Sieving for Shortest Vectors in Ideal Lattices: a Practical Perspective

Joppe W. Bos[1], Michael Naehrig[2], and Joop van de Pol[3⋆]

[1] NXP Semiconductors, Leuven, Belgium
joppe.bos@nxp.com
[2] Microsoft Research, Redmond, USA
mnaehrig@microsoft.com
[3] Dept. Computer Science, University of Bristol, United Kingdom
joop.vandepol@bristol.ac.uk

Abstract. The security of many lattice-based cryptographic schemes relies on the hardness of finding short vectors in integral lattices. We propose a new variant of the parallel Gauss sieve algorithm to compute such short vectors. It combines favorable properties of previous approaches resulting in reduced run time and memory requirement per node. Our publicly available implementation outperforms all previous Gauss sieve approaches for dimensions 80, 88, and 96.
When computing short vectors in ideal lattices, we show how to reduce the number of multiplications and comparisons by using a symbolic Fourier transform. We computed a short vector in a negacyclic ideal lattice of dimension 128 in less than nine days on 1024 cores, more than twice as fast as the recent record computation for the same lattice on the same computer hardware.

Keywords: Lattice cryptanalysis, parallel Gauss sieve, ideal lattices, ring LWE

1 Introduction

Since the late 1990s, there has been increasing interest in constructing cryptographic functions with security based on the computational hardness of lattice problems, initiated by works such as [2, 3, 26]. *Lattice-based cryptography* has become a promising source of cryptographic primitives for several reasons. One of them is its post-quantum potential (cf. the survey [39]). Another reason is that it allows new capabilities that are not possible with classical schemes: e.g. fully homomorphic encryption [21] and multilinear maps [19]. Many of the most recent lattice-based systems reduce their security to the hardness of the learning with errors (LWE) problem [47] or its ring variant, the ring-learning with errors (R-LWE) problem [35]. The latter provides an additional algebraic ring structure and is often preferred for efficiency. Ideals in the ring $\mathbb{Z}[X]/(X^n - 1)$ have already been used in the NTRU cryptosystem [26]; multiplication of polynomials is a cyclic convolution and [26] notes that it can be computed efficiently with the Fast Fourier Transform (FFT). For R-LWE, the ideals lie in the ring of integers of a cyclotomic number field. Arithmetic in such a ring is performed in $\mathbb{Z}[X]/(\Phi_m(X))$ where $\Phi_m(X)$ is the m-th cyclotomic polynomial of degree $n = \varphi(m)$. A popular choice is $m = 2^k$, since then $n = 2^{k-1}$ and $\Phi_{2^k}(X) = X^{2^{k-1}} + 1$. This polynomial is maximally sparse and polynomial arithmetic can be performed efficiently using the FFT [34]. Many lattice-based constructions (e.g. [11, 51, 10, 9]) and most implementations (e.g. [34, 24, 25, 6, 45, 8, 9]) use this particular instance of a cyclotomic polynomial.

The additional structure raises questions about the security of the resulting schemes. Can a cryptanalyst use this structure and break the scheme? How hard are these resulting R-LWE

⋆ Part of this work was done while the third author was an intern in the Cryptography Research group at Microsoft Research.

instances in practice? In this paper, we look to the area of lattice cryptanalysis to answer these questions.

Lattice Cryptanalysis. There are roughly two research directions in lattice cryptanalysis. One direction is concerned with algorithms that provably solve lattice problems such as the (approximate) shortest and closest vector problems. Such algorithms include LLL [31], BKZ [50], enumeration [16, 28], sieving [4, 54], algorithms based on Voronoi-cell computations [40] and more recently on Discrete Gaussian Sampling [1]. The other direction considers algorithms based on heuristics which can be used to solve lattice problems efficiently in practice, trying to push the dimension of the lattice for which the problem can be solved as high as possible. Here, the aforementioned provable algorithms are often taken and modified according to some heuristics to achieve better practical performance. This has been done for BKZ [13], enumeration [18] and sieving [42, 55, 56, 54, 30]. Recently a novel heuristic algorithm similar to sieving was proposed that is not based on any provable algorithm [7].

The hardness of the R-LWE problem is related to the difficulty of finding a sufficiently short non-zero vector in a lattice associated to an ideal in a cyclotomic ring. E.g., the distinguishing attack from [39] requires a short vector of a certain length to achieve a given distinguishing advantage. Hence, parameters for R-LWE-based schemes are chosen such that polynomial run-time exponential-approximation algorithms such as LLL [31] are unlikely to find such short vectors (cf. [52, 32]). The known algorithms which are capable of finding exact solutions to the lattice problem all have exponential run time. Furthermore, there seems to be a disconnect between theory and practice. In theory, enumeration has a super-exponential asymptotic running-time complexity in the lattice rank, whereas sieving algorithms have only single-exponential run-time complexities (at the necessary cost of exponential space requirements). In practice, enumeration algorithms using heuristic extreme pruning [18] in conjunction with heuristically improved BKZ [13] seem to perform best as can be seen from the results of the SVP challenge [49]. Implementations of the heuristic versions of sieving algorithms cannot reach as high dimensions and take more time for lower dimensions. Also, in the majority of cryptographic applications an attacker only needs an approximate shortest vector to break the scheme. Hence, algorithms that give an exact solution such as enumeration and sieving do more work than necessary, and BKZ appears to be a better option for the cryptanalyst.

So is all then lost for sieving-based algorithms? Under suitable heuristic assumptions, the asymptotically faster run times still hold, and experimental results suggest that run times lie on an exponential curve with reasonable constants. This suggests that there exists a cross-over point in the size of the dimension beyond which sieving algorithms beat enumeration-based algorithms in practice. While the latter are superior for dimensions we are able to solve in practice (roughly up to 140), cryptographic schemes typically have parameters far out of reach of current capabilities [25, 45, 8]. For these, dimensions are at least a factor two larger and for some applications such as homorphic encryption the dimensions are orders of magnitude greater [22, 52, 32]. Even if the cross-over point turns out to be much higher than the dimensions for current parameter sets, it is still interesting to determine its exact location to assess a potential effect on parameter selection. Another reason to look at sieving algorithms is that algorithms like BKZ and enumeration manipulate the Gram-Schmidt orthogonalization, which does not retain the ideal lattice structure due to projection. This prevents such algorithms from taking advantage of it. As sieving methods work with actual lattice vectors, they seem to be the only type of algorithm that can take advantage of this structure in order to find short vectors.

However, the above reasoning only applies to acquiring vectors of very short length, i.e., solving SVP almost exactly. As mentioned, in practical cryptographic applications an approximate solution often suffices (where the approximation factor is generally polynomial in the dimension). In this setting, BKZ with its heuristic improvements seems to be the better algorithm. Still, we think it is worthwhile to explore the use of the additional structure by sieving because in the future sieving might become applicable even to cryptographic instances. For example, [33, Lemma 10] shows that certain sieving algorithms achieve better asymptotic run times when used as an approximation rather than an exact algorithm. Furthermore, if cryptographic constructions can tighten the approximation factor required to break the scheme, BKZ will have to use larger block sizes, and hence come closer to exact algorithms.

Related work. One such algorithm that can take advantage of the ideal structure is Gauss sieve, due to Micciancio and Voulgaris [54] who already observed this property. It was later examined by Schneider [48] and Ishiguro et al. [27]. It is useful for an implementation to support efficient parallelization such that it can take advantage of modern hardware architectures. There are three approaches that have been attempted to parallelize Gauss sieve, one due to Milde and Schneider [41], one due to Ishiguro et al. [27] and more recently one due to Mariano et al. [37]. The first is a distributed approach which runs into synchronization issues, which the second aims to solve by using shared memory and a synchronized algorithm. The third instead uses shared memory combined with special techniques to resolve concurrency issues.

Contributions. In this paper we take a close look at the Gauss sieve algorithm from a *practical* point of view. Our contributions are two-fold. Firstly, we propose yet another variant of Gauss sieve which is inspired by the ideas from both Milde and Schneider [41] and Ishiguro et al. [27]. We keep the concept adopted by Milde and Schneider to split up the global list of vectors across all compute instances and do not use the method of Ishiguro et al. to have all instances maintain a full copy of the list. This evenly distributes the required storage space for the list vectors among the instances instead of duplicating it. We deviate from [41] how sampled and reduced vectors are treated. Instead of sampling vectors locally and propagating them through the system in a circular fashion, we sample new vectors and collect reduced vectors globally, to feed them into the system by broadcasting a batch of the same vectors to all computational units. This approach is closer to [27] and ensures that arbitrary pairs of vectors from the list are Gauss reduced after each round of computation; this lowers the global maximum list size and therefore the expected total run time.

Secondly, we extend the approach taken by Schneider [48] and Ishiguro et al. [27] to use the ideal structure in the ring $\mathbb{Z}[X]/(X^n + 1)$. As done previously, we use that one can represent the n rotations $X^i \cdot \mathbf{a}$ (of the same norm as $\mathbf{a}$) by only storing a single vector $\mathbf{a}$, providing a factor n reduction in storage for the same list size. Beyond that, we use rotations to improve the efficiency of computing scalar products, which are used to check whether two vectors are Gauss-reduced. In Section 4, we show how all the scalar products of one vector with the n rotations of a second vector can be computed at once via the FFT (cf. [34] for the application in the cryptographic setting). This provides an improvement from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \ln n)$ operations over the naive way of computing all scalar products separately. We also show that comparing the absolute values of n such products to the square of the norms of the two original vectors is sufficient to decide whether all n^2 pairs of rotations of the two vectors are Gauss reduced. This decreases the number of comparisons from n^2 to n. In Section 4.4

we point out that our techniques also apply to other special rings such as $\mathbb{Z}[X]/(X^n - 1)$ as used in the NTRU cryptosystem [26].

We have implemented our parallel Gauss sieve variant for general (non-ideal) lattices with additional stages of optimizations for negacyclic ideal lattices. The first stage uses vector rotations as in previous approaches and the second stage deploys all optimizations mentioned above, including the FFT to compute scalar products. Our approach is suitable for single instruction, multiple data (SIMD) platforms and we have optimized our implementation by using commonly available vector instruction set extensions. The source code of our implementations is publicly available[4] and we hope that it will serve as a basis for more experiments. The practical benefits of our algorithm are demonstrated by computing short vectors for generic lattices of dimensions 80, 88, and 96, using a variable number of nodes, clearly indicating the communication overhead. In the ideal lattice setting, we show the practical benefits of working in $\mathbb{Z}[X]/(X^{2^7} + 1)$ of dimension 128 by computing the ring multiplication using Nussbaumer's symbolic FFT [43].

2 Preliminaries

Consider the Euclidean space $\mathbb{R}^n$ with its usual topology. We denote column vectors by bold letters, the inner product of two vectors by $\langle \mathbf{x}, \mathbf{y} \rangle$ and the Euclidean norm by $\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle}$. For real numbers r, we define $\lfloor r \rceil = \lfloor r + \frac{1}{2} \rfloor$ to be the integer closest to r (rounded up if not unique). A *lattice* L is a discrete subgroup of $\mathbb{R}^n$. We only consider full-rank, integral lattices $L \subset \mathbb{Z}^n$, represented by a basis $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$. The lattice $L = \mathcal{L}(B)$ consists of all linear combinations with integral coefficents of the basis vectors, i.e., $\mathcal{L}(B) = \{\sum_{i=1}^{n} \lambda_i \mathbf{b}_i : \lambda_i \in \mathbb{Z}\}$. We denote by $\lambda_1(L)$ the length of a shortest non-zero vector in L, i.e., $\lambda_1(L) = \min_{\mathbf{x} \in L \setminus \{\mathbf{0}\}} \|\mathbf{x}\|$. The *Shortest vector problem (SVP)* is defined as follows: given a basis B of a lattice $L = \mathcal{L}(B)$, find a vector $\mathbf{v}$ in L such that $\|\mathbf{v}\| = \lambda_1(L)$.

A central notion for describing the Gauss sieve is that of Lagrange or Gauss reduction. Two vectors $\mathbf{x}$ and $\mathbf{y}$ are called *Gauss reduced* if $2 \cdot |\langle \mathbf{x}, \mathbf{y} \rangle| \leq \min\{\langle \mathbf{x}, \mathbf{x} \rangle, \langle \mathbf{y}, \mathbf{y} \rangle\}$. Equivalently, this means that $\|\mathbf{x} \pm \mathbf{y}\| \geq \max\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$ and hence that the two-dimensional lattice spanned by the basis $\{\mathbf{x}, \mathbf{y}\}$ does not contain two linearly independent vectors $\mathbf{x}'$ and $\mathbf{y}'$ such that either $\|\mathbf{x}'\| < \min\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$ or $\|\mathbf{x}'\| \leq \|\mathbf{y}'\| < \max\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$. Any basis $\{\mathbf{a}, \mathbf{b}\}$ of a 2-dimensional lattice can be Gauss reduced by repeating the following two steps until $\mathbf{b}$ does not change: (1) If $\|\mathbf{b}\| < \|\mathbf{a}\|$, swap $\mathbf{a}$ and $\mathbf{b}$. (2) Replace $\mathbf{b}$ by $\mathbf{b} - \left\lfloor \frac{\langle \mathbf{a}, \mathbf{b} \rangle}{\langle \mathbf{a}, \mathbf{a} \rangle} \right\rceil \cdot \mathbf{a}$. In the remainder of the paper, we use the following terminology: a vector $\mathbf{y}$ *can be reduced* with respect to a vector $\mathbf{x}$, if $2 \cdot |\langle \mathbf{x}, \mathbf{y} \rangle| > \langle \mathbf{x}, \mathbf{x} \rangle$. In this case, *reducing* $\mathbf{y}$ with respect to $\mathbf{x}$ means, replacing $\mathbf{y}$ by $\mathbf{y} - \left\lfloor \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\langle \mathbf{x}, \mathbf{x} \rangle} \right\rceil \cdot \mathbf{x}$. We sometimes also mean *reducing* $\mathbf{y}$ with respect to $\mathbf{x}$ to include checking the condition $2 \cdot |\langle \mathbf{x}, \mathbf{y} \rangle| > \langle \mathbf{x}, \mathbf{x} \rangle$, and if it is not satisfied, no further computation takes place and the vector $\mathbf{y}$ remains unchanged.

Ideal lattices are lattices that arise from ideals in a ring R. For example, consider the ring $R = \mathbb{Z}[X]/(f)$, where f is a monic polynomial of degree n, such that R consists of polynomials with degree at most $n - 1$. One way of associating a lattice to an ideal is via the so-called *coefficient embedding*. A polynomial in R can be identified with its coefficient vector, which is an element from $\mathbb{Z}^n$. Any ideal $I \subseteq R$ is an additive subgroup, and therefore the corresponding coefficient vectors form a lattice over $\mathbb{Z}^n$. Furthermore, an ideal is closed under multiplication

[4] See: `http://www.joppebos.com/src/ParallelGaussSieve-1.0.tgz`.

with arbitrary ring elements $r \in R$. Hence, the ideal lattice inherits an additional algebraic structure from the ring multiplication. If f is irreducible, then for any $v \in R$, the coefficient vectors corresponding to v, $X \cdot v$, ..., $X^{n-1} \cdot v \mod f$ are linearly independent and therefore together span a full-rank lattice. This means that the lattice corresponding to the principal ideal generated by v can be represented by using only one element v.

Gauss Sieve. The Gauss sieve algorithm was described by Micciancio and Voulgaris in [54] (see Algorithm 3 in Appendix A for an outline including some modifications). The input to Gauss Sieve consists of a basis $\mathbf{B}$, a target length μ and a maximum number of collisions c. Given a pairwise Gauss reduced list of vectors, it samples a new vector $\mathbf{v}_{\text{new}}$ and ensures that it is Gauss reduced with respect to all vectors in the list. This is achieved by reducing $\mathbf{v}_{\text{new}}$ with respect to list vectors shorter than itself and inserting it in the list. Finally, the (updated) $\mathbf{v}_{\text{new}}$ is used to reduce list vectors longer than itself. List vectors that are reduced are moved to the stack (since it is not guaranteed that they are still Gauss reduced with respect to all other list vectors). The sampling procedure takes vectors from the stack before sampling new ones.

The algorithm terminates either when a vector $\mathbf{v} \in \mathcal{L}(B)$ is found such that $\|\mathbf{v}\| \leq \mu$ or when the number of collisions is at least c. The motivation for the latter condition is that we do not have to specify a target length μ. One expects (heuristically) that when a short vector is found, it is found repeatedly (by subtracting different vectors), resulting in many collisions, i.e. vectors being "lost" when they are reduced to zero. This reasoning seems valid in practice, the proportion of collisions appears to increase dramatically, once a shortest vector of the lattice is in the list. In practice, one typically adapts the algorithm such that the termination condition depends exclusively either on the target length μ or the collision bound c.

The maximum list size of Gauss sieve can be bounded using the kissing number, but there is no provable bound on the run time of the algorithm, mostly because there is no bound on the number of collisions. Because all pairs of vectors need to be Gauss-reduced, the run time is at least quadratic in the list size, but practical experiments by Voulgaris and Micciancio suggest a run time of $2^{0.48n}$.

3 Parallel Gauss Sieve

In order to tap into the vast wealth of modern computing power, it is preferable that algorithms can be computed on many computational cores concurrently. In this section we briefly examine previous attempts to compute the Gauss sieve in parallel and propose modifications resulting in our new variant of the algorithm.

Non-synchronized, circular parallelization. Milde and Schneider [41] split up the problem into several "instances" and connect each instance in a circular fashion. Each instance is computed on a separate computational unit and consists of a list, a stack and a queue (all local). Each instance acts as an independent Gauss sieve, taking new vectors from the queue (filled by its neighboring instance), stack (filled by its local Gauss sieve) and the sampler in that order. However, instead of inserting vectors in the list once they are pairwise reduced with respect to the list (as in Algorithm 3 in Appendix A), they are sent on to the queue of the next instance. Only vectors that have made a full round across all instances are inserted. In this scenario, there is no explicit synchronization between nodes. All communication consists of vectors being sent from one instance to the next.

The data in [41] shows that this method does not appear to scale well with the number of parallel instances. It seems that the list sizes vary greatly, and nodes with small lists sample lots of new vectors, which in turn get stuck in the queues of nodes with bigger lists. Attempts to work around these traffic jams by relaxing the pairwise reducedness between instances cause the global list size to increase dramatically. Another side-effect is that a vector inserted in a local list is not necessarily reduced with respect to vectors in the queue and stack of other instances. The number of such vectors can be significant and this property increases the overall list size and hence the run time of the algorithm.

Synchronized parallelization. To resolve this issue, Ishiguro et al. [27] propose to have a copy of the full global list available to each computational unit (instead of splitting it up across the units) and instead, process the vectors in batches. This process consists of three stages: reducing the new batch of vectors with respect to the global list, reducing the new batch of vectors with respect to each other and reducing the list vectors with respect to the batch vectors. If at any stage any vector is reduced, it is removed and moved to the (global) stack, whereas all vectors that "survived" the three stages comprise the new list. Both before and after these three stages, every pair of vectors in the list is guaranteed to be reduced. This results in a smaller overall list size and lower run time. In experiments, this approach was used to solve a 128-dimensional ideal lattice challenge. The obvious drawback is that all nodes need to have access to the full list.

3.1 A New Parallel Gauss Sieve Approach

We propose a modified version of the parallel Gauss sieve algorithm which combines the best of both previous approaches. Each node maintains its own local list as in the approach by Milde and Schneider (reducing the storage per node), but the algorithm consists of synchronized rounds which ensure that all vectors in the union of all local lists are pairwise reduced as in the Ishiguro et al. approach (lowering the list size and therefore the run time). Sampling of vectors is done globally for all nodes at one node. Each vector in a newly sampled batch is reduced with respect to all vectors in the local list of each node. Then each node reduces a selection of the batch vectors with respect to the other batch vectors. Next, all nodes communicate which sample vectors have survived globally, i.e. which vectors are pairwise Gauss reduced with respect to all list vectors and each other, and what the sizes of their local lists are. The node with the smallest local list inserts the surviving batch vectors and the next round begins. The algorithm is depicted in Figure 1. Algorithm 1 is an algorithmic description of this process, in which the steps that require communication are marked in red and are underlined. Next, we give a detailed description.

Node layout and splitting of the global list. Let there be N nodes available for the parallel computation, and denote by N_i the i-th node for $0 \leq i < N$. Each node N_i locally maintains a list L_i, a list of samples Q_i, a list of reduced samples Q'_i and a stack S_i. The union $L = \bigcup_i L_i$ of the local lists can be viewed as the global list in the Gauss sieve (in the form of Algorithm 3, note that Algorithm 1 does not require the lists to be sorted). One node (here node N_0) also maintains the sampling procedure and a global stack S from which vectors are taken into the global queue Q. The global stack S can be equal to the local stack S_0, which is done in Algorithm 1 and our implementation.

Broadcast of new samples. At the start of each round, the first node N_0 compiles a list Q of k new samples by taking them from the stack S or sampling them using the GPV

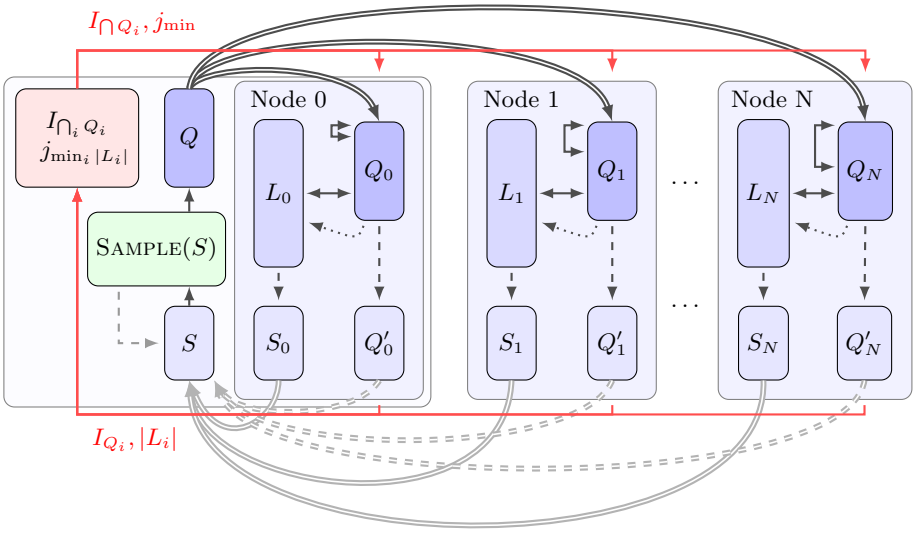

Fig. 1. Parallel Gauss Sieve algorithm. Double, dark grey arrows depict the broadcast of vectors; double-sided, single-line arrows represent mutual Gauss reduction; dashed, single-line arrows stand for removing reduced vectors. Double, light grey arrows show the sending of stack vectors; dashed, double, light grey arrows show the sending of minimal representatives. Communication of indices of survivors in the local stacks, local list sizes and indices of vectors to be inserted are represented by the red arrows.

algorithm [23]. The number of samples k is a parameter of the algorithm. It then broadcasts Q to all other nodes and each node N_i copies Q into their local sample list Q_i.

Reduction phase. Each node now locally compares all pairs of vectors from Q_i and L_i to see if they are pairwise reduced and reduces vectors wherever possible. Vectors from Q_i that are reduced are removed and stored in the reduced samples list Q'_i, whereas vectors from L_i that are reduced are removed and stored in the local stack S_i. When all pairs have been checked, each node compares the surviving samples that are still in Q_i to each other. This is done in a structured way to split the work between all nodes: each node only checks a predetermined subset of Q against the rest of Q_i. Again, any sample that is reduced during this stage is removed from Q_i and added to Q'_i. Note that, although lines 7-8 and 10-11 of Algorithm 1 both list the reduction twice, they require only a single inner product, as they make the two comparisons $2 \cdot |\langle \mathbf{v}, \mathbf{l} \rangle| > \langle \mathbf{l}, \mathbf{l} \rangle$ and $2 \cdot |\langle \mathbf{l}, \mathbf{v} \rangle| > \langle \mathbf{v}, \mathbf{v} \rangle$, which share the same left-hand term.

Inserting vectors into the list. At the end of the reduction phase any two vectors from the set $(\bigcup_i L_i) \cup (\bigcap_i Q_i)$ are Gauss reduced. This is the case because this set contains exactly the *survivors* of the reduction phase, i.e. all vectors that could not be reduced with respect to any other vector in this set. The surviving sample vectors are the vectors in the set $\bigcap_i Q_i$ which need to be inserted into the global list $L = \bigcup_i L_i$, i.e. the collection of local lists, and it has to be decided into which local list L_i they are inserted. In order to determine this, the nodes collectively compute $\bigcap_i Q_i$, i.e. the set of original Q-vectors that have survived, and also which node holds the smallest list. The first task requires the computation of a collective bitwise AND on a bitmask that lists the surviving vectors per node. The second requires a gathering of the list-sizes and a broadcast of the result (both are elementary standard operations available in any parallel computation library). The node with the smallest list inserts all the surviving

Algorithm 1 Parallel Gauss sieve variant. Given a basis $\mathbf{B}$, length bound μ, and list of N nodes N_i, $0 \leq i < N$, return a short vector $\mathbf{v}$ with $\|\mathbf{v}\| \leq \mu$ as soon as it is found. Let (L_i, Q_i, Q'_i, S_i) resp. be the list, sample list, non-survivor list and stack of node N_i. Communication steps are red and underlined.

function GaussSieve($\mathbf{B}, \mu, \{N_i\}_{i=0}^{N}$)
  while short vector not found **do**
    N_0 samples a list Q of k vectors via Sample(S_0) (see Algorithm 3)
    N_0 broadcasts Q
    for i in {0,1,...,N-1} **do** $Q_i \leftarrow Q$, $Q'_i \leftarrow \varnothing$, $S_i \leftarrow \varnothing$
      for $(\mathbf{v}, \mathbf{l}) \in Q_i \times L_i$ **do**
        if Reduce($\mathbf{v}, \mathbf{l}$) **then** Move $\mathbf{v}$ from Q_i to Q'_i
        if Reduce($\mathbf{l}, \mathbf{v}$) **then** Move $\mathbf{l}$ from L_i to S_i
      for $(\mathbf{v}, \mathbf{l}) \in Q_i \times Q_i$ **do**
        if Reduce($\mathbf{v}, \mathbf{l}$) **then** Move $\mathbf{v}$ from Q_i to Q'_i
        if Reduce($\mathbf{l}, \mathbf{v}$) **then** Move $\mathbf{l}$ from Q_i to Q'_i
    Compute $\bigcap_i Q_i$
    for $\mathbf{v} \in (\bigcap_i Q_i)$ **do**
      if $\|\mathbf{v}\| \leq \mu$ **then return** $\mathbf{v}$
    Compute index j such that $|L_j|$ is minimal, $L_j \leftarrow L_j \cup (\bigcap_i Q_i)$
    for $\mathbf{v} \in Q \setminus (\bigcap_i Q_i) = \bigcup_i Q'_i$ **do** Add minimal representative of $\mathbf{v}$ to S_0
    for $\mathbf{v} \in \bigcup_i S_i$ **do**
      if $\|\mathbf{v}\| \leq \mu$ **then return** $\mathbf{v}$
    for i in {0,1,...,N-1} **do** N_i sends S_i to N_0
    $S_0 = S_0 \cup (\bigcup_i S_i)$

vectors into its list, which requires *no additional communication* because the survivors are exactly those vectors from the original Q that were unchanged by all nodes.

Recycling of reduced vectors. The last step of the round is to gather all vectors that were reduced in this round in the global stack S. These are the reduced samples in the Q'_i, together with the reduced list vectors in the local stacks S_i. Not all vectors in $\bigcup_i Q'_i$ are propagated into the global stack for sampling because this leads to unnecessary collisions. A vector $\mathbf{v}$ which is reduced at two different nodes N_i and N_j by two different list vectors $\mathbf{l}_i$ and $\mathbf{l}_j$, produces two vectors $\mathbf{v}_i \in Q'_i$ and $\mathbf{v}_j \in Q'_j$. If both of them are passed to the next rounds and are inserted into Q, it is likely that $\mathbf{v}_i$ will be reduced at node N_j by $\mathbf{l}_j$ and $\mathbf{v}_j$ at N_i by $\mathbf{l}_i$, leading to the same vector going back to the global stack at the end of the round. This behavior can be (and has been) observed in practice. Therefore, from all vectors that arise from the same sample vector by reduction with respect to different list vectors, we only propagate the vector with the minimal norm. We call this vector the *minimal representative* of $\mathbf{v}$. For each non-survivor from the original list Q, this requires a collective computation across all nodes of the vector with the minimal norm among the vectors in $\bigcup_i Q'_i$ that originate from the same vector in Q. The node that contains this *minimal representative* of the non-survivor sends it to the first node. Finally, the vectors in the local stacks S_i are all taken into the next round and each node sends the vectors from their stack S_i to the first node.

4 Sieving in Ideal Lattices

Ideal lattices are ideals in a ring R. We consider lattices which are ideals in the ring of integers of the cyclotomic number field $R = \mathbb{Z}[X]/(\Phi_m(X))$, where $m = 2n$ is a power

of 2 such that the cyclotomic polynomial is $\Phi_m(X) = X^n + 1$. Ideals are invariant under multiplication by arbitrary ring elements; if $a(X)$ is an element in an ideal, then $X \cdot a(X)$ also belongs to the ideal. This means that for any vector a in the ideal lattice, one obtains lattice vectors $X^i \cdot a$, $i \in \mathbb{Z}$. An element $a \in R$ is of the form $a(X) = \sum_{i=0}^{n-1} a_i X^i$ and is given by its coefficient vector $\mathbf{a} = (a_0, a_1 \ldots, a_{n-1}) \in \mathbb{Z}^n$. The coefficient vectors of $X^i \cdot a(X) \mod (X^n + 1)$ are denoted by $X^i \cdot \mathbf{a}$ for $i \in \mathbb{Z}$, and we call these vectors the *rotations* of $\mathbf{a}$. Indeed, the polynomial $X \cdot a(X) \mod (X^n + 1)$ corresponds to a negacyclic rotation of the coefficient vector $\mathbf{a}$: $X \cdot \mathbf{a} = (-a_{n-1}, a_0, \ldots, a_{n-2})$. Conversely, rotating to the left corresponds to $X^{-1} \cdot \mathbf{a} = -X^{n-1} \cdot \mathbf{a} = (a_1, \ldots, a_{n-1}, -a_0)$.

Schneider [48] and Ishiguro et al. [27] previously have used the fact that an ideal lattice contains all rotations of a lattice vector. Since a single stored vector actually represents its n rotations, the list storage is reduced by a factor n. However, these works do not seem to use the ring structure any further. In this section, we show how to use the relation between ring multiplication and the scalar products of rotations to further improve the efficiency of Gauss sieve.

The next lemma (the proof can be found in Appendix B) shows that by computing the n scalar products $\langle \mathbf{a}, X^\ell \cdot \mathbf{b} \rangle$, $0 \le \ell < n$, one can easily check whether all n^2 possible combinations of two vectors $X^i \cdot \mathbf{a}$ and $X^j \cdot \mathbf{b}$, $i, j \in \mathbb{Z}$, are Gauss reduced.

Lemma 1. *Let $a, b \in R = \mathbb{Z}[X]/(X^n + 1)$ with coefficient vectors $\mathbf{a}, \mathbf{b}$. If $2|\langle \mathbf{a}, X^\ell \cdot \mathbf{b} \rangle| \le \min\{\langle \mathbf{a}, \mathbf{a} \rangle, \langle \mathbf{b}, \mathbf{b} \rangle\}$ for all $0 \le \ell < n$, then $X^i \cdot \mathbf{a}$ and $X^j \cdot \mathbf{b}$ are Gauss reduced for all $i, j \in \mathbb{Z}$.*

If the condition in Lemma 1 is not satisfied, then either $\mathbf{a}$ or $\mathbf{b}$ can be reduced by some rotation of the other vector. Namely, if $2|\langle \mathbf{a}, X^\ell \cdot \mathbf{b} \rangle| > \langle \mathbf{b}, \mathbf{b} \rangle$, then $\mathbf{a}$ can be reduced by $X^\ell \cdot \mathbf{b}$ and if $2|\langle \mathbf{a}, X^\ell \cdot \mathbf{b} \rangle| > \langle \mathbf{a}, \mathbf{a} \rangle$, then $\mathbf{b}$ can be reduced by $X^{n-\ell} \cdot \mathbf{a}$. Note that (by the proof of Lemma 1 in Appendix B) this is equivalent to saying that, if $2|\langle \mathbf{a}, X^\ell \cdot \mathbf{b} \rangle| > \langle \mathbf{b}, \mathbf{b} \rangle$, then any rotation of $\mathbf{a}$ can be reduced by some rotation of $\mathbf{b}$. The same holds for $\mathbf{b}$ with the second condition. The function `ReduceRot` in Algorithm 2 can be used instead of `Reduce` in the ideal lattice setting. When included in Algorithm 1, the checks of the inequalities in 7-8 and 10-11 can be combined as mentioned in Section 3.1 and require only n comparisons. Next, we show that all n scalar products can be computed by a single ring product. Since the ring product is a negacyclic convolution, it can be computed via an FFT algorithm in $\mathcal{O}(n \ln n)$ arithmetic operations instead of $\mathcal{O}(n^2)$ for a naive, separate computation of the scalar products.

4.1 Computing n Scalar Products by a Single Ring Product

Given two elements $a, b \in R$, let $c(X) = a(X) \cdot b(X) \mod (X^n + 1)$. In this subsection, we describe the relation of this product to the scalar products of the rotations of the elements a and b. For $a, b, c \in R$, let $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$, $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})$ and $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ be their coefficient vectors. Define the reflex polynomial of $b(X)$ as $b^{(R)}(X) = X^{n-1} \cdot b(X^{-1})$, and let $\mathbf{b}^{(R)} = (b_{n-1}, b_{n-2}, \ldots, b_0)$ be its coefficient vector. The coefficients c_i are given by the following equations involving scalar products of $\mathbf{a}$ with rotations of $\mathbf{b}^{(R)}$:

$$
\begin{array}{lllll}
c_0 & = a_0 b_0 - a_1 b_{n-1} - \cdots - a_{n-1} b_1 & = \langle \mathbf{a}, (b_0, -b_{n-1}, \ldots, -b_1) \rangle & = \langle \mathbf{a}, -X \cdot \mathbf{b}^{(R)} \rangle, \\
c_1 & = a_0 b_1 + a_1 b_0 - a_2 b_{n-1} - \cdots - a_{n-1} b_2 & = \langle \mathbf{a}, (b_1, b_0, -b_{n-1}, \ldots, -b_2) \rangle & = \langle \mathbf{a}, -X^2 \cdot \mathbf{b}^{(R)} \rangle, \\
\vdots & \vdots & \vdots & \vdots \\
c_{n-1} & = a_0 b_{n-1} + a_1 b_{n-2} \cdots + a_{n-1} b_0 & = \langle \mathbf{a}, (b_{n-1}, b_{n-2}, \ldots, b_0) \rangle & = \langle \mathbf{a}, \mathbf{b}^{(R)} \rangle.
\end{array}
$$

Algorithm 2 Algorithms for pairwise reducing $(\mathbf{a}, X^i \cdot \mathbf{b})$ for all $i \in \{0, \ldots, n-1\}$ The function `ReduceFFT` assumes as additional input the precomputed values $\hat{a} = \mathrm{FFT}(\mathbf{a})$ and $\hat{b}^{(R)} = \mathrm{FFT}(-X \cdot \mathbf{b}^{(R)})$.

```
1: function ReduceRot(a, b)
2:     for i = 0, ..., n − 1 do
3:         if 2 · |⟨a, X^i b⟩| > ⟨b, b⟩ then
4:             a ← a − ⌈⟨a, X^i b⟩ / ⟨b, b⟩⌋ X^i b
5:             return true
6:     return false
7: function ReduceFFT((a, â), (b, b̂^(R)))
8:     z ← FFT^{-1}(â ⊙ b̂^(R))
9:     for i = 0, ..., n − 1 do
10:        if 2 · |z_i| > ⟨b, b⟩ then
11:            a ← a − ⌈z_i / ⟨b, b⟩⌋ X^{n−i+1} · b
12:            return true
13:    return false
```

In general, this means that we get $c_i = \langle \mathbf{a}, -X^{i+1}\mathbf{b}^{(R)} \rangle$, $0 \leq i < n$. If we replace b by $-b^{(R)}(X)$ and instead compute $c(X) = a(X) \cdot (-b^{(R)}(X)) \mod (X^n + 1)$, we obtain in the coefficients of c the scalar products $c_i = \langle \mathbf{a}, X^{i+1} \cdot \mathbf{b} \rangle$ because $(b^{(R)})^{(R)}(X) = b(X)$. Now the last scalar product is $c_{n-1} = \langle \mathbf{a}, -\mathbf{b} \rangle$. Using one of the properties in Lemma 3 (in Appendix B) and computing c as the product with $-X \cdot b^{(R)}(X)$ instead, we get that $c(X) = a(X) \cdot (-X \cdot b^{(R)}(X)) \mod (X^n+1)$ has coefficients $c_i = \langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$ because the product $c(X) = (Xa(X)) \cdot (-b^{(R)}(X)) \mod (X^n + 1)$ has coefficients $c_i = \langle X \cdot \mathbf{a}, X^{i+1} \cdot \mathbf{b} \rangle = \langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$. We have proved the following lemma.

Lemma 2. *Let $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^n$ be two coefficient vectors corresponding to the elements $a, b \in R = \mathbb{Z}[X]/(X^n+1)$. Let $c(X) = a(X) \cdot (-X \cdot b^{(R)}(X)) \mod (X^n+1)$ and let $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}) \in \mathbb{Z}^n$ be its coefficient vector. Then $c_i = \langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$ for $0 \leq i < n$.*

This means that on input of two vectors $\mathbf{a}$ and $\mathbf{b}$ corresponding to polynomials $a(X)$ and $b(X)$ in R, by reordering the coefficients of b and adjusting the signs to get the polynomial $-X \cdot b^{(R)}(X)$ and then computing the ring product $c(X) = a(X) \cdot (-X \cdot b^{(R)}(X)) \mod (X^n+1)$ using an FFT-algorithm in $\mathcal{O}(n(\ln n)(\ln \ln n))$ bit operations (see the approach described in Section 4.2), one obtains all scalar products $c_i = \langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$ of $\mathbf{a}$ with the rotations of $\mathbf{b}$.

4.2 Nussbaumer's Algorithm for Negacyclic Convolutions

The product of two polynomials in R, i.e. a product $a(X)b(X) \mod (X^n + 1)$ where n is a power of 2, is a negacyclic convolution and we use Nussbaumer's symbolic algorithm [43] to compute it. Appendix C provides a technical description; see [29, Exercise 4.6.4.59] and [15, Section 9.5.7] for more details. We would like to point out that one can use other types of number theoretic transforms, e.g. see the fast implementation using vector instructions from [25].

The algorithm recursively reduces the negacyclic convolution of vectors of length n, by a re-organization of coefficients, to additions, subtractions and negacyclic convolutions of shorter vectors. It can be divided into three steps: first, the two polynomials a, b are re-ordered and split up into sequences A, B of length $2s$ (half of the entries being zero), each of which is converted to its DFT. The resulting sequences $\tilde{A}, \tilde{B}$ of length $2s$ are then multiplied coefficient-wise via negacyclic convolutions, resulting in a sequence $\tilde{C}$. Finally, this sequence is converted back from the DFT representation and unified into a single polynomial that corresponds to the negacyclic convolution of the two original polynomials. The second step

of computing negacyclic convolutions of shorter sequences can be expanded recursively by splitting each of the $2s$ coefficients of each sequence into $2s'$ pairs themselves, resulting in even smaller negacyclic convolutions that need to be computed. This is equivalent to expanding the conversions in the first and third steps to account for the levels of recursion and applying step two to all $4ss'$ sequence elements.

For a vector $\mathbf{a}$ corresponding to $a \in R$, we denote the DFT representation of $\mathbf{a}$ after the combined computations of the conversions described above by $\mathrm{FFT}(\mathbf{a})$. For two such vectors $\mathbf{a}, \mathbf{b}$, the pairwise negacyclic convolution of these DFT representations in step two above is denoted by $\mathrm{FFT}(\mathbf{a}) \odot \mathrm{FFT}(\mathbf{b})$. We denote the representation achieved by computing the inverse DFT of a vector $\hat{c}$ in step three by $\mathrm{FFT}^{-1}(\hat{c})$. Furthermore, let $\hat{a} = \mathrm{FFT}(\mathbf{a})$ and $\hat{b}^{(R)} = \mathrm{FFT}(-X \cdot \mathbf{b}^{(R)})$.

During the (possibly recursive) conversion in step one (when computing $\hat{a}$ and $\hat{b}^{(R)}$), the two polynomials do not interact. Therefore, this step only needs to be performed once and the result can be stored for each sampled vector in the Gauss sieve (significantly decreasing the cost of computing these n inner-products at the cost of additional memory). Then, when two vectors are compared, it only remains to compute the pairwise negacyclic convolutions ($\hat{c} = \hat{a} \odot \hat{b}^{(R)}$) and the (possibly recursive) reverse transformation ($\mathrm{FFT}^{-1}(\hat{c})$). As a result, these two latter steps account for most of the computation time during the algorithm. Functionality to compute such a DFT can be used to speed-up the reduction step in the Gauss sieve as outlined in the function `ReduceFFT` in Algorithm 2 and can be used instead of `Reduce` in the ideal lattice setting.

4.3 Nussbaumer's Algorithm Using SSE instructions

We assume that $\hat{a} = \mathrm{FFT}(\mathbf{a})$ and $\hat{b}^{(R)} = \mathrm{FFT}(-X \cdot \mathbf{b}^{(R)})$ are precomputed. Hence, the two most costly steps of Nussbaumer's algorithm are the negacyclic convolutions $\tilde{A}_i \tilde{B}_i \mod (Z^r + 1)$ and the inverse FFT transformations. We implemented Nussbaumer's algorithm completely using SSE instructions to target dimension 128 using two levels of recursion. In this case $n = n_1 = 128 = 2^{k_1} = s_1 \cdot r_1$, where $k_1 = 7$, $s_1 = 2^{\lfloor 7/2 \rfloor} = 8$ and $r_1 = 2^{\lceil 7/2 \rceil} = 16$. For the second level of recursion, we have $n_2 = 16 = 2^{k_2} = s_2 \cdot r_2$, where $k_2 = 4$, $s_2 = 2^{\lfloor 4/2 \rfloor} = 4$ and $r_2 = 2^{\lceil 4/2 \rceil} = 4$. As a result, step two computes $(2s_1)(2s_2) = 16 \cdot 8 = 128$ negacyclic convolutions in dimension $r_2 = 4$, which corresponds to the following computation:

$$z_0 = x_0y_0 - x_1y_3 - x_2y_2 - x_3y_1, \quad z_1 = x_0y_1 + x_1y_0 - x_2y_3 - x_3y_2,$$
$$z_2 = x_0y_2 + x_1y_1 + x_2y_0 - x_3y_3, \quad z_3 = x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0,$$

where $\mathbf{x} = (x_0, x_1, x_2, x_3)$ and $\mathbf{y} = (y_0, y_1, y_2, y_3)$ are one of the 128 pairs and $\mathbf{z} = (z_0, z_1, z_2, z_3)$ is our desired output.

We assume that the size of each vector coefficient is less than 2^{16} such that all coefficients can be stored in 16 bits. This eases adaptation in our vector instruction based implementation. The same assumption has also been made in previous implementations of the parallel Gauss sieve [27] and does not seem to pose any restrictions on the dimensions which are considered in practice nowadays. Moreover, we assume that the entries of the DFT representations after the transformation of step one fit in 16 bits. The main reason for this restriction is that in the second step two of these entries are multiplied together and such values are added which can be done efficiently using the SSE2 "multiply and add packed integers" instruction `PMADDWD` (available as the C-intrinsic `_mm_madd_epi16`). This instruction takes as input two 4-way SIMD 32-bit registers $((a_0, a_1), (a_2, a_3), (a_4, a_5), (a_6, a_7))$ and $((b_0, b_1), (b_2, b_3), (b_4, b_5), (b_6, b_7))$, where

the a_i, b_i are 16-bit values, two of which are stored together in one 32-bit register. It then computes (c_0, c_1, c_2, c_3) such that

$$c_0 = (a_0 \cdot b_0) + (a_1 \cdot b_1), \quad c_1 = (a_2 \cdot b_2) + (a_3 \cdot b_3),$$
$$c_2 = (a_4 \cdot b_4) + (a_5 \cdot b_5), \quad c_3 = (a_6 \cdot b_6) + (a_7 \cdot b_7).$$

I.e. it multiplies the 16-bit integers stored in a 32-bit word and adds these results. Hence, the results are stored as 32-bit entries. This restriction is not a problem since we expect that the inner-products of the list vectors get smaller while the algorithm progresses. We always double-check the inner-product with a regular routine without any size restrictions to catch false-positives: e.g. in case the inner-product of a candidate overflows. This does not increase the computational time noticeably since a reduction of a list vector happens only very infrequently (relative to the total number of computed inner-products).

For each of the pairs $(\mathbf{x}, \mathbf{y})$ we store

$$\mathbf{X} = (x_0, x_1, x_2, x_3, x_0, x_1, x_2, x_3),$$
$$\mathbf{Y}_1 = (y_0, -y_3, -y_2, -y_1, y_1, y_0, -y_3, -y_2),$$
$$\mathbf{Y}_2 = (y_2, y_1, y_0, -y_3, y_3, y_2, y_1, y_0).$$

in an SSE register. Now we obtain $\mathbf{z}$ by computing the "multiply and add packed integers" on the pairs $(\mathbf{X}, \mathbf{Y}_1)$ and $(\mathbf{X}, \mathbf{Y}_2)$ after performing the appropriate unpack routines. From this point on, the DFT is stored in SSE registers with 32 bits per entry. Note that the storage of the DFT is asymmetric for $\mathbf{x}$ and $\mathbf{y}$. In our implementation, we generally store the DFT of the sampled vectors in the form of $\mathbf{X}$ and the DFT's of list vectors in the form of $\mathbf{Y}_1$ and $\mathbf{Y}_2$. This way, we always have one representation of each type when comparing sample vectors to the list. The third step is implemented along the lines of [29, Exercise 4.6.4.59]. Negacyclic rotations are performed by using appropriate left and right shifts combined with additions and subtractions.

4.4 A Note on NTRU Lattices

The NTRU cryptosystem [26], published in 1998 long before the appearance of schemes based on R-LWE, uses a similar ring structure. The original NTRU setting works with the ring $R = \mathbb{Z}[X]/(X^n - 1)$. An ideal in R containing a also contains $X^i \cdot a(X)$. The corresponding vectors are cyclic rotations of $\mathbf{a}$ and this time there is no sign flip, i.e. $X^n \cdot \mathbf{a} = \mathbf{a}$ and the other properties in Lemma 3 (in Appendix B) hold in this case as well. The analog of Lemma 2 holds accordingly: Let $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^n$ with corresponding $a, b \in R$ and let $c(X) = a(X) \cdot (X \cdot b^{(R)}(X))$ mod $(X^n - 1)$, then $c_i = \langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$ for $i \in \{0, \ldots, n-1\}$. This means that one can also obtain n scalar products for the price of a single ring product.

Multiplication in R corresponds to the standard cyclic convolution of vectors, and can therefore be carried out with an FFT algorithm. If n is chosen to be a power of 2, we can use Nussbaumer's algorithm for cyclic convolution which recursively calls cyclic and negacyclic convolutions as described above. An implementation using SSE instructions can be done similarly to what we have described in Section 4.3. But in light of Gentry's attack [20] on composite degrees, the parameter n is usually chosen to be prime. In this case, the work by Rader [46] shows that the DFT can still be computed in $\mathcal{O}(n \ln n)$ digit operations. For example, the first approach described by Rader is to separate the computation of the 0-th coefficient from the others and transform the relevant sequences into sequences of length

Table 1. Results in seconds and scaling of our parallel Gauss sieve variant for different dimensions on BlueCrystal Phase 2 . For a varying number of cores C we state the speedup S with respect to the 8-core setting and the efficiency $E = S \cdot 8/C$.

C	80				88				96			
	t_{GH}	t_{Coll}	S	E	t_{GH}	t_{Coll}	S	E	t_{GH}	t_{Coll}	S	E
8	1918	3010	-	-	28961	45106	-	-	506841	654483	-	-
16	883	1592	1.9	.95	16794	23994	1.9	.95	252246	336288	2.0	1.0
32	533	820	3.7	.93	10445	14176	3.2	.80	125907	167790	3.9	.98
64	248	455	6.6	.83	4259	6037	7.5	.94	75676	101303	6.5	.81
96					3638	4984	9.1	.76	46774	70740	9.3	.77
128					2292	3392	13.2	.83	41196	54907	11.9	.74
160					2267	3206	14.1	.71	27864	36048	18.2	.91
192					2276	3104	14.5	.60	27038	35179	18.6	.78
224					1960	2849	15.8	.56	24873	33115	19.8	.71
256					1980	2726	16.5	.51	23442	30374	21.5	.67

$n-1$. If $n-1$ is highly composite or even a power of 2, the DFT can then be computed by convolutions of length $n-1$ on a permutation of the original sequence and a sequence of corresponding roots of unity. If $n-1$ itself has large prime factors, a zero-padding approach to some dimension $n' > n$ might be more suitable. A different approach is using discrete weighted transforms as outlined by Crandall and Fagin [14].

5 Experimental results

In order to assess the viability of the algorithmic techniques presented in this work, we created an implementation which allows to be executed concurrently using the message passing interface standard [38]. In this section, we summarize the experiments we ran and discuss the results. First, we discuss experiments with our new variant of the parallel Gauss sieve algorithm (see Section 3.1) in order to establish how it scales when increasing the number of nodes. Next, we consider the ideal lattice variant using the FFT approach (see Section 4) and compare this to an ideal lattice variant using rotations only as well as to the regular (non-ideal) algorithm. All experiments were run on one of two compute clusters, the BlueCrystal Phase 2 cluster and the BlueCrystal Phase 3 cluster of the Advanced Computing Research Centre at the University of Bristol.

The nodes in BlueCrystal Phase 2 are equipped with two 2.8 GHz quad-core Intel Harpertown E5462 processors with 8 GB RAM per node (1 GB per core). They are connected via a QLogic Infinipath high-speed network. The nodes in BlueCrystal Phase 3 each have two 2.6 GHz Intel Xeon E5-2670 with 16 cores and 64 GB RAM per node (4 GB per core). We had to use two clusters because the more modern BlueCrystal Phase 3 is dedicated to running larger and more compute intensive jobs and thus has a much lower availability than BlueCrystal Phase 2.One benefit of using the latter for experiments on scalability is that it has fewer cores per node, and therefore demonstrates the need and the impact of communication between the different threads in our algorithm more quickly than the former. Recently there have been some heuristic improvements to the (serial) GaussSieve [17]. These serial improvements should be fully applicable to our parallel (non-ideal) variant, whereas only the improvements to the sampler can be combined with our DFT-based approach for ideal lattices.

The New Parallel Gauss Sieve Variant. To benchmark our modified parallel Gauss sieve variant, we ran experiments on BlueCrystal Phase 2 using Goldstein-Mayer lattices from the

SVP challenge [49] of dimensions 80, 88, and 96. Similar to Ishiguro et al. [27], we also used BKZ [50] with block size 30 (as implemented in the fplll library [12]) to preprocess the basis before applying the algorithm. Note that these results are not comparable to Ishiguro et al., due to the different computer architecture. We repeated these experiments using a varying number C of cores, and measured the time $t_{GH}(C)$ in seconds that it took for these C cores to find a vector with norm smaller than the desired bound, i.e. less than 1.05 times the Gaussian heuristic, as well as the time $t_{\text{Coll}}(C)$ it took to generate enough collisions to satisfy the termination condition. As the baseline we use 8 cores, as a single node in the cluster contains 8 cores, which corresponds to the "shared memory" case. For each higher number of cores, we computed the *speedup* $S = t_{\text{Coll}}(C)/t_{\text{Coll}}(8)$ and the *efficiency* $E = S \cdot 8/C$. Consider the dimension 96 results from Table 1. When using four nodes (32 cores) the communication overhead between the nodes is relatively low: we see a factor 3.9 reduction in wall-clock time compared to the single-node setting, while we can hope for at most a factor four speed-up. This communication overhead becomes more prominent when using much more compute nodes. When running on 256 cores we still get a factor 22.5 reduction in wall-clock time while we can hope for a maximum factor of 32.

The results from [27] are obtained when finding short vectors using their version of the parallel Gauss sieve algorithm for random and ideal lattices on the AmazonEC2 cluster. One "instance" in this cluster contains two Intel Xeon E5-2670 processors (containing two times eight cores and which supports running 32 threads in parallel). Mariano et al. [37] use the same CPU-chip model, allowing for an easier comparison. To this end, we performed single-machine experiments on the BlueCrystal Phase 3 cluster, which also has the same CPU-chip model as the AmazonEC2 cluster used in [27]. Note that Mariano et al. use a different BKZ block size of 34 to pre-process the basis, which means comparison is not completely straightforward, since they start with a set of shorter vectors. Ishiguro et al. require 0.9 hours to find a short vector in a random 80-dimensional lattice when running on a single AmazonEC2 instance, whereas Mariano et al. require 2896 seconds or about 0.8 hours. Our algorithm requires only 1201 seconds, or 0.33 hours, which is a factor three improvement over Ishiguro et al. and at least a factor two improvement over Mariano et al. Furthermore, Ishiguro et al. report a run-time of 200 instance hours for finding a short vector in a 96-dimensional random lattice, whereas Mariano et al. do not tackle this dimension. Our variant of the parallel Gauss sieve algorithm can do this without shared memory in 61.4 hours, which is again a speed-up of at least a factor three. These performance numbers highlight the potential of our approach.

Recently, Laarhoven [30] proposed a similar sieving algorithm that uses locality-sensitive hashing to achieve better asymptotic run times under heuristic assumptions. Mariano et al. [36] implement a parallel version using a similar lock-free approach as for the parallel Gauss sieve. Since our implementation appears to give better results in the Gauss sieve case, this suggests that it is worth trying a similar approach for the Hash sieve algorithm and comparing the results.

The Ideal Lattice Setting. To benchmark the ideal Gauss sieve variant, we ran experiments on BlueCrystal Phase 2 using a power of two cyclotomic lattice of dimension 128 from the ideal lattice challenge [44]. We used BKZ with block size 30 to preprocess the basis, which is consistent with the work of Ishiguro et al. [27]. We aim to measure the difference between our regular parallel variant using no ideal-specific techniques (see Section 3.1), a version which additionally uses information about the rotations (see the function `ReduceRot` in Algorithm 2 and the properties in Lemma 3) and a version which uses our FFT approach using SSE

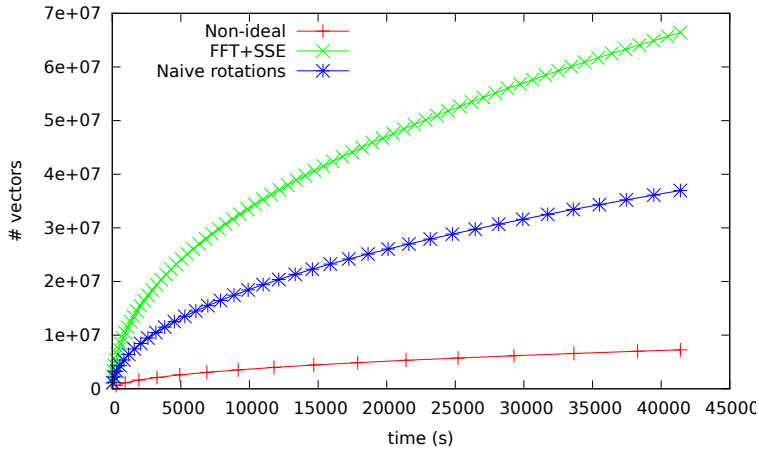

Fig. 2. The number of vectors in the list against the running time in a power of two cyclotomic ideal lattice of dimension 128 for three parallel Gauss sieve variants.

instructions (see Section 4.3). While fixing the number of compute cores to 64, we measure how many vectors are in the (global) list after a certain time. This characteristic is useful since the size of list provides information on how far the algorithm has proceeded.

Figure 2 shows the results of these experiments. Note that for the versions that use ideal-specific techniques, the "list" only stores one vector for each 128 vectors in the system because all the rotations of vectors are taken into account as well. The number of vectors printed in the graph is the complete number of vectors in the system including rotations, i.e., the total number of vectors that are all pairwise Gauss reduced. Figure 2 clearly shows the advantage of the approaches which use the ideal structure over the regular algorithm. After eleven and a half hours (41 400 seconds) the version using only rotations had completed 19 428 rounds, and had 288 783 vectors in its system, representing 288 783·128 vectors in total, whereas the version using FFT achieved the same in about three hours and twenty minutes, or 12 000 seconds, which corresponds to a speed-up of a factor 3.45. Finally, we applied our FFT implementation to the same SVP ideal lattice challenge of dimension 128 [44] that Ishiguro et al. [27] solved. We again used BlueCrystal Phase 3, with central processing units identical to the ones used in [27]. Again, we used BKZ with block size 30 to preprocess the basis obtained from the ideal lattice challenge. We ran our software on 64 nodes (1024 cores) and found a short vector after 750 478 seconds wall-clock time (8.69 days). The short vector found is a rotation of the short vector presented in [27]. In [27], this short vector was obtained after computing 14.88 days on 1344 cores. Our computation required 24.4 core years, is thus more than twice as fast as the computation carried out in [27], and shows the practical advantages of our approach and implementation on conventional compute clusters.

6 Discussion

What should the reader take away from these results? Lattice cryptanalysis is a complex subject, with different types of algorithms that are combined to find short vectors in lattices. Some works even claim that LWE-type problems are asymptotically best solved using non-lattice algorithms [5]. The most recent results of the SVP challenge [49] suggest that in currently tractable dimensions, a combination of BKZ and (some variant of) enumeration

appears to give the best results. However, it is not possible to reproduce and verify most of those results, because too many details of the experiments are not public: including, and most importantly, in most cases the source code used. This makes comparing the different algorithms and their results difficult.

In this paper, we have shown that for sieving algorithms such as the Gauss sieve, there are additional techniques to speed up the search for short vectors in cyclic and negacyclic lattices. Given the current performance details by various results on the SVP challenge [49], it seems that our speed-ups are not enough to defeat other SVP-solvers such as enumeration in tractable dimensions. However, cryptographic applications are not designed to utilize such relatively small dimensions, and the asymptotic run times of the various algorithms for solving the shortest vector problem show that sieving-based algorithms will eventually become faster than enumeration based approaches.

But even then, an approximate solution often suffices for cryptographic applications and thus it is currently hard to justify that sieving algorithms are practical in this setting. Still, we feel that examining these techniques and their effect are worthwhile, as it is unknown how the landscape may change in the future.

Acknowledgements

This work has been supported in part by the European Union's H2020 Programme under grant agreement number ICT-644209 and in part by EPSRC via grant EP/I03126X. We thank Nigel Smart for his comments on an earlier version of the paper and Thorsten Kleinjung for useful discussion.

References

1. D. Aggarwal, D. Dadush, O. Regev, and N. Stephens-Davidowitz. Solving the shortest vector problem in 2^n time via discrete Gaussian sampling. *CoRR*, abs/1412.7994, 2014.
2. M. Ajtai. Generating hard instances of lattice problems (extended abstract). In G. L. Miller, editor, *STOC*, pages 99–108. ACM, 1996.
3. M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In F. T. Leighton and P. W. Shor, editors, *STOC*, pages 284–293. ACM, 1997.
4. M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In J. S. Vitter, P. G. Spirakis, and M. Yannakakis, editors, *STOC*, pages 601–610. ACM, 2001.
5. M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. `http://eprint.iacr.org/`.
6. R. E. Bansarkhani and J. Buchmann. Improvement and efficient implementation of a lattice-based signature scheme. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 48–67. Springer, 2013.
7. A. Becker, N. Gama, and A. Joux. A sieve algorithm based on overlattices. *LMS Journal of Computation and Mathematics*, 17:49–70, 1 2014.
8. J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. Cryptology ePrint Archive, Report 2014/599, 2014. `http://eprint.iacr.org/599`.
9. J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In M. Stam, editor, *IMA Cryptography and Coding*, volume 8308 of *LNCS*, pages 45–64. Springer, 2013.
10. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In S. Goldwasser, editor, *Innovations in Theoretical Computer Science*, pages 309–325. ACM, 2012.

11. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In P. Rogaway, editor, *CRYPTO*, volume 6841 of *LNCS*, pages 505–524. Springer, 2011.
12. D. Cadé, X. Pujol, and D. Stehlé. fplll library, version 4.0.4. Available at `http://perso.ens-lyon.fr/damien.stehle/fplll/`, 2013.
13. Y. Chen and P. Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 1–20, 2011.
14. R. Crandall and B. Fagin. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation*, 62(205):305–324, 1994.
15. R. Crandall and C. Pomerance. *Prime numbers: a computational perspective*. Springer, second edition, 2005.
16. U. Fincke and M. Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44(170):pp. 463–471, 1985.
17. R. Fitzpatrick, C. Bischof, J. Buchmann, O. Dagdelen, F. Gopfert, A. Mariano, and B.-Y. Yang. Tuning GaussSieve for speed. Cryptology ePrint Archive, Report 2014/788, 2014. `http://eprint.iacr.org/788`.
18. N. Gama, P. Q. Nguyen, and O. Regev. Lattice enumeration using extreme pruning. In H. Gilbert, editor, *EUROCRYPT*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010.
19. S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *LNCS*, pages 1–17. Springer, 2013.
20. C. Gentry. Key recovery and message attacks on NTRU-composite. In B. Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *LNCS*, pages 182–194. Springer, 2001.
21. C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
22. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *LNCS*, pages 850–867. Springer, 2012.
23. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, pages 197–206. ACM, 2008.
24. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In E. Prouff and P. Schaumont, editors, *CHES*, volume 7428 of *LNCS*, pages 512–529. Springer, 2012.
25. T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe. Software speed records for lattice-based signatures. In P. Gaborit, editor, *Post-Quantum Cryptography*, volume 7932 of *LNCS*, pages 67–82. Springer-Verlag, 2013.
26. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. Buhler, editor, *ANTS-III*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.
27. T. Ishiguro, S. Kiyomoto, Y. Miyake, and T. Takagi. Parallel Gauss sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice. In H. Krawczyk, editor, *PKC*, volume 8383 of *LNCS*, pages 411–428. Springer, 2014.
28. R. Kannan. Improved algorithms for integer programming and related lattice problems. In *STOC*, pages 193–206. ACM, 1983.
29. D. E. Knuth. *Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 3rd edition, 1997.
30. T. Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. Cryptology ePrint Archive, Report 2014/744, 2014. `http://eprint.iacr.org/`.
31. A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.
32. T. Lepoint and M. Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. In D. Pointcheval and D. Vergnaud, editors, *AFRICACRYPT*, volume 8469 of *LNCS*, pages 318–335. Springer, 2014.
33. M. Liu, X. Wang, G. Xu, and X. Zheng. Shortest lattice vectors in the presence of gaps. Cryptology ePrint Archive, Report 2011/139, 2011. `http://eprint.iacr.org/`.
34. V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: A modest proposal for FFT hashing. In K. Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 54–72. Springer, 2008.
35. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *EUROCRYPT*, volume 6110 of *LNCS*, pages 1–23. Springer, 2010.
36. A. Mariano, T. Laarhoven, and C. Bischof. Parallel (probable) lock-free hashsieve: a practical sieving algorithm for the svp. Cryptology ePrint Archive, Report 2015/041, 2015. `http://eprint.iacr.org/`.

37. A. Mariano, S. Timnat, and C. Bischof. Lock-free GaussSieve for linear speedups in parallel high performance svp calculation. Cryptology ePrint Archive, Report 2014/775, 2014. http://eprint.iacr.org/775.
38. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0. Standard 3, Message Passing Interface Forum, 2012.
39. D. Micciancio and O. Regev. Lattice-based cryptography. In D. J. Bernstein and J. Buchmann, editors, *Post-quantum Cryptography*. Springer, 2008.
40. D. Micciancio and P. Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. In L. J. Schulman, editor, *STOC*, pages 351–358. ACM, 2010.
41. B. Milde and M. Schneider. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In V. Malyshkin, editor, *PaCT*, volume 6873 of *LNCS*, pages 452–458. Springer, 2011.
42. P. Q. Nguyen and T. Vidick. Sieve algorithms for the shortest vector problem are practical. *J. Mathematical Cryptology*, 2(2):181–207, 2008.
43. H. J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(2):205–215, 1980.
44. T. Plantard and M. Schneider. Ideal lattice challenge. http://latticechallenge.org/ideallattice-challenge/index.php, 2012.
45. T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 68–85. Springer, 2013.
46. C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.
47. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In H. N. Gabow and R. Fagin, editors, *STOC*, pages 84–93. ACM, 2005.
48. M. Schneider. Sieving for shortest vectors in ideal lattices. In A. Youssef, A. Nitaj, and A. E. Hassanien, editors, *AFRICACRYPT*, volume 7918 of *LNCS*, pages 375–391. Springer, 2013.
49. M. Schneider and N. Gama. SVP challenge. http://latticechallenge.org/svp-challenge/index.php, 2010.
50. C. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(1-3):181–199, 1994.
51. D. Stehlé and R. Steinfeld. Making ntru as secure as worst-case problems over ideal lattices. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *LNCS*, pages 27–47. Springer, 2011.
52. J. van de Pol and N. P. Smart. Estimating key sizes for high dimensional lattice-based systems. In M. Stam, editor, *IMA Cryptography and Coding*, volume 8308 of *LNCS*, pages 290–303. Springer, 2013.
53. P. Voulgaris. gsieve library. Available at http://cseweb.ucsd.edu/~pvoulgar/impl.html, 2011.
54. P. Voulgaris and D. Micciancio. Faster exponential time algorithms for the shortest vector problem. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:65, 2009.
55. X. Wang, M. Liu, C. Tian, and J. Bi. Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, and D. S. Wong, editors, *ASIACCS*, pages 1–9. ACM, 2011.
56. F. Zhang, Y. Pan, and G. Hu. A three-level sieve algorithm for the shortest vector problem. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 29–47. Springer, 2013.

A Gauss sieve

Algorithm 3 outlines the Gauss sieve algorithm as described by Micciancio and Voulgaris [54] including some modifications by Voulgaris that were also mentioned in [41], related to the reduction condition (e.g. the original condition $\|\mathbf{u}\pm\mathbf{w}\| > \|\mathbf{w}\|$ is replaced by $2\cdot|\langle\mathbf{u},\mathbf{w}\rangle| > \langle\mathbf{w},\mathbf{w}\rangle$ since this can be computed more efficiently when the squared norms of the vectors are stored with the vectors). The structure has been slightly adapted to reflect current implementations in practice (e.g. [53] uses a linked list to represent the list of pairwise Gauss reduced vectors). Note that Algorithm 3 makes use of a sorted linked list L. In practice one could modify it to use an unsorted array instead. We choose to display the algorithm with a sorted linked list to ease the explanation.

Algorithm 3 Gauss sieve [54]. Given a basis $\mathbf{B}$ and bounds $\mu, c > 0$, return a short vector $\mathbf{v}$. The algorithm terminates when $\mathbf{v}$, $\|\mathbf{v}\| \leq \mu$, is found, or when the number of collisions is at least c. The i-th vector in the sorted linked-list L is denoted by ℓ_i, the cardinality of L by $\#L$, and S is a stack.

```
 1: function GaussSieve(B, μ, c)
 2:     L ← {0}, S ← {}, K ← 0
 3:     while K < c do
 4:         v_new ← Sample(S), i ← 0
 5:         while i < #L and ‖ℓ_i‖ ≤ ‖v_new‖ do
 6:             if Reduce(v_new, ℓ_i) then
 7:                 i ← −1
 8:                 if ‖v_new‖ == 0 then K ← K + 1, v_new ← Sample(S)
 9:                 if ‖v_new‖ ≤ μ then return v_new
10:             i ← i + 1
11:         if ‖v_new‖ > 0 then Insert v_new into L at position i, i ← i + 1
12:         while i < #L and ‖ℓ_i‖ ≥ ‖v_new‖ do
13:             if Reduce(ℓ_i, v_new) then
14:                 if ‖ℓ_i‖ == 0 then Remove ℓ_i from L, i ← i − 1, K ← K + 1
15:                 else
16:                     if ‖ℓ_i‖ ≤ μ then return ℓ_i
17:                     Move ℓ_i from L to S
18:             i ← i + 1
19:     return the smallest ℓ_i from L

20: function Sample(S)
21:     if S is empty then
22:         Sample v ≠ 0 (GPV [23])
23:     else
24:         Pop v from S
25:     return v

26: function Reduce(u, w)
27:     if 2 · |⟨u, w⟩| > ⟨w, w⟩ then
28:         u ← u − ⌈⟨u,w⟩/⟨w,w⟩⌋ w
29:         return true
30:     return false
```

B Identities for negacyclic rotations and proof of Lemma 1

In the following lemma we collect a few useful identities for negacyclic rotations, which can be easily proved by explicitly writing them out.

Lemma 3. *Let $a, b \in R = \mathbb{Z}[X]/(X^n + 1)$ for n a power of 2, let $\mathbf{a}, \mathbf{b}$ be their coefficient vectors, and let $i, j \in \mathbb{Z}$. Then with notation as above, we have:*

$$X^i \cdot (X^j \cdot \mathbf{a}) = X^{i+j} \cdot \mathbf{a}, \quad X^i \cdot (\mathbf{a} + \mathbf{b}) = X^i \cdot \mathbf{a} + X^i \cdot \mathbf{b}, \quad X^n \cdot \mathbf{a} = -\mathbf{a},$$
$$\langle X^i \cdot \mathbf{a}, X^i \cdot \mathbf{b} \rangle = \langle \mathbf{a}, \mathbf{b} \rangle, \quad \langle X^i \cdot \mathbf{a}, X^j \cdot \mathbf{b} \rangle = \langle \mathbf{a}, -X^{n-i+j}\mathbf{b} \rangle.$$

Using the above identities, we can now prove Lemma 1 in Section 4.

Proof of Lemma 1. The properties in Lemma 3 show that $|\langle X^i \cdot \mathbf{a}, X^j \cdot \mathbf{b} \rangle| = |\langle \mathbf{a}, X^{n-i+j}\mathbf{b} \rangle| = |\langle \mathbf{a}, X^\ell \mathbf{b} \rangle|$, where $\ell = (n - i + j) \mod n$, i.e. $0 \leq \ell < n$, as well as $\langle X^i \cdot \mathbf{a}, X^i \cdot \mathbf{a} \rangle = \langle \mathbf{a}, \mathbf{a} \rangle$ and $\langle X^j \cdot \mathbf{b}, X^j \cdot \mathbf{b} \rangle = \langle \mathbf{b}, \mathbf{b} \rangle$. Hence the lemma follows. □

C Details on Nussbaumer's negacyclic convolution algorithm

The key observation is that the ring extension $R = \mathbb{Z}[X]/(X^n + 1)$ of $\mathbb{Z}$ can be decomposed into two extensions as follows. Let $n = 2^k = s \cdot r$ where $s \mid r$, e.g. $s = 2^{\lfloor k/2 \rfloor}$ and $r = 2^{\lceil k/2 \rceil}$. Then $R \cong S = T[X]/(X^s - Z)$, where $T = \mathbb{Z}[Z]/(Z^r + 1)$.

Note that $Z^{r/s}$ is an s-th root of -1 in T and $X^s = Z$ in S. This isomorphism is expressed simply by a re-ordering of coefficients. An element $a(X) = \sum_{i=0}^{n-1} a_{i-1} X^i \in R$ can be written as an element of S by replacing all powers X^s by Z, which gives

$$a(X) = \sum_{i=0}^{s-1} A_i(Z) X^i, \text{ with } A_i(Z) = a_i + a_{i+s} Z + \cdots + a_{i+s(r-1)} Z^{r-1} \in T.$$

Nussbaumer's algorithm for computing the negacyclic convolution of two polynomials $a, b \in R$ begins by interpreting the coefficients of a, b as the sequences of elements $A_0, \ldots, A_{s-1} \in T$ and $B_0, \ldots, B_{s-1} \in T$ as above. The result $a(X)b(X) \mod (X^n + 1)$ can then be computed as a cyclic convolution of the sequences $A = (A_0, A_1, \ldots A_{s-1}, 0, \ldots, 0)$ and $B = (B_0, B_1, \ldots B_{s-1}, 0, \ldots, 0)$ of length $2s$. The products of the A_i with the B_j for $1 \leq i, j < s$ are computed in the ring T and are thus negacyclic convolutions of sequences of length r.

The cyclic convolution can be computed by a symbolic fast Fourier transform algorithm (FFT) using the $2s$-th root of unity $Z^{r/s}$. This means that one computes the discrete Fourier transforms (DFT) $\tilde{A} = (\tilde{A}_0, \ldots, \tilde{A}_{2s-1}), \tilde{B} = (\tilde{B}_0, \ldots, \tilde{B}_{2s-1})$ of the sequences of length $2s$ above, carries out the coefficient-wise multiplications $\tilde{C}_i = \tilde{A}_i \tilde{B}_i$, $0 \leq i < 2s$ by negacyclic convolutions of length r in T, and computes the inverse DFT of $\tilde{C} = (\tilde{C}_0, \ldots, \tilde{C}_{2s-1})$ (using $Z^{r/s}$) to obtain the result $C = (C_0, \ldots, C_{2s-1})$. The final result $c(X) = a(X)b(X) \mod (X^n + 1)$ is given by

$$c(X) = \sum_{i=0}^{2s-1} C_i(X^s) X^i \mod (X^n + 1) \in R$$

and can be computed by s additions and subtractions in T.