Nearest Planes in Practice

Christian Bischof[1], Johannes Buchmann[1], Özgür Dagdelen[1], Robert Fitzpatrick[2], Florian Göpfert[1], and Artur Mariano[1]

[1] Technische Universität Darmstadt, Germany
[2] Academia Sinica, IIS, Taiwan

Abstract. The learning with errors (LWE) problem is one of the most attractive problems that lattice-based cryptosystems base their security on. Thus, assessing the hardness in theory and practice is of prime importance. Series of work investigated the hardness of LWE from a theoretical point of view. However, it is quite common that in practice one can solve lattice problems much faster than theoretical estimates predict.
The most promising approach to solve LWE is the decoding method, which converts an LWE instance to an instance of the closest vector problem (CVP). The latter instance can then be solved by a CVP solver. In this work, we investigate how the nearest planes algorithm proposed by Lindner and Peikert (CT-RSA 2011) performs in practice. This algorithm improves an algorithm by Babai, and is a state-of-the-art CVP solver.
We present the first parallel version of the nearest planes algorithm. Our implementation achieves speedup factors of more than 11x on a machine with four CPU-chips totaling 16 cores. In fact, to the best of our knowledge, there is not even a single parallel implementation publicly available of any LWE solver so far. We also compare our results with heuristics on the running time of a single nearest planes run claimed by Lindner and Peikert and subsequently used by others for runtime estimations.

Keywords cryptanalysis, lattices, decoding attack, nearest planes, implementation

1 Introduction

The Learning with Errors (LWE) problem has attracted a considerable amount of attention since its introduction by Regev [26]. Along with its 'sister problem', the Short Integer Solutions (SIS) problem, LWE enjoys currently unique security guarantees, in effect asserting that 'weak' instances do not exist. Additional reasons for the current popularity of LWE and its more efficient variant (Ring-LWE) lie in their asymptotic efficiency, conjectured invulnerability to solution by large-scale quantum computers, the relatively 'lightweight' atomic operations required for their implementation and, lastly, the remarkable flexibility of LWE as a basis for cryptographic constructions. In consequence, a wide variety of schemes based on LWE have been proposed in recent years, ranging from basic public

key encryption [19, 22, 25, 26] and signature schemes [6, 11, 14, 21] to advanced schemes like fully and somewhat homomorphic encryption, e.g., [8, 9, 12].

In contrast to strong theoretical results, however, the hardness of concrete LWE instances (and of lattice problems in general) in practice is still a remarkably unexplored and, at times, bewildering area. Obviously, this comparative neglect of practical hardness considerations presents a (arguably the principal such) potential problem with respect to the practical adoption of lattice-based cryptography in the future.

Restricting our attention to LWE, there are essentially three approaches to solve LWE instances known at present. The indirect way of solving LWE is by reducing LWE to a unique Shortest Vector Problem (uSVP) instance, and solve this derived instance using an (approximate) SVP solver, such as LLL and BKZ (2.0). This approach is also called the embedding attack [13,17]. Dedicated algorithms for solving LWE such as the combinatorial BKW algorithm [7] and the *decoding algorithm* [19] have been subsequently proposed. Except for the BKW algorithm, all of these algorithms for LWE rely on strong lattice reduction (i.e., BKZ 2.0). We will not consider BKW in the following, since it requires exponentially many samples and is therefore not practical in realistic scenarios.

While there are a series of works analyzing the embedding attack and the BKW algorithm in practice [1–3, 5, 15, 23, 24], the practical behavior of the decoding algorithm is still unexplored. In this work, we endeavor to enlighten this area a little further by showing experiments with a parallel version of the nearest planes attack proposed by Lindner and Peikert [19] following the decoding approach.

1.1 Our Contribution

In [19], a brief discussion is given with regard to the parallelization of the nearest planes algorithm, however, this consisted of largely high-level heuristic observations with no practical experiments or detailed consideration of such being made (to the best of our knowledge).

Since the decoding attack is widely believed to be the currently optimal method of attacking LWE in practice, we believe that a concrete instantiation and concrete consideration of such issues is of significant importance. We present experimental and theoretical results with regard to the performance of the nearest planes algorithm for LWE, with an emphasis on the parallel implementation. This includes exhaustive experiments with a concrete parallel implementation of nearest planes that scales very well on multi-core machines. The results from our experiments are used as a basis to predict the running time of nearest planes on concrete LWE instances (here we follow the approach by Lindner and Peikert [19]). We compare the results with other attacks and show that nearest planes is in fact the most promising known attack (in practice and theory) on those LWE instances.

Our sequential implementation can find up to 2^9 close lattice vectors per second. Since the parallel version scales quite well, we can conclude that it should

be possible to find more than 2^{16} close lattice vectors per second, which is the bound given by Lindner and Peikert [19].

1.2 Related Work

Lindner and Peikert [19] proposed the nearest planes algorithm and showed (to some extent) how to simulate its performance. Albrecht et al. [2] evaluated the performance of the BKW algorithm on LWE instances. BKW is a combinatorial attack on LWE that is very suitable for parallelization, but only the sequential variant was considered in [2]. Another attack on LWE is the embedding approach by Kannan [17], the application of which was examined by Albrecht, Fitzpatrick and Göpfert [1], but there is no natural way to parallelize the implementation. To the best of our knowledge, there are no other studies on the parallelization of an LWE solver.

Liu and Nguyen [20] presented recently a very interesting work related to the nearest planes algorithm. They show that nearest planes can be viewed as an instance of enumeration (more commonly studied with regard to solving the exact shortest vector problem) and apply known improved variants of enumeration to nearest planes to obtain theoretical and practical improvements over [19]. In particular, those improvements are randomization and pruning. The idea of randomization is to apply the attack many times with parameters that provide only a small success probability with random bases. Applying this approach with a parallel implementation of nearest plane is easily possible. The idea of pruning is to cut off parts of the search trees that contribute significantly to the running time but only slightly to the success probability. This leads to unbalanced search trees and makes parallelization more difficult, but not impossible as the parallel implementation of the pruned enumeration by Kuo et al. [18] shows (see also [10, 16]).

While Liu and Nguyen show that their approach outperforms the nearest planes as proposed in [19], we observe the following. In a nutshell, the goal of [20] is to minimize the number of “false positives”, i.e the number of vectors returned by nearest planes that are not close to the target. Our goal, however, is to maximize the number of vectors we can find by calculating many nodes in parallel. Hence, the approaches are complementary and a combination of both approaches (for parallelization) would be very promising for future work.

2 Preliminaries

2.1 Lattice Background

For an integer n, we define $[n] = \{1, 2, \dots, n\}$. We denote vectors by bold lower-case letters and matrices by bold upper-case letters.

A lattice Λ is a discrete subgroup of the space $\mathbb{R}^m$. Lattices are represented by linearly independent vectors $\mathbf{b}_1, \dots, \mathbf{b}_n \in \mathbb{R}^m$, where n is called the dimension of the lattice. If $n = m$, the respective lattice has full rank. We call a set of

vectors $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ a basis of a lattice $\Lambda(\mathbf{B})$ if the vectors are linearly independent and n is equal to the rank of the lattice. The lattice $\Lambda(\mathbf{B})$ is defined by all integer combinations of elements of $\mathbf{B}$, i.e.,

$$\Lambda(\mathbf{B}) = \left\{ \mathbf{x} \in \mathbb{R}^m \mid \exists \alpha_1, \ldots, \alpha_n \in \mathbb{Z} : \mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{b}_i \right\} .$$

We are particularly interested in modular integer lattices. These are also the lattices one considers when solving LWE instances. A modular (or q-ary) lattice, for a given $q \in \mathbb{N}$, is a full-ranked lattice Λ such that $q\mathbb{Z}^m \subseteq \Lambda \subseteq \mathbb{Z}^m$. The determinant of a full-ranked lattice $\Lambda(\mathbf{A})$ is defined as $\det(\Lambda(\mathbf{A})) = \det(\mathbf{A})$. It is well known that the determinant of a lattice is well-defined (i.e. does not depend on the particular basis) and the definition can be generalized for lattices that are not full-ranked.

For a set of vectors $\mathbf{B}$, we write $\pi_{\mathrm{span}(\mathbf{B})}(\mathbf{t})$ for the projection of the vector $\mathbf{t}$ onto the span of the vectors of $\mathbf{B}$, i.e., $\pi_{\mathrm{span}(\mathbf{B})}(\mathbf{t}) = \mathbf{B}(\mathbf{B}^T\mathbf{B})^{-1}\mathbf{B}^T \cdot \mathbf{t}$.

The Gram-Schmidt orthogonalization $\widetilde{\mathbf{B}} = \{\widetilde{\mathbf{b}}_1, \ldots, \widetilde{\mathbf{b}}_n\}$ of a basis $\mathbf{B}$ is defined through $\widetilde{\mathbf{b}}_i = \mathbf{b}_i - \pi_{\mathrm{span}(\mathbf{b}_1,\ldots,\mathbf{b}_{i-1})}(\mathbf{b}_i)$ for $i \in [n]$. Note that the Gram-Schmidt basis is typically not a basis of the lattice. The fundamental parallelepiped of a basis B is given by

$$\mathcal{P}(\mathbf{B}) = \left\{ \mathbf{v} = \sum_{i=1}^{m} \alpha_i \mathbf{b}_i \mid \forall i \in [m] : 0 \leq \alpha_i < 1 \right\},$$

and the shifted fundamental parallelepiped by

$$\mathcal{P}_{1/2}(\mathbf{B}) = \left\{ \mathbf{v} = \sum_{i=1}^{m} \alpha_i \mathbf{b}_i \mid \forall i \in [m] : -\frac{1}{2} \leq \alpha_i < \frac{1}{2} \right\} .$$

Analogously, we can consider the fundamental parallelepiped (and shifted parallelepiped) determined by the Gram-Schmidt vectors of a given basis by replacing $\mathbf{b}_i$ in the above definitions with $\tilde{\mathbf{b}}_i$ – we denote these cases by $\mathcal{P}(\widetilde{\mathbf{B}})$ and $\mathcal{P}_{1/2}(\widetilde{\mathbf{B}})$, respectively. Note that, in these cases, the orthogonality of the basis vectors implies that $\mathcal{P}(\widetilde{\mathbf{B}})$ and $\mathcal{P}_{1/2}(\widetilde{\mathbf{B}})$ are n-dimensional rectangles.

The quality of a basis is typically measured with the Hermite delta δ. A basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ of an n-dimensional lattice Λ has Hermite delta δ if $\|\mathbf{b}_1\| = \delta^m \det(\Lambda)^{1/n}$. Below we recall the learning with errors problem formally whose hardness we investigate in this work.

Definition 1 ((Search) LWE Problem). *Let n, q be positive integers, χ be a probability distribution on $\mathbb{Z}_q$ and $\mathbf{s}$ be a secret vector following the uniform distribution on $\mathbb{Z}_q^n$. We denote by $L_{\mathbf{s},\chi}^{(n)}$ the probability distribution on $\mathbb{Z}_q^n \times \mathbb{Z}_q$ obtained by choosing $\mathbf{a}$ from the uniform distribution on $\mathbb{Z}_q^n$, choosing e according to χ and returning $(\mathbf{a}, c) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$. Search-LWE is the problem of finding $\mathbf{s} \in \mathbb{Z}_q^n$ given pairs $(\mathbf{a}_i, c_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ sampled according to $L_{\mathbf{s},\chi}^{(n)}$.*

Naturally, we can extend this definition to 'Matrix-LWE' in which LWE samples (with a common secret vector) are concatenated to obtain a vector $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$ and we are again asked to recover $\mathbf{s}$. By adopting this view, we can now view the matrix $\mathbf{A}$ as determining a q-ary lattice with $\mathbf{A}\mathbf{s}$ being a lattice point and $\mathbf{t}$ being the 'noisy' lattice point, the recovery of which is required to solve the LWE instance.

2.2 Definitions on Parallel Computing

We now recap some concepts pertaining to parallel computing. *Threads* (of computation) are sequences of instructions that can be executed independently from one another. Variables that are accessed by all threads are said to be *shared* variables, while variables of which each thread has a private copy of are said to be *private* variables. A *task* is computational work that is assigned to a thread. For the sake of simplicity, we deal with tasks that are never preempted from one thread to be assigned to another. *Barriers* are synchronization points for threads. Threads are only released from a specific barrier when every thread in the system reaches it. A *parallel zone* denotes a region of the code that is executed by all the threads in the system. Threads are created at the beginning of the parallel zone and die at the end of the region. Finally, a *single zone* denotes a region of the code that is executed by a single, unspecified, thread.

3 The Decoding Attack

3.1 The Idea

Since LWE is essentially a closest vector problem instance (given a modular lattice L and a target vector $\mathbf{t}$, find the lattice vector that is closest to $\mathbf{t}$), one natural approach is to apply the well-known nearest plane algorithm (due to Babai [4]) to recover a lattice point relatively close to the 'noisy' target point. In short, the idea of nearest plane is to solve the problem by dealing with one dimension after the other. For every basis vector $\mathbf{b}_i$, it subtracts the integer multiple c_i of $\mathbf{b}_i$ that minimizes the distance to the hyperplane spanned by the basis vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ from the target vector and continues with the smaller basis $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ and the new target vector. At the end, it returns the sum of all vectors $c_i \mathbf{b}_i$, which is obviously a lattice vector.

Understanding why the result is (to some extent) close to the target vector requires more effort, but fortunately there is an easy geometric interpretation of the output: When called on an LWE instance $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$, the (polynomial-time) algorithm nearest plane returns the unique lattice point $\mathbf{v}$ such that $\mathbf{t} - \mathbf{v} = \mathbf{e}$ lies in the shifted Gram-Schmidt fundamental parallelepiped $\mathcal{P}_{1/2}(\widetilde{\mathbf{B}})$.

One phenomenon which arises when examining lattices bases is that the logarithms of the norms of the Gram-Schmidt vectors appear to decline linearly, this phenomenon being known as the 'Gram-Schmidt Log Assumption'. As noted in [19], this phenomenon also manifests in the case of the modular lattices arising

from LWE. If the last vectors in the Gram-Schmidt basis are too short, the 'search rectangle' $\mathcal{P}_{1/2}(\widetilde{\mathbf{B}})$ will be long and narrow and the returned point will in general be far from the actual closest lattice point.

One natural way to improve the success probability is to apply a basis reduction (typically BKZ) before running nearest plane. This will lead to "more orthogonal" basis vectors, which leads to a smaller gradient being for the log of norms of the Gram-Schmidt vectors and therefore to a search rectangle that is less narrow in the last dimension.

Another natural improvement and forming the crux of the Lindner-Peikert algorithm is to recurse on more than one plane at each step, i.e., instead of subtracting one multiple of the last basis vector, we subtract several, each leading to a vector close to the span of the other basis vectors. Each such vector value then leads to a further set of recursive calls as opposed to just one. Clearly, however, if we even deviate from the nearest plane algorithm by recursing on not just the closest plane but the closest and second-closest plane at each level, we obtain exponential complexity. In the nearest planes algorithm, the number of such branches at each level is specified by a vector $\mathbf{d}$, leading a generalization with the original nearest plane algorithm corresponding to $\mathbf{d} = (1, 1, \ldots, 1)$. Similarly to $\mathcal{P}_{1/2}(\widetilde{\mathbf{B}})$, the search rectangle of nearest plane, we define

$$\mathcal{P}^{\mathbf{d}}_{1/2}(\widetilde{\mathbf{B}}) = \left\{ \mathbf{v} = \sum_{i=1}^{m} \alpha_i \tilde{\mathbf{b}}_i \mid \forall i \in [m] : -\frac{d_i}{2} \leq \alpha_i < \frac{d_i}{2} \right\}$$

as the search rectangle of nearest planes.

To find the optimal choice of the vector $\mathbf{d}$, assuming the Gram-Schmidt Log assumption holds, we can observe that, to minimize the probability of the exact closest vector not being found through our projections, we should recurse on more planes when $\|\tilde{\mathbf{b}}_i\|$ is small and on fewer planes when $\|\tilde{\mathbf{b}}_i\|$ is large. In general, as observed in [19], the entries of $\mathbf{d}$ should be chosen to maximize $\min_i(d_i \cdot \|\tilde{\mathbf{b}}_i\|)$. Such issues are dealt with in the work of Lindner and Peikert, with natural optimal conditions being arrived at. However, since this work is concerned only with the nearest planes algorithm and not with obtaining the optimal distribution of time between the pre-processing and the main algorithm, we do not discuss such issues further.

3.2 Variants of Nearest Planes

Nearest planes traverses the tree as follows: for a given target vector $\mathbf{t}$, it calculates a new target vector $\mathbf{t}'$ and calls nearest planes with the new target vector (i.e. goes down the tree by one level). In every node, a part of the result is calculated. To get the final results, the algorithm goes up the tree again, combining the partial results of every level. In contrast to this, we have implemented a variant which has a slightly different workflow. Instead of going up and down on the tree in a depth-first manner, we go down and right in a breadth-first manner. This implies that we do not accumulate the target vector when we go

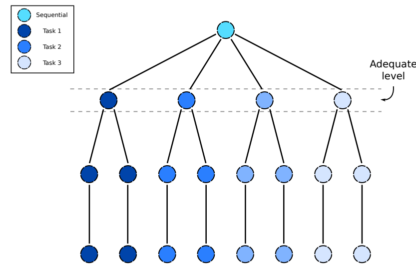


Fig. 1. Map of the algorithm's workflow on a tree, partitioned into tasks, for $\mathbf{d} = \{1, 2, 4\}$, for a number of threads ≥ 4.

up in the tree, but we pass the parts of the target vector to the lower nodes and accumulate while we go down on the tree. In the next section we describe in more detail our chosen variant for implementation.

4 Parallel Implementation

In this section we explain the mapping of the algorithm's workflow on a weighted tree and the parallelization of the traversal and computation of the tree.

4.1 Mapping of the Workflow on a Weighted Tree

The workflow of the algorithm can be viewed as a traversal of a tree, with $\prod_{i=n-k}^{n} \mathbf{d}_i$ is the number of nodes in level k. The values in $\mathbf{d}$ dictate the number of branches per level on a reversed order: position $\mathbf{d}_n$ indicates the number of branches on the first level, $\mathbf{d}_{n-1}$ indicates the number of branches on the second level, and so on.

Figure 1 shows a tree with an array $\mathbf{d} = \{1, 2, 4\}$. A new target vector is calculated for each node, on a certain level k, and used in the level $k + 1$ by its child nodes. The processing of vectors in a given level $k + 1$, after the execution of nodes in the level k, is a process referred to as *going down* in the tree.

As opposed to direct implementation of nearest planes [19], which does not need to carry vectors from one level of the tree to the other, our implementation hands error vectors from a given level k to its subsequent level $k + 1$. This is equivalent to process the tree in a depth-first manner, versus to process the tree in a breadth-first manner, in our implementation. This has a direct impact in data collection, further discussed in Section 4.3.

4.2 Approach

Our parallel implementation is based on creating a task for each branch of the tree, starting at some level, as seen in Figure 1 (tasks are in different colors).

Tasks are very well suited for the parallelization of this algorithm because, unlike other abstractions, such as threads, it is easy to specify parallel workload (branches of the tree). Our parallelization scheme is based on sequentially executing (and *going down*) a certain number of levels on the tree, until an *adequate level* is reached. Conceptually, an *adequate level* k is a level which satisfies:

1. The number of nodes and child nodes on the level provide enough computation to utilize the capacity of all running threads
2. The computation associated to the levels between 1 and k is not a significant part of the overall computation of the tree.

This means that the *adequate level* depends on the number of running threads and on the amount of computation required by the nodes on the levels that precede it. Then, once the *adequate level* has been reached, the implementation defines as many tasks as the number of nodes on the level. Each task entails the computation of each node on the *adequate level* and its child nodes. Tasks can then be executed in parallel, by unspecified computation units, without any need for synchronization. There are a couple of data structures and variables that need to be initialized accordingly. For example, each task receives its own target vector and has his own variable for len.

Each task traverses itself a tree, rooted by the node that is on its starting level. All tasks receive the value of the *adequate level*, so that they can calculate the number of child nodes that they have, by accessing the vector $\mathbf{d}$ accordingly.

4.3 Implementation

In our implementation, we calculate the *adequate level* as the first level on the tree that has at least as many nodes as the number of running threads. Since the *adequate level*, as it is defined by us, is straightforward to compute, we split the original loop into two different loops, as shown by steps 6 and 16 in Algorithm 1. From here on, we refer to these loops as loops `lp1` and `lp2`, respectively. Note that it is also trivial to compute the number of nodes #nodes in the *adequate level*.

In addition to this, our parallel implementation differs from the Lindner and Peikert implementation in two other ways. The first difference is that, instead of going through the tree in a recursive depth-first manner, we traverse it iteratively in a breadth-first manner. The second difference is that we do not add up multiples of the basis vectors to find a lattice vector. Instead, we update the target vector by subtracting a multiple of a basis vector. This leads to an error vector (i.e., a vector $\mathbf{e}$ such that the original target vector subtracted by $\mathbf{e}$ is a lattice vector), which can easily be used to calculate the desired close lattice vector.

This parallel execution is implemented with OpenMP. Our implementation has a parallel region, that creates as many threads as defined by the user. Inside the parallel region, a single region (region executed by one, unspecified, thread) embodies both loops `lp1` and `lp2`. These regions are represented in Algorithm 1,

Algorithm 1: Nearest Planes

Input: $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\} \subset \mathbb{R}^m$, $\mathbf{d} \in (\mathbb{Z}^+)^m$, $\mathbf{t} \in \mathbb{R}^m$, $al \in \mathbb{N}$, $\#nodes \in \mathbb{N}$

Output: All error vectors $\mathbf{e} \in \mathcal{P}^{\mathbf{d}}_{1/2}(\{\tilde{\mathbf{b}}_1, \ldots, \tilde{\mathbf{b}}_m\})$ such that $\mathbf{t} - \mathbf{e} \in \Lambda(\{\mathbf{b}_1, \ldots, \mathbf{b}_m\})$

```
1  begin
2    calculate Gram-Schmidt basis B~ = {b~_1, ..., b~_m};
3    OpenMP_parallel_region
4      OpenMP_single_region
5        len = 1;
6        for k = n; k ≥ n − al; k = k − 1 do
7          for i = 0; i < len; i = i + 1 do
8            Let {c_1, ..., c_{d_k}} ∈ Z^{d_k} be the distinct integers closest to ⟨b~_k, t⟩ / ⟨b~_k, b~_k⟩;
9            for j = 1; j ≤ d_k; j = j + 1 do
10             t*_{i·d_k+j} = t_i − c_j · b_k;
11           end
12         end
13         t = t*;
14         len = len · d_k;
15       end
16       for node = 0; node ≤ #nodes; node = node + 1 do
17         create_task
18           len = 1;
19           for k = n − al − 1; k ≥ 1; k = k − 1 do
20             for i = 0; i < len; i = i + 1 do
21               Let {c_1, ..., c_{d_k}} ∈ Z^{d_k} be the distinct integers closest to
                 ⟨b~_k, t⟩ / ⟨b~_k, b~_k⟩;
22               for j = 1; j ≤ d_k; j = j + 1 do
23                 t*^{node}_{i·d_k+j} = t_i − c_j · b_k;
24               end
25             end
26             t = t*;
27             len = len · d_k;
28           end
29         end
30       end
31     end
32   end
33   return t;
34 end
```

steps 3 and 4. While both loops are executed sequentially, by a given thread *t*, *t* creates #nodes tasks in `lp2`, each of which entailing the body of the `lp2` loop. As soon as tasks are created, they are assigned to one thread, which means that the issue of tasks is, very likely, overlapped with the execution of other tasks, by a different thread. The OpenMP runtime manages the task scheduling among the running tasks.

Workload balance. The *adequate level* is defined to be the first level on the tree that contains a number of nodes that is at least as big as the number of threads. This is because all the tasks are, computationally speaking, very well balanced (in terms of FLOPS they are, in fact, equal). This means that, as long as (1) the number of nodes in the *adequate level* is a multiple of the computing units, (2) the computing units compute the same number of tasks and (3) computing units are equally capable, the workload distribution is balanced.

Data collection. A vector of #nodes pointers is allocated outside of the parallel region, and defined as shared in the parallel region, which means that every thread has access to it. As tasks have an id (from 1 to #nodes), it is easy for threads to write in a different location, that will be available outside of the parallel region. Once the parallel region is finished, which means that every task is also finished (there is an implicit barrier at the end of the parallel region), the structure is accessed and the shortest (error) vector among all the stored vectors is chosen.

5 Results

	$m = 404$						$m = 517$					
	number of enumerations											
	2^{12}		2^{15}		2^{18}		2^{12}		2^{15}		2^{18}	
Threads	R	S	R	S	R	S	R	S	R	S	R	S
1	7.04	1.00	56.03	1.00	446.93	1.00	11.65	1.00	92.63	1.00	736.19	1.00
2	3.61	1.95	28.54	1.96	227.43	1.97	5.93	1.96	47.14	1.96	373.78	1.97
4	1.87	3.77	14.88	3.77	117.18	3.81	3.06	3.81	24.19	3.83	192.71	3.82
8	1.01	6.99	8.04	6.97	63.81	7.00	1.65	7.07	13.16	7.04	104.20	7.06
16	0.66	10.71	5.36	10.45	42.01	10.64	1.08	10.75	8.64	10.72	67.93	10.84

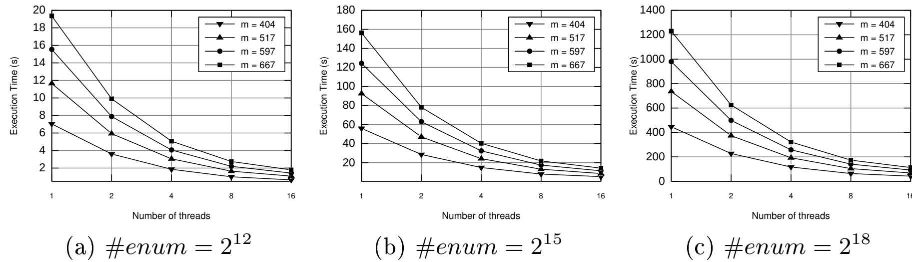
	$m = 597$						$m = 667$					
	number of enumerations											
	2^{12}		2^{15}		2^{18}		2^{12}		2^{15}		2^{18}	
Threads	R	S	R	S	R	S	R	S	R	S	R	S
1	15.54	1.00	124.31	1.00	979.78	1.00	19.36	1.00	156.34	1.00	1229.38	1.00
2	7.88	1.97	63.03	1.97	499.26	1.96	9.91	1.95	78.20	2.00	624.76	1.97
4	4.07	3.82	32.37	3.84	257.64	3.80	5.07	3.82	40.36	3.87	321.76	3.82
8	2.21	7.05	17.45	7.12	140.85	6.96	2.75	7.04	21.83	7.16	173.44	7.09
16	1.43	10.86	11.46	10.85	92.28	10.62	1.79	10.80	14.22	10.99	112.54	10.92

Table 1. Runtime in seconds (R) and speed-up (S) for our implementation for LWE instances proposed in [19]

We evaluated the running time of nearest planes on LWE instances proposed by Lindner and Peikert [19] for their encryption scheme. For a given secret size n, we selected the optimal lattice dimension m for a basis that is reduced with Hermite delta $\delta = 1.006$, i.e., $m = \sqrt{n \log(q)/\log(1.006)}$.

Given the runtime of our nearest planes implementation, we can estimate the time we require to solve LWE instances of practical dimensions with high probability. To this end, we select the encryption scheme by Lindner and Peikert [19] and revisit the proposed security with respect to our practical algorithm. We consider the sequential attack and attackers that are in possession of 128 and 2^{21} cores.

There are two main reasons to consider 128 cores. Firstly, it is not common that shared memory CPU system have higher core counts. Secondly, our experiments show that the sequential runtime of nearest planes is about 2^{-9} seconds, which means that 128 cores can output $128 \cdot 2^9 = 2^{16}$ close vectors per second, which is exactly the bound proposed by Lindner and Peikert [19]. Those values



(a) $\#enum = 2^{12}$ (b) $\#enum = 2^{15}$ (c) $\#enum = 2^{18}$

Fig. 2. Performance of our implementation executing the nearest planes algorithm on random lattices, for dimensions 404, 517, 597 and 667, with $\#enum = 2^{12}$ in (a), $\#enum = 2^{15}$ in (b) and $\#enum = 2^{15}$ in (c). Runtime in seconds (less is better).

can therefore be used to predict the security of the instances with their runtime assumption against the decoding attack with randomization and a perfect balance between the basis reduction and the decoding step. For an adversary with many resources, we consider a parallel attack on 2^{21} cores, which is about the number of cores of the leading supercomputer in the last `top500` list[3], a rank for high-performance computers.

Our experiments confirmed that the runtime of nearest planes is nearly linear to the number of returned vectors (#enum in the following), see Table 1 and Figure 2. Considering the fact that our implementation is not optimal, it is reasonable to assume that an attacker has an implementation that scales (almost) perfectly linear. It is not surprising that the time for nearest planes depends on the dimension of the lattice. Nevertheless, we choose the runtime of nearest planes for our smallest parameter set as a lower bound, which renders our estimates very conservative. Together with the prediction of nearest planes given in [19], it is possible to find the Hermite delta and number of enumerations that distribute the computational amount equally between nearest planes and BKZ and minimize the expected computational effort.

Acknowledgments. Özgür Dagdelen is supported by the German Federal Ministry of Education and Research (BMBF) within EC-SPRIDE. This work has been co-funded by the DFG as part of project P1 within the CRC 1119 CROSSING.

References

1. Albrecht, M., Fitzpatrick, R., Göpfert, F.: On the efficacy of solving LWE by reduction to unique-SVP. In: to appear at ICISC. Lecture Notes in Computer Science (2013)
2. Albrecht, M.R., Cid, C., Faugère, J.C., Fitzpatrick, R., Perret, L.: On the complexity of the BKW algorithm on LWE. Designs, Codes and Cryptography (2013)

[3] `http://www.top500.org/`

3. Albrecht, M.R., Faugere, J.C., Fitzpatrick, R., Perret, L.: Lazy modulus switching for the BKW algorithm on LWE. In: PKC, vol. 8383, pp. 429–445 (2014)
4. Babai, L.: On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica 6(1), 1–13 (1986)
5. Bai, S., Galbraith, S.D.: Lattice decoding attacks on binary LWE. Cryptology ePrint Archive, Report 2013/839 (2013), `http://eprint.iacr.org/`
6. Bai, S., Galbraith, S.D.: An improved compression technique for signatures based on learning with errors. In: CT-RSA. pp. 28–47 (2014)
7. Blum, A., Kalai, A., Wasserman, H.: Noise-tolerant learning, the parity problem, and the statistical query model. J. ACM 50(4), 506–519 (2003)
8. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: ITCS. pp. 309–325 (2012)
9. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-lwe and security for key dependent messages. In: Advances in Cryptology – CRYPTO 2011, pp. 505–524 (2011)
10. Dagdelen, Ö., Schneider, M.: Parallel enumeration of shortest lattice vectors. In: Euro-Par 2010 - Parallel Processing, vol. 6272, pp. 211–222 (2010)
11. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal gaussians. In: Advances in Cryptology - CRYPTO 2013. pp. 40–56 (2013)
12. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford, CA, USA (2009)
13. Goldreich, O., Goldwasser, S., Halevi, S.: Public-key cryptosystems from lattice reduction problems. In: CRYPTO, vol. 1294, pp. 112–131 (1997)
14. Güneysu, T., Lyubashevsky, V., Pöppelmann, T.: Practical lattice-based cryptography: A signature scheme for embedded systems. In: CHES. pp. 530–547 (2012)
15. Han, D., Kim, M.H., Yeom, Y.: Cryptanalysis of the paeng-jung-ha cryptosystem from PKC 2003. In: PKC, vol. 4450, pp. 107–117 (2007)
16. Hermans, J., Schneider, M., Buchmann, J., Vercauteren, F., Preneel, B.: Parallel shortest lattice vector enumeration on graphics cards. In: AFRICACRYPT 2010, vol. 6055, pp. 52–68 (2010)
17. Kannan, R.: Minkowski's convex body theorem and integer programming. Mathematics of operations research 12(3), 415–440 (1987)
18. Kuo, P.C., Schneider, M., Dagdelen, O., Reichelt, J., Buchmann, J., Cheng, C.M., Yang, B.Y.: Extreme enumeration on GPU and in clouds. In: CHES (2011)
19. Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: Topics in Cryptology – CT-RSA 2011. pp. 319–339 (2011)
20. Liu, M., Nguyen, P.: Solving BDD by enumeration: An update. In: Topics in Cryptology – CT-RSA 2013, pp. 293–309 (2013)
21. Lyubashevsky, V.: Lattice signatures without trapdoors. In: Advances in Cryptology – EUROCRYPT 2012. pp. 738–755 (2012)
22. Micciancio, D., Peikert, C.: Trapdoors for lattices: Simpler, tighter, faster, smaller. In: Advances in Cryptology – EUROCRYPT 2012, pp. 700–718 (2012)
23. Nguyen, P.: Cryptanalysis of the goldreich-goldwasser-halevi cryptosystem from crypto '97. In: CRYPTO, vol. 1666, pp. 288–304 (1999)
24. Plantard, T., Susilo, W.: Broadcast attacks against lattice-based cryptosystems. In: ACNS, vol. 5536, pp. 456–472 (2009)
25. Pöppelmann, T., Güneysu, T.: Towards practical lattice-based public-key encryption on reconfigurable hardware. In: Selected Areas in Cryptography. pp. 68–85 (2013)
26. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: STOC. pp. 84–93 (2005)