DESIGN CONCEPTS FOR MATRICES AND LATTICES IN LiDIA

Patrick Theobald
TH Darmstadt
FB 20 - Informatik
Lehrstuhl Prof. Buchmann
Alexanderstr. 10
D-64283 Darmstadt
Germany

theobald@cdc.informatik.th-darmstadt.de

Susanne Wetzel Werner Backes
Universität des Saarlandes
FB 14 - Informatik
Postfach 15 11 50
D-66041 Saarbrücken
Germany

{wetzel,wbackes}@cs.uni-sb.de

Keywords: Matrix Concept, Lattice Concept, Template Modules, Template Kernel

ABSTRACT

In this paper we present a new template class concept for matrices and lattices. In the design we have focussed on the efficiency of single as well as sequences of operations. In order to get concise and efficient programming code we present a new technique based on template mechanisms and suitable inheritance structures.

1 INTRODUCTION

In many fields in mathematics and computer science, computations on matrices and lattices have various applications, e.g. in class group theory, cryptography, combinatorial optimization and scientific computing. Most of those applications are extremely time consuming and therefore require efficient implementations of the underlying data-types and basic operations. In general, many applications do not call only one but a sequence of lattice and matrix operations.

Through the design and the implementation of a new object-oriented concept for matrix and lattice operations in LiDIA[1] [4] we take into account the efficiency of a single algorithm as well as a sequence of different operations. Other computer algebra systems are typically single-algorithm-oriented which stands in contrast to our approach.

In LiDIA we collect specific information in each step of an operation sequence which is subsequently used in the next step of the on-going computation. In order to achieve the maximal efficiency, this concept obviously requires the implementation of several variations of the same algorithm. Using conventional programming methods would result in a large amount of duplicated code. In this paper we present a new concept solving the problem by combining template mechanisms with suitable inheritance structures. Our implementation is very efficient (for single algorithms as well as operation sequences), flexible, extendible and results in concise programming code which is easy to maintain.

In the following we first describe the inheritance structure of the template matrix and lattice classes (section 2.1). Subsequently (section 3), we present a new programming technique using so-called template modules and kernels for implementing the classes efficiently. On the basis of running timings (section 4) of LiDIA in comparison with other computer algebra systems we show the efficiency of our new concept.

[1] LiDIA is a Library for Computational Number Theory which is being developed by the LiDIA Group at the Institute of Prof. Dr. J. Buchmann at the Universität des Saarlandes/TH Darmstadt

2 STRUCTURE OF THE MATRIX AND LATTICE CLASSES IN LiDIA

In this section we will describe the structure of the matrix and lattice classes of the C++ library LiDIA [4] and present the advantages of our design concept. The basic two concepts are the application of inheritance to template structures and the use of bit fields for coding structure information, storage information and other high level information. By means of the first technique we can support matrices and lattices over different types. In addition, we achieve a gradation of the functions of the classes depending on the operations of the used template type. The second concept allows us to ensure that fast single operations also imply the efficiency of applications calling a sequence of operations.

2.1 Inheritance Structure and Bit Fields

The inheritance structure of the template matrix and lattice classes in LiDIA 1.3 is shown in figure 1.

The basis of our matrix classes consists of the two classes **dense_matrix** and **sparse_matrix** which are invisible for the user. For efficiency reasons, these classes are no virtual base classes [2, 13]. In these classes we define the data structure for sparse and dense matrices (see figure 2) which is explained by some examples in figure 3. As shown in figures 2 and 3 it is possible to create a matrix with dense, sparse or even a mixed representation by combining the dense and sparse data structure elements.

Bit fields such as **storage_mode**, **structure_mode**, **info_mode** and **print_mode** which are defined in the class **sparse_matrix** store specific matrix information which is extensively used by the derived classes. The bit fields can

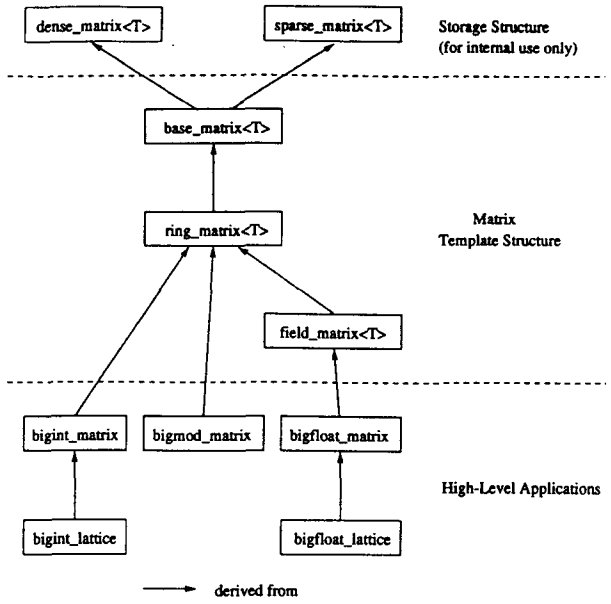

Figure 1: Inheritance Structure

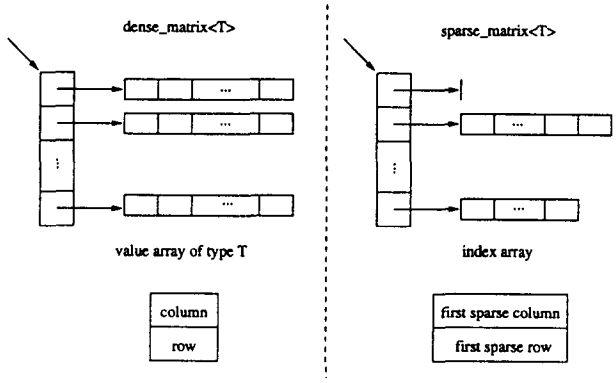

Figure 2: Dense/Sparse Data Structure

directly be set by the user or manipulated by the implemented algorithms.

- **storage_mode:** In this field we keep the information on the representation of the matrix (sparse, dense or mixed) as well as the information on whether the matrix is stored column or row-oriented. The advantage of this bit field is obvious since for each high level function the best representation can be chosen individually. This way we can ensure the most efficient computation for each function. Fast conversion routines build the connection between the different representations and orientations.

- **structure_mode:** The idea of this bit field is to use information about the distribution of the entries in the matrix in order to achieve a major speed-up. For example, we code the information whether a matrix has diagonal or triangular structure etc.

- **info_mode:** With this field we provide special information on the matrix e.g. whether the columns or rows

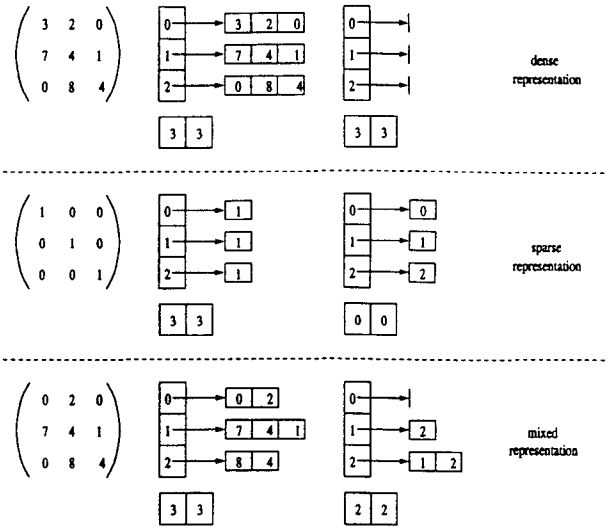

Figure 3: Example

are linearly independent or whether this is a special kind of matrix e.g. Gram matrix.

- **print_mode:** This bit field is the basis of all input and output operations of the matrix classes. The input/output format is chosen according to the bit field (e.g. in Maple[2]/PARI [3]/ Mathematica[3]/LiDIA format).

The class **base_matrix** is derived from the two classes **dense_matrix** and **sparse_matrix**. The idea of the class **base_matrix** is, that a matrix can be interpreted as a 2-dimensional array with a more advanced functionality. The requirements on the used template type are rather small. The template type only has to have an assign-operator, an input and output-operator as well as a swap function. In general, the class **base_matrix** provides access and structure functions. We refer to the LiDIA-Manual [11] for a more detailed description of the functionality.

In a second step the class **ring_matrix** is derived from the class **base_matrix**. This class gives the LiDIA user the possibility to create matrices over rings. Consequently, there are more restrictions to the used template type. In addition to the requirements of the class **base_matrix** this class demands the operators $+$, $-$ and $*$ as well as a member function called one() (returning the one element of the ring) and a member function called zero() (returning the zero element of the abelian group). At the moment this class only offers simple matrix operations over rings. For the future we plan to implement further high-level matrix functions operating over arbitrary rings.

By a third derivation step we obtain the class **field_matrix**. With this class it is possible to instantiate and work with matrices over fields. In addition to the requirements for the class **ring_matrix**, the template type has to have a / operator. At the moment this class also offers only simple matrix operations over fields.

Based on the described hierarchy it is now possible to derive the following high-level classes:

[2] Maple is a trademark of Waterloo Maple Software
[3] Mathematica is a trademark of Wolfram Research Inc.

- **bigint_matrix** – a class for doing linear algebra over $\mathbb{Z}$. This class offers functions for computing determinants, the Hermite normal form, the Smith normal form etc. [7].
- **bigfloat_matrix** – a class for doing matrix operations over **bigfloats** (in work).
- **bigmod_matrix** – a class providing modular matrix operations, e.g. computing the kernel modulo n [11].
- **bigint_lattice** and **bigfloat_lattice** – classes whose functionality comprises lattice algorithms for computing reduced lattice bases (e.g. Schnorr-Euchner algorithm [10, 12]), computing relations from a given generating system (e.g. Buchmann-Kessler algorithm [5]), handling Gram matrices as well as computing shortest and closest vectors [1, 7, 8].

3 TEMPLATE MODULES AND KERNELS

In the classes described in the previous section, many variations of the same basic algorithm have to be implemented depending on the various information stored in the bit fields. In general, the differences are rather small and the diversity would normally result in a large amount of programming code. One approach to solve that problem is applying the well-known C–technique of using function pointers [9]. The gained variability is tremendous but might imply a major loss of efficiency [14]. It seems that one only has the choice between a large and efficient or a small but slower code basis. Fortunately, there is a third way comprising the advantages of both techniques. In the following we will describe this new technique on the basis of the lattice classes **bigfloat_lattice** and **bigint_lattice** [11].

A lattice is an additive discrete subgroup of the $\mathbb{R}^n$ and can be determined by a basis, a generating system, a Gram matrix or a quadratic form [7]. Since matrices are a common representation for lattices, the lattice classes in LiDIA are derived from the matrix classes (see figure 1). In each of the classes **bigint_lattice** and **bigfloat_lattice** we have summarized the lattice algorithms for lattice bases, generating systems as well as Gram matrices. In earlier versions of LiDIA we had a class distinction between bases and generating systems. This approach was turned down because there was no real practical use for the class diversity.

As already mentioned before, the lattice classes in LiDIA offer algorithms for computing reduced lattice bases, computing relations from generating systems, handling Gram matrices etc. [5, 7, 10, 12]. For most algorithms the classes do not only contain an implementation of the original algorithm but also comprise several variations (e.g. using different scalar products). In order to keep the code basis small but very efficient and variable, we have developed a new concept which we will describe in the following subsections.

3.1 Template Module: Vector Operations

As an example, we will now focus on the Schnorr-Euchner algorithm [12] for motivating and explaining the various design decisions. The Schnorr-Euchner algorithm for computing a reduced lattice basis or relations from a generating system can be applied to **bigint** as well as **bigfloat** lattices. The necessary approximations within the algorithm can be done by using **doubles**, **xdoubles** (floating point arithmetic with double+double precision) or **bigfloats**. This diversity would normally imply the implementation and maintenance of six algorithms (later on referred to as versions), i.e. providing n variations of the original algorithms would result in the tremendous amount of $6 * n$ algorithms. The differences between those six algorithms are rather small and essentially result from vector operations over different types. Therefore we have implemented template vector classes as shown in figure 4.

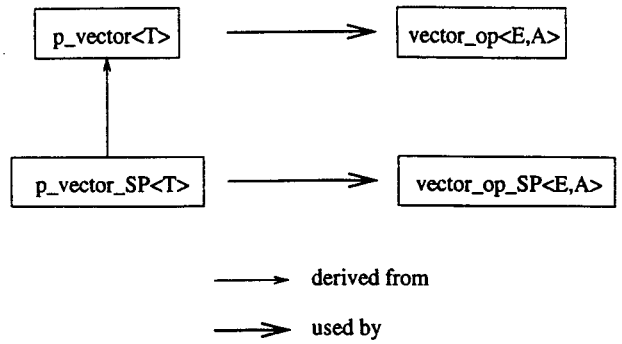

Figure 4: Template Module: Vector Operations

The class **p_vector<T>** (see figure 4) comprises vector operations working on pointers of the specified template type **T**. The class **p_vector_SP<T>** which is derived from the class **p_vector<T>** re-implements the scalar product by using a function pointer **void (*scalprod)(T&, T*, T*, lidia_size_t)**. Providing a function pointer the user can define a special scalar product which might be important for more elaborate applications. Since the use of function pointers implies a loss of efficiency, we are providing a separate class for this feature to restrict this disadvantage to the case of very specialized applications. The two classes **p_vector<T>** and **p_vector_SP<T>** are for internal use only. The classes are specialized for **bigints** and **bigfloats**. In addition, we are using those classes for the types **double** and **xdouble**.

Since the Schnorr-Euchner algorithm is working on the exact vector as well as its approximation at the same time, we simplify the handling by introducing the two classes **vector_op<E,A>** and **vector_op_SP<E,A>** (E=exact, A=approximation). The template structs **vector_op<E,A>** and **vector_op_SP<E,A>** consist of two elements **p_vector(E)**, **p_vector(A)** and **p_vector_SP(E)**, **p_vector_SP(A)** respectively.

3.2 Template Module: Type Conversions

Working on exact representations and the appropriate approximations at the same time causes major conversion problems and requires efficient conversion routines. The problems are solved by implementing the elementary conversion functions (from **E** to **A** and vice versa) in the class **base_modules<E,A,Var>** (the parameter **Var** is only used to delimitate the different variations of an algorithm). In the derived classes **basis/gensys_modules<E,A,Var>** we provide operations for converting vectors and lattices (see figure 5).

For lattices determined by a generating system, conversions require additional checks (e.g. for 0-vectors) and therefore have to be implemented in a separate class.

In principle we do now have all basic components, necessary for implementing the different versions and variations of an algorithm. The final cooperation is enabled by the kernel.

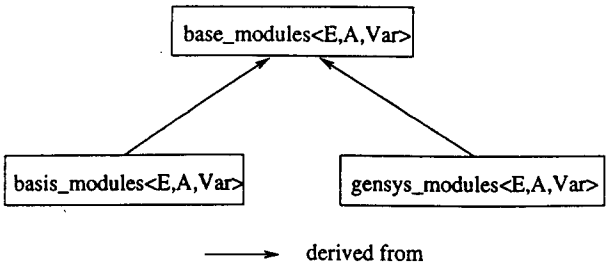

Figure 5: Template Module: Type Conversions

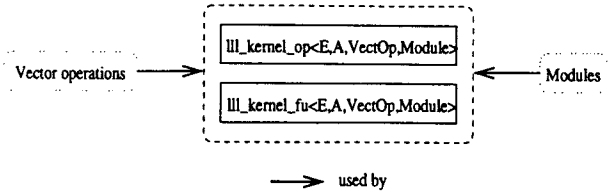

Figure 6: Template Kernel

3.3 Template Kernel

The kernel contains those parts of the various lattice algorithms which are the same for all versions (see figure 6).

Because of efficiency reasons, the kernel had to be implemented twice, based on operators for `doubles` and `xdoubles` called `lll_kernel_op` as well as on functions for `bigfloats` (`lll_kernel_fu`) since operators on data types which are not incorporated in the compiler cause an additional copy operation and are therefore less efficient.

The combination of the components will now be illustrated by the example of reducing a `bigint_lattice A`. With the call `A.lll(0.99)` we would like to reduce the lattice `A` by using the normal Schnorr-Euchner algorithm [12] with reduction parameter 0.99 and doing the approximations by means of `doubles`. The call `A.lll(0.99)` invokes an instantiation of the class `lll_kernel_op` as

```
lll_kernel_op<bigint,double,vector_op<bigint,double>,
    basis_modules<bigint,double,Normal> > alg;
```

followed by the operation `alg.lll(A,0.99)`. Depending on the compiler, the template construction gets resolved either at compile time of the library (e.g. GNU g++) or during compiling the application (e.g. AT&T CC). If the compiler can handle inlining well (e.g. GNU g++), the compiler will generate the full source code from the components template kernel and the template modules. This will result in the same source code as explicitly programming each version and variation of the algorithm by hand. In the other case, the compiler will keep the function calls. Our new concept is designed for compilers which can deal with inlining and then achieves the best performance. With slight modifications our new technique is also applied to all matrix classes (see figure 1).

4 DISCUSSION

On the basis of some examples we will now show the efficiency of our new concept described in the previous sections. For the following running tests we have used LiDIA 1.2, LiDIA 1.3 [11] (which already includes the new lattice concept but does not support the bit fields yet), PARI 1.39.03 [3], Mathematica 2.2[4] [15] and Maple V Release 3[5] [6]. In addition we have tested a preliminary version of LiDIA 2.0 which includes the bit field mechanisms.

At first we have tested a program sequence of computing the Hermite normal form of a regular $n \times n$ matrix with randomly chosen integer entries in $[0, \ldots, 10^4]$ followed by transposing the resulting matrix and computing their determinant. The Hermite normal form of a regular $n \times n$ matrix is an upper triangular matrix. Knowing this information their determinant can simply be computed by multiplying the diagonal entries instead of using more elaborate techniques designed for computing the determinant of arbitrarily chosen matrices.

n	LiDIA 1.3	LiDIA 2.0	PARI	Maple
10	0.03	0.02	0.04	0.63
20	0.37	0.28	0.82	8.53
30	2.03	1.14	6.36	56.95
40	5.26	4.36	29.30	237.28
50	20.89	16.86	104.62	761.78
60	56.51	52.00	297.37	1797.45
70	141.14	135.01	737.03	3653.92
80	340.37	330.58	1609.53	–
90	844.04	799.13	–	–
Timings in seconds on a Sparc20				

Table 1: Program Sequence $det((HNF(A))^T)$

In table 1 '–' symbolizes that the running time was longer than 6000 seconds when the computation was aborted.

Comparing LiDIA 1.3 and the preliminary version of LiDIA 2.0 the timings in table 1 show that by using the bit field mechanisms we achieve a major speed-up. It results from the fact that the HNF computation manipulates the bit fields such that the flags for coding the triangular structure of the matrix is set. This information can therefore be used to speed-up further computations.

As another example we have computed the LLL reduction of integer lattices using the implementation of the original Schnorr-Euchner algorithm [12] of the `bigint_lattice` (reduction parameter 0.99) based on the new concept described in section 3.

Table 2 lists timings for sparse $(n+1) \times n$ lattices where the last row contains randomly chosen integer entries in $[0, \ldots, 10^{25}]$.

n	LiDIA 1.3	LiDIA 1.2	PARI	Maple	Mathematica
10	0.03	0.07	0.32	16.92	1.70
20	0.35	0.59	3.89	95.23	6.67
30	0.98	1.76	8.97	270.65	16.93
40	2.70	4.02	29.83	530.05	28.65
50	3.68	5.19	35.02	738.72	42.42
60	4.28	6.32	52.42	1071.43	61.13
70	4.84	6.61	70.31	1451.08	85.38
80	5.95	7.73	76.10	1938.90	107.90
90	7.89	9.24	101.19	2733.40	133.38
100	8.23	11.00	144.29	3634.63	162.88
Timings in seconds on a Sparc20					

Table 2: LLL reduction for sparse matrices

[4] Mathematica is a trademark of Wolfram Research, Inc.
[5] Maple is a trademark of Waterloo Maple Software

Table 3 lists the running timings for dense $n \times n$ lattices with randomly chosen integer entries in $[0, \ldots, 10^{21}]$.

n	LiDIA 1.3	LiDIA 1.2	PARI	Maple	Mathematica
5	0.03	0.06	0.11	5.45	0.57
10	0.19	0.36	1.78	112.47	4.58
15	0.97	2.12	24.41	1391.03	27.87
20	3.63	7.42	119.07	6168.70	70.60
25	13.75	21.27	465.42	–	169.43
30	32.03	38.71	1555.31	–	365.23
Timings in seconds on a Sparc20					

Table 3: LLL reduction of dense matrices

In tables 2 and 3 '–' symbolizes that the running time was longer than 6000 seconds when the computation was aborted or the result was not correct.

The timings show that the new concept does not imply a loss of efficiency of our algorithms but even results in a slight speed-up which was achieved by improved inlining. Altogether, the examples document a major advantage comparing the timings of LiDIA 1.3 to the ones of LiDIA 1.2 as well as Maple, PARI and Mathematica. This shows the efficiency of our new concepts.

5 CONCLUSIONS

In this paper we have proposed a new design concept for sparse and dense matrices as well as lattices which allows mixed representations and uses bit fields for storing specific information (section 2.1). Consequently, we achieve a major speed-up for single operations and especially for sequences of operations (section 4). In addition we have introduced a new template technique (section 3). In combination, our new concepts ensure

- **Flexibility** - Good adjustment of the matrix and lattice classes to the requirements of specific applications by collecting structure and high-level information.
- **Good Maintenance** - As less code duplication as possible.
- **Extendibility** - Easy integration of new versions and variations of algorithms and data-types by defining new template modules and extending the template kernel.

For the future we plan to integrate the new programming technique of expression templates [14] which should result in another major speed-up of the operations. In addition we are working on extending the functionality of the matrix and lattice classes in LiDIA.

References

[1] Babai, L.: *On Lovász' Lattice Reduction and the Nearest Lattice Point Problem.* Combinatorica 6, 1–13 (1986).

[2] Barton, J.J., and Nackman, L.R.: *Scientific and Engineering C++ - An Introduction with Advanced Techniques and Examples.* Addison Wesley (1994).

[3] Batut, C., Bernardi, D., Cohen, H., and Olivier, M.: *User's Guide to PARI-GP.* University of Bordeaux, `ftp://megrez.math.u-bordeaux.fr` (1995).

[4] Biehl, I., Buchmann, J., and Papanikolaou, T.: LiDIA: *A Library for Computational Number Theory.* Technical Report 03/95, SFB 124, Universität des Saarlandes (1995).

[5] Buchmann, J., and Kessler, V.: *Computing a Reduced Lattice Basis from a Generating System.* Preprint, Universität des Saarlandes (1992).

[6] Char, B.W., Geddes, K.O., Gonnet, G.H., Leong, B.L., Monagan, M.B., and Watt, S.M.: *Maple V Library Reference Manual.* Springer-Verlag and Waterloo Maple Publishing (1991).

[7] Cohen, H.: *A Course in Computational Algebraic Number Theory.* Second Edition, Springer Verlag Heidelberg (1993).

[8] Fincke, U., and Pohst, M.: *Improved Methods for Calculation Vectors of Short Length in a Lattice, including a Complexity Analysis.* Math. Comp. 44, 463–471 (1985).

[9] *Kernighan, B.W., and Ritchie, D.M.:* The C Programming Language. Second Edition, Prentice Hall, Englewood Cliffs, New Jersey (1988).

[10] Lenstra, A. K., Lenstra, H. W. Jr., and Lovász, L.: *Factoring Polynomials with Rational Coefficients.* Math. Ann. 261, 515–534 (1982).

[11] LiDIA Group: LiDIA *Manual.* Universität des Saarlandes/TH Darmstadt, see LiDIA homepage: `http://www-jb.cs.uni-sb.de/LiDIA/linkhtml/lidia/lidia.html` (1996).

[12] Schnorr, C.P., and Euchner, M.: *Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems.* Springer Lecture Notes in Computer Science LNCS 529, 68–85 (1991).

[13] Stroustrup, B.: *The C++ Programming Language.* Second Edition, Addison Wesley (1991).

[14] Veldhuizen, T.: *Expression Templates.* Developer Information and Resources, Rogue Wave Software, Inc. (1994).

[15] Wolfram, S.: *Mathematica - A System for Doing Mathematics by Computer.* Second Edition, Addison Wesley (1991).

Patrick Théobald is a Ph.D. student at the Lehrstuhl für Theoretische Informatik at the Technische Hochschule in Darmstadt, Germany. He holds a Dipl.-Inform. degree from the Universität des Saarlandes, Germany. His research interest is in Computational Number Theory. He is currently supported by the Deutsche Forschungsgemeinschaft.

Susanne Wetzel is a Ph.D. candidate at the Universität des Saarlandes, Germany and holds a scholarship from the Deutsche Forschungsgemeinschaft. She received her Dipl.-Inform. degree from the Universität Karlsruhe (TH), Germany. Her current research interests are in Computational Number Theory and Cryptography.

Werner Backes is a Master student at the Universität des Saarlandes, Germany. His research interest is in Computational Number Theory.