutively. The set of *partial relations* that form a cycle i is called $edges(i)$. By the length $len(i)$ of a cycle i we denote the number of *partial relations* that the cycle i consists of. As a short notation for $edges(i) = \{a, b, c\}$ we will use $i = abc$. In this example $len(i) := |edges(i)| = 3$.
We define the following maps

$$\begin{array}{rccl} cyc: & \mathcal{E} & \rightarrow & \mathcal{C} \\ & a & \mapsto & \{i | a \in edges(i)\} \end{array}$$

as the algorithm needs to know in which cycles each *partial relation* occurs, and

$$\begin{array}{rccl} h: & \mathcal{E} & \rightarrow & \mathbb{N} \\ & a & \mapsto & |cyc(a)| \end{array}$$

in order to know in how many cycles each *partial relation* occurs. A *partial relation* a that only occurs in one cycle ($h(a) = 1$) will be called *solitary relation*; otherwise we will talk about a *multiple relation*. It is important to know how many *solitary relations* occur in each cycle.

Therefore we define

$$\begin{array}{rccl} one\_entry: & \mathcal{C} & \rightarrow & \mathbb{N} \\ & i & \mapsto & \Big|\{a|a \in edges(i) \wedge h(a) = 1\}\Big| \ . \end{array}$$

The number of *multiple relations* that occur in a cycle can easily be computed as the difference of the length of a cycle and its number of *solitary relations*.

We define the following functions on two cycles (better on the sets of their *partial relations*)

$$\begin{array}{rccl} common: & \mathcal{C} \times \mathcal{C} & \rightarrow & \mathcal{E} \\ & (i,j) & \mapsto & edges(i) \cap edges(j) \end{array}$$

which will be called the *common part* of cycle i and j,

$$\begin{array}{rccl} \bigoplus: & \mathcal{C} \times \mathcal{C} & \rightarrow & \mathcal{E} \\ & (i,j) & \mapsto & (edges(i) \cup edges(j)) - common(i,j) \end{array}$$

which will be called the *combination* of cycle i and j and

$$\begin{array}{rccl} rem: & \mathcal{C} \times \mathcal{C} & \rightarrow & \mathcal{E} \\ & (i,j) & \mapsto & edges(i) - common(i,j) \end{array}$$

which will be called the *remaining part* of cycle i (when it is combined with cycle j). The following observations for cycles i and j are easy to verify :

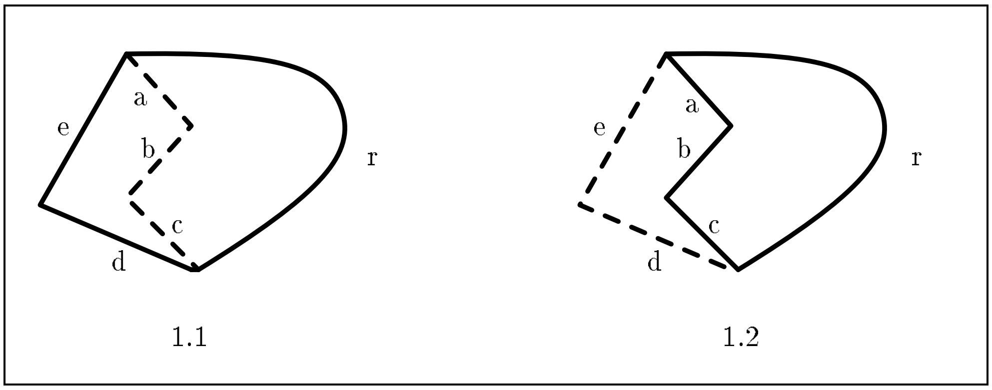
$$\begin{array}{rcl} common(i,j) & = & common(j,i) \ , \\ edges(i) & = & common(i,j) \cup rem(i,j) \quad \text{and} \\ |common(i,j)| & > & |rem(i,j)| \quad \Longleftrightarrow \quad 2 \cdot |common(i,j)| > len(i) \ . \end{array}$$

3 Idea of the algorithm

Practical experiences made with the standard graph algorithm lead to the following observation which is crucial for our algorithm: the sets of edges of two fundamental cycles computed by the standard graph algorithm have very often a large intersection. We will show in the following two subsections how to make use of this fact.

3.1 Reducing the number of partial relations in a cycle

Let us first consider the case where there exist two cycles which have more than half of the *partial relations* of one of them in common. We will illustrate the situation with the help of figure 1.1. Assume that we know the two cycles $i = abcde$ and $j = abcr$, where r is a path of relations of length at least 2. We have marked the common part of i and j by dashed lines. Then we can exchange in cycle j the edges abc by ed without destroying the cycle property. Figure 1.2 shows the situation after this replacement ($i = abcde$ and $j = der$). In this example the length of cycle j has decreased by 1.



This observation leads to the following reduction procedure: Assume that we find a pair (i, j) of cycles, where w.l.o.g. $len(i) \leq len(j)$ and $|common(i,j)| > |rem(i,j)|$. Then the original cycle j is replaced by the combination of cycle i and j ($j = i \oplus j$). This operation does not change the characteristic of cycles that their *partial relations* can be combined to an ordinary relation. The replacement $j = i \oplus j$ has the effect that in $edges(j)$ the *partial relations* that the cycles had in common ($common(i,j)$) are replaced by the *partial relations* that only occur in cycle i ($rem(i,j)$). Therefore the length of cycle j has decreased due to the assumption that the common part of cycle i and j ($common(i,j)$) is bigger than their remaining part ($rem(i,j)$). We will say *cycle i reduces cycle j*.

3.2 Replacing partial relations

In the last section we were interested in reducing the length of a cycle. Sometimes we find the situation, where exactly half of the edges of cycle i occur in the common part of cycle i and j (w.l.o.g. $len(i) \leq len(j)$). In this case we cannot reduce the length of one of the cycles, but nevertheless it is useful to replace cycle j by $i \oplus j$ if the probability for the "new" cycle ($i \oplus j$) to be reduced in a following iteration is higher than for cycle j. There are several ways to measure this probability. We decided to use the following function

$$\begin{array}{rccl} prob: & \mathcal{C} & \rightarrow & \mathbb{N} \\ & i & \mapsto & \displaystyle\sum_{e \in edges(i)} h(e)\ . \end{array}$$

Then $prob(i)$ sums up for all *partial relations* e of cycle i the number of cycles in which e occurs. This choice uses the heuristic that a cycle whose edges are members of many other cycles is more likely to be reduced in a further iteration. In order to decide whether to replace cycle j by $i \oplus j$ or not we compare $prob(j)$ and $prob(i \oplus j)$. To compute

$$diff := prob(j) - prob(i \oplus j) = \sum_{e \in edges(j)} h(e) \quad - \quad \sum_{e \in edges(i \oplus j)} h(e)$$

we use the equations

$$\begin{aligned} edges(j) &= common(j,i) \cup rem(j,i) = common(i,j) \cup rem(j,i) \quad \text{and} && (1) \\ edges(i \oplus j) &= (edges(i) \cup edges(j)) - common(i,j) \\ &= (common(i,j) \cup rem(i,j) \cup common(i,j) \cup rem(j,i)) - common(i,j) \\ &= rem(i,j) \cup rem(j,i) && (2) \end{aligned}$$

and compute

$$diff = \sum_{e \in common(i,j)} h(e) \quad - \quad \sum_{e \in rem(i,j)} h(e) \; .$$

If $diff < 0$, then cycle j is replaced by $i \oplus j$. We will say *cycle i changes cycle j.*

3.3 Inactive cycles

In order to reduce the running time of the algorithm it is useful to recognize cycles that cannot be reduced any more. We will call these cycles *inactive* and delete them from $\mathcal{C}$. This operation changes the values of *cyc* and h for all *partial relations* that occur in inactive cycles. Removing cycles that cannot be reduced any more could lead to more *solitary relations* in other cycles and could help to take better decisions in the changing step since *multiple relations* of inactive cycles are no longer taken into account in the probability computation (see section 3.2).

In general it is not easy to decide whether a cycle will be reduced any more or not. Nevertheless, the following observations help us to find some of the inactive cycles and to classify all the cycles:

- A cycle that only consists of *solitary relations* is inactive.
- A cycles that has more *solitary relations* than *multiple relations* will not change nor reduce other cycles.
- A cycles that only has a single *multiple relation* cannot be reduced nor reduce another cycles. It may be possible that this cycle can be changed by other cycles or change other cycles, but this changing would not have a big effect on other cycles.

Using these facts, we compute a status for each cycle which indicates whether it is inactive (0), can only be reduced or changed (1), or can also reduce or change another cycle (2) (keep in mind that cycles of status 2 can also be reduced or changed). At the beginning, the status of a cycle is computed as

$$status(i) = \begin{cases} 0 & \text{if } len(i) - one\_entry(i) \leq 1 \; , \\ 2 & \text{if } len(i) - one\_entry(i) > 1 \;\wedge\; 2 \cdot one\_entry(i) \leq len(i) \; , \\ 1 & \text{otherwise} \; . \end{cases}$$

The status of a cycle depends only on its length and number of *solitary relations.* Therefore the status of a cycle must only be recomputed if it is involved in a reduction or changing step.

4 The algorithm

All algorithms described in this section will be given in C^{++}-pseudo code in the final version of this paper. In this extended abstract, we only describe the basic ideas of the different parts of the algorithm.

4.1 Precomputation

In the first step the algorithm collects information about the structure of the cycles. It reads the cycles and counts in how many cycles each *partial relation* occurs. Then it determines the number of *solitary relations* that occur in each cycle (*one_entry*). This information is needed to compute the status of each cycle. In order to efficiently test whether a cycle reduces or changes other cycles, the precomputation step also computes for each *multiple relation* a the set of cycles $cyc(a)$ in which it occurs.

4.2 Testing the reduction and changing conditions

After the precomputation step the algorithm searches for cycles that reduce or change other cycles until there are no more status 2 cycles left. Since we know the length of each cycle and the equations

$$len(i) = |common(i,j)| + |rem(i,j)|\,, \quad \text{for all cycles } i \text{ and } j \text{ and}$$

$$len(i) - 2 \cdot |common(i,j)| \leq 0 \iff |common(i,j)| \geq |rem(i,j))|\,,$$

we only need to compute the common part of cycle i and j (or better its cardinality) to decide whether cycle i reduces or changes cycle j (or vice versa). Depending on the situation, our algorithm uses two different ways to determine the common part of cycles. Given two cycles i, j, the first version computes the number of relations that occur in both cycles. It uses their sets of *partial relations* (which are implemented as sorted arrays of relations) to determine the cardinality of their common part.

In general, we are interested in all cycles j that could be reduced or changed by a cycle i. Therefore the second version uses the sets $cyc(a)$ that have been computed in the precomputation step to count how many *partial relations* each cycle has in common with a fixed cycle i. It is sufficient to examine the *multiple relations* of cycle i since *solitary relations* cannot be in any common part. For each cycle a counter is initialized to 0. For each *multiple relation a* of cycle i, the counter for cycles that occur in the set $cyc(a)$ is increased. After this procedure the counter for cycle j ($j \neq i$) contains the number of *partial relations* that occur in cycle i and cycle j. This information is used to decide whether cycle j is replace by $i \oplus j$ or not.

4.3 Different variations of the algorithm

Whenever a cycle is replaced, the number of *solitary relations* of this cycle is updated and the changes on the sets of *multiple relations* are done. If *multiple relations* become *solitary relations* or vice versa, the *one_entry* value (and maybe the status) of the corresponding cycle is updated. In order to avoid useless tests and to ensure the termination of the algorithm, it is necessary to change the status of cycles that have been tested. We extend the status definition by a status 3 for cycles that were tested. Whenever a cycle j is replaced by $i \oplus j$, it may be possible that a status 3 cycle (a cycle that has already been tested) can now reduce or change this new cycle. Therefore we examine under which conditions a cycle k reduces (or changes) cycle $i \oplus j$ and does not reduce (or change) cycle j. In this situation we have $2 \cdot |edges(k) \cap edges(i \oplus j)| \geq len(k)$ and $2 \cdot |edges(k) \cap edges(j)| < len(k)$, which implies

$$|edges(k) \cap edges(i \oplus j)| \quad > \quad |edges(k) \cap edges(j)|\,.$$

With (1), (2) we get

$$|edges(k) \cap (rem(i,j) \cup rem(j,i))| \quad > \quad |edges(k) \cap (common(j,i) \cup rem(j,i))|\,.$$

Since the intersections

$$\begin{aligned} rem(i,j) \cap rem(j,i) &= (edges(i) - common(i,j)) \cap (edges(j) - common(j,i))\,, \\ common(j,i) \cap rem(j,i) &= common(j,i) \cap (edges(j) - common(j,i)) \end{aligned}$$

are empty and

$$|edges(k) \cap (rem(i,j) \cup rem(j,i))| = |edges(k) \cap rem(i,j)| + |edges(k) \cap rem(j,i)|\,,$$

$$|edges(k) \cap (common(j,i) \cup rem(j,i))| = |edges(k) \cap common(j,i)| + |edges(k) \cap rem(j,i)| \ ,$$

it follows that in this situation we have

$$|edges(k) \cap rem(i,j)| \quad > \quad |edges(k) \cap common(j,i)| \quad \geq \quad 0 \ .$$

Therefore all status 3 cycles that have at least one *partial relation* in common with $rem(i,j)$ may reduce or change the new cycle. In the so called recursive variation of our algorithm we use the first version of the testing procedure described in section 4.2 to check whether such a cycle can reduce or change the new cycle. The running time of the algorithm with recursive variation is normally only slightly bigger than without it. But practical experiences show that it also normally leads to only slightly better results. We will show some examples in section 5.

The main problem of the algorithm is to find the order in which the cycles should be tested, i.e. the order that leads to the greatest reduction effect. In order to minimize recursive effects (and the running time) we decided to start with those cycles that probably lead to the smallest changes on the set of cycles. This quickly gives a first impression on the reduction that can be achieved and will be called variation I. You can get slightly better results by starting with those cycles that probably eliminate larger parts but the running time will almost double. This will be called variation II. The difference between *multiple* and *solitary relations* of a cycle is used to predict the possible changes on the set of cycles. If we want to minimize (maximize) the effects that a changing or reduction step with a cycle will probably have on the sets of cycles, this difference must be as small (big) as possible. A bound is used to decide whether a cycle with status 2 is tested or not. Depending on what variation we use this bound is at the beginning set to zero or to the maximum length of all cycles. All status 2 cycles are tested that have at most (least) bound *multiple relations* more than *solitary relations*. If there are no more status 2 cycles with the desired property left, the bound is increased (decreased) until there are no more status 2 cycles left.

5 Practical experiences

In this section we will give practical results achieved with our implementation of the new algorithm. We used this algorithm for reducing the weight of matrices over $\mathbb{F}_2$ (produced by the *NFS* for factoring integers) and $\mathbb{F}_p$ (*NFS* for computing discrete logarithms). The data were produced with Jörg Zayer's implementation of the *GNFS* factoring algorithm [14] and Damian Weber's implementation of the DL-*NFS* algorithm [13]. All timings were done on a Sparc 10 computer having 128 MB main memory.

5.1 Examples from factoring

The first table gives running times for the different strategies described in section 4.3. We show the data for a 30-digit, a 55-digit, a 65-digit and a 75-digit factorization, all done with the Quadruple Large Prime Variation of the *NFS* (for details see [14]). We list the variants of the algorithm (as explained in section 4.3), the number of cycles computed by the original graph algorithm, the sum of the lengths of all cycles, the number of *partial relations* that were eliminated and the running time.

Fact. / Variant	# cycles	sum of all cycle lengths	# eliminated relations	Running time
30d	4355	361402		
I			69896 (19.34 %)	1:19 min
II			132836 (36.76 %)	5:49 min
I rec.			95339 (26.38 %)	5:38 min
II rec.			136651 (37.81 %)	27:45 min
55d	59823	495493		
I			37973 (7.66 %)	14:46 min
II			38122 (7.69 %)	21:20 min
I rec.			38061 (7.68 %)	14:39 min
II rec.			38140 (7.69 %)	21:00 min
65d	24692	495296		
I			73948 (14.93 %)	3:04 min
II			75064 (15.16 %)	5:45 min
I rec.			74532 (15.05 %)	3:23 min
II rec.			75090 (15.16 %)	6:07 min
75d	55457	7004103		
I			1981756 (28.29 %)	38:45 min
II			2115652 (30.21 %)	2:59:22 h
I rec.			2058057 (29.38 %)	1:55:26 h
II rec.			2120191 (30.27 %)	9:22:56 h

The table shows the practical behavior we found in most of our examples: the recursive strategy leads to only slightly better results as the non-recursive version and needs similar time. Note that the 30-digit factorization is the only example we have found where the recursive version is significantly better than the non recursive one. On the other hand, the improvements of up to 30 % in these applications could be achieved in a short time compared to the time needed for solving the linear systems.

The next table lists the total number of different relations in all fundamental cycles ($|\mathcal{E}|$) and the number of multiple relations of it.

	30d	55d	65d	75d
# relations	14760	135928	99907	334863
# mult. relations	7011	52781	49676	186485
	47.5 %	38.8 %	49.7 %	55.7 %

One possible reason for the moderate improvements in the 55-digit factorization example might be the fact that the number of partial relations per cycle was relatively small (on average 8.28). In the other examples, this number was significantly higher (82.98, 20.05, 126.29, respectively). This argument is supported by another *NFS* computation for the 65-digit number, but using Double Large Prime instead of Quadruple Large Prime Variation. Then the average number of relations per cycle is only 2.61 and the improvements obtained with our algorithm are marginally. Within a few seconds the algorithm found out, that only 0.0693 % of the *partial relations* could be eliminated. Thus it seems that the ideas of this paper correlate directly to the Quadruple Large Prime Variation of *NFS*.

5.2 Examples from DL

Finally, we show some experimental data computed with the Quadruple Large Prime *NFS* variant for solving discrete logarithms. Here the situation is quite different. The average number of partial relations per cycle lies between 22.38 and 76.52, and the improvements by using the algorithm of this paper were almost 30 %. The first example is part of a DL computation modulo a 50-digit prime. The second example is from the actual record for DL computations (see [1]).

DL problem / Variant	# cycles	sum of all cycle lengths	# mult. relations	# eliminated entries	Running time
50d DL	16344	365842	23936		
I				102168 (27.93 %)	1:50 min
II				105355 (28.80 %)	3:56 min
I rec.				104116 (28.46 %)	2:22 min
II rec.				105645 (28.88 %)	4:33 min
65d DL	23759	102906	49053		
I				116083 (14.01 %)	5:43 min
II				117865 (14.22 %)	13:02 min
I rec.				117091 (14.13 %)	7:30 min
II rec.				118110 (14.25 %)	16:33 min

The following data are from current work on breaking K. McCurley's 129-digit challenge [10]. In the first variation all cycles with less than 80 *partial relations* were used as input for our algorithm and in the second one we only took those cycles with less than 150 *partial relations*.

DL problem / Variant	# cycles	# entries	# mult. relations	# eliminated entries	Running time
126d DL I	64383	277930	142255		
I				439351 (17.23 %)	22:36 min
II				443318 (17.38 %)	57:46 min
I rec.				441343 (17.31 %)	26:37 min
II rec.				443761 (17.40 %)	75:32 min
126d DL II	79286	346265	186172		
I				1581437 (26.06 %)	45:05 min
II				1624037 (26.76 %)	192:47 min
I rec.				1604705 (26.44 %)	81:39 min
II rec.				1627594 (26.82 %)	365:07 min

6 Conclusion

We conclude that the algorithm described in this paper can reduce the number of relations per cycle significantly. The running time needed for this reduction is small compared to the time needed for solving the linear systems (especially for the $\mathbb{F}_p$-case). Unfortunately, we do not know how "good" the output of our algorithm really is, i.e. we do not know the number of relations in an optimal basis of fundamental cycles. For finding the "best strategy", we want to examine more examples, in particular with the *NFS* algorithm for the DL problem. In the DL *NFS*-algorithm, the reduction algorithm of this paper plays an important role in the critical last step of combining relations and solving the resulting linear system over a large finite prime field.

Acknowledgment: We would like to thank Johannes Buchmann, Damian Weber and Jörg Zayer for the support given during this work was done.

References

[1] T. F. Denny, D. Weber, *A 65-digit prime DL computation,* Message published on the NumberTheory Net, Oct. 1995

[2] B. Dodson, A. K. Lenstra, *NFS with Four Large Primes: An Explosive Experiment*, Advances in Cryptology, CRYPTO '95, Lecture Notes in Computer Science, vol. 963 (1995), Springer Verlag, pp. 372 - 385

[3] D. M. Gordon, *Discrete Logarithms in GF(p) using the Number Field Sieve*, SIAM J. Discrete Math. 6 (1993), pp. 124 - 138

[4] J. D. Horton, *A polynomial-time algorithm to find the shortest cycle basis of a graph*, SIAM J. Comput. 16 (1987) , pp. 344 - 355

[5] R. M. Huizing,*An implementation of the number field sieve*, Technical report, CWI Report NM-R9511, July 1995

[6] B. A. LaMacchia, A. M. Odlyzko, *Solving large sparse systems over finite fields*, Advances in Cryptology, CRYPTO '90, Lecture Notes in Computer Science, vol. 537 (1991), Springer Verlag, pp. 109 - 133

[7] C. Lanczos, *An iterative method for the solution of the eigenvalue problem of linear differential and integral operators*, J. Res. Nat. Bur. Standards, Sec. B 45, pp. 255 - 282

[8] A. K. Lenstra, M. S. Manasse, *Factoring with two large primes*, Mathematics of Computation, 63 (1994), pp. 72 - 82

[9] A. K. Lenstra, H. W. Lenstra, *The development of the number field sieve*, Springer–Verlag, 1993

[10] K. McCurley, *The discrete logarithm problem, cryptology and computational number theory*, Proc. Symp. in Applied Mathematics, American Mathematical Society, 1990

[11] P. L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2)*, Advances in Cryptology, EUROCRYPT '95, Lecture Notes in Computer Science, vol. 921 (1995), Springer Verlag, pp. 106 - 120

[12] O. Schirokauer, *Discrete logarithms and local units*, Phil. Trans. R. Soc. Land. A 345 (1993), pp. 409 - 423

[13] D. Weber, *An Implementation of the General Number Field Sieve to Compute Discrete Logarithms mod p*, Advances in Cryptology, EUROCRYPT '95, Lecture Notes in Computer Science, vol. 921 (1995), Springer Verlag, pp. 95 - 105

[14] J. Zayer, *Faktorisieren mit dem Number Field Sieve*, PhD Thesis, Universität des Saarlandes, 1995