Progressive lattice sieving

Thijs Laarhoven[1] and Artur Mariano[2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
mail@thijs.com
[2] University of Coimbra, Coimbra, Portugal
artur.mariano@co.ip.pt

Abstract. Most algorithms for hard lattice problems are based on the principle of rank reduction: to solve a problem in a d-dimensional lattice, one first solves one or more problem instances in a sublattice of rank $d-1$, and then uses this information to find a solution to the original problem. Existing lattice sieving methods, however, tackle lattice problems such as the shortest vector problem (SVP) directly, and work with the full-rank lattice from the start. Lattice sieving further seems to benefit less from starting with reduced bases than other methods, and finding an approximate solution almost takes as long as finding an exact solution. These properties currently set sieving apart from other methods.
In this work we consider a progressive approach to lattice sieving, where we gradually introduce new basis vectors only when the sieve has stabilized on the previous basis vectors. This leads to improved (heuristic) guarantees on finding approximate shortest vectors, a bigger practical impact of the quality of the basis on the run-time, better memory management, a smoother and more predictable behavior of the algorithm, and significantly faster convergence – compared to traditional approaches, we save between a factor 20 to 40 in the time complexity for SVP.

Keywords: lattice-based cryptography, lattice sieving, shortest vector problem (SVP), nearest neighbor searching

1 Introduction

Finding short lattice vectors. A central hard problem in the study of lattices is the shortest vector problem (SVP): given a lattice $\mathcal{L} = \{\sum_{i=1}^{d} c_i \boldsymbol{b}_i : c_i \in \mathbb{Z}\} \subset \mathbb{R}^d$, find a non-zero lattice vector $\boldsymbol{s}$ of minimum norm, i.e. find $\boldsymbol{s} \in \mathcal{L}$ satisfying $\|\boldsymbol{s}\| = \lambda_1(\mathcal{L}) := \min_{\boldsymbol{x} \in \mathcal{L} \setminus \{\boldsymbol{0}\}} \|\boldsymbol{x}\|$. The security of many lattice-based cryptographic primitives can be traced back to the hardness of SVP or approximate SVP (γ-SVP), where a non-zero vector $\boldsymbol{s} \in \mathcal{L}$ of norm at most $\gamma \cdot \lambda_1(\mathcal{L})$ suffices as a solution. Being able to estimate the computational hardness of these problems is crucial for accurately assessing capabilities of cryptographic adversaries, and for selecting parameters in cryptographic schemes [MLC+17]. Algorithms for exact SVP are essential both for solving exact and approximate SVP, as the latter problem can (heuristically) be reduced to solving several exact SVP instances in lower-dimensional lattices through the BKZ algorithm [Sch87, SE94, CN11].

Exact SVP algorithms. There are currently two classes of algorithms for exact SVP: algorithms requiring superexponential time $2^{\omega(d)}$ in the lattice dimension d, using a negligible amount of memory (such as enumeration [Kan83, FP85, SE94, GNR10, AN17]), and algorithms requiring single exponential time and space $2^{\Theta(d)}$ (such as lattice sieving [AKS01, HPS11] and Voronoi cell approaches [SFS09, MV10a, Laa16]). Although the latter methods have clear drawbacks due to the large memory requirement, these algorithms will inevitably surpass the former algorithms in terms of the time complexity in high dimensions. Conservative estimates of the (post-quantum) hardness of SVP are therefore often based on the state-of-the-art asymptotics for the best (quantum) lattice sieving algorithms [ADPS16, DLL$^+$18, BDK$^+$18].

Lattice sieving. Lattice sieving was introduced in 2001 by Ajtai–Kumar–Sivakumar [AKS01], and was only made somewhat practical less than 10 years ago with fast heuristics [NV08, MV10b]. Recent years have seen further developments in decreasing the asymptotic time complexity of sieving at the cost of using more space [WLTB11, ZPH13, BGJ14, Laa15, BGJ15, LdW15, BL16, BDGL16], while tradeoffs in the reverse direction have also been studied recently to reduce the large memory requirement [BLS16, HK17, HKL18]. Various efforts have further been made to make these algorithms competitive in high-performance computing environments [Sch11, MS11, Sch13, FBB$^+$14, MTB14, MODB14, IKMT14, MLB15, BNvdP16, MB16, MLB17, YKYC17, Duc18]. The theoretically fastest method in high dimensions is currently the LDSieve, with asymptotic time and space complexities $2^{0.29d+o(d)}$ [BDGL16], while in practice the GaussSieve and HashSieve appear to be the most practical in high dimensions [MB16, MLB17, YKYC17].

Differences with other approaches. Existing lattice sieving methods are fundamentally different from other SVP approaches in several ways. Whereas for instance enumeration and Voronoi cell-based methods use a rank reduction step to reduce a d-dimensional problem to problems in lattices of rank $d-1$, lattice sieving never considers other lattices than the full-rank one. And unlike other methods, lattice sieving (i) does not appear to benefit greatly from being given better lattice bases as input (i.e. BKZ-reduced bases with larger block sizes), and (ii) does not appear to be much faster when an approximate solution suffices – often the algorithm only starts to find shorter vectors after the algorithm is already 80% finished with finding an exact solution. All these properties currently set sieving apart from other methods, and so one might ask whether lattice sieving can be adjusted and perhaps improved by applying similar rank-reduction techniques, so that sieving also obtains these "natural" properties of finding short vectors faster, and performing better when given more reduced bases.

1.1 Contributions

We present *progressive lattice sieving* as a new baseline sieving approach, which resolves many of the above differences with other methods, and greatly improves the performance of heuristic sieving algorithms in practice. Progressive

lattice sieving uses a bottom-up approach, initially starting with sieving in a low-rank sublattice of the input lattice. Then, when this subspace of the original space has been saturated with vectors from this sublattice, new basis vectors are introduced to find shorter lattice vectors in sublattices of higher rank, until ultimately the algorithm reaches the full-rank lattice and attempts to find short vectors in the original lattice. Using this rank reduction approach, progressive sieving offers various benefits over standard sieving techniques, which we list below. These are mostly independent of the exact underlying sieve used (e.g. the GaussSieve [MV10b], HashSieve [Laa15], or LDSieve [BDGL16]), although in some cases the improvements are bigger than in other cases.

Faster convergence: Overall, progressive sieving finds shortest vectors in lattices much faster than using current methods, with speedups as large as a factor 40 in dimension 70. Tables 1–2 illustrate improvements for 70-dimensional lattices for the GaussSieve and HashSieve, and Figure 1 shows the improvements for various dimensions using progressive sieving.

Better memory usage: Working on low-rank lattices requires fewer vectors to make progress, allowing one to do a large part of the algorithm using much less memory than current approaches. Experiments further show progressive sieving uses slightly less memory overall (Figure 2b). This is especially relevant as the main bottleneck in sieving is hitting the memory wall [MB16].

Heuristic guarantees for approximate SVP: Using the Geometric Series Assumption, progressive sieving heuristically finds approximate solutions faster than the full solution, unlike other approaches which commonly require a large number of vectors and reductions to make any progress at all.[3]

Larger impact of reduced bases: The more reduced the input basis is, the faster shorter lattice vectors are found, which contributes to a faster overall running time. Similar to pruning in enumeration, this further opens up some new directions for potential improvements (see Section 5.2).

Better predictability: As illustrated in the experimental profiles (Figures 2–3), various aspects of sieving are easier to predict and easier to explain theoretically with progressive sieving, which may lead to more accurate predictions when extrapolating to higher dimensions.

Less resource contention: Vectors in the sieved list are only modified sporadically. Instead of using the lock-free mechanism of [MB16, MLB17], to avoid collisions between different nodes, we may therefore be able to construct parallel implementations with lower-overhead incurring mechanisms.

Outline. The remainder of this paper is organized as follows. In Section 2, we introduce notation and related work on lattice sieving. Section 3 describes progressive sieving, and an experimental comparison with previous approaches is given in Section 4. Section 5 discusses various aspects of these different approaches to sieving, and Section 6 concludes with open problems for future work.

[3] Arguably for current sieving approaches one could also take a sublattice of the full lattice, based on the GSA, and do sieving on that lattice. However, in that case the lattice may be too small (no solutions found) or too big (taking too much time). With progressive sieving no a priori choices need to be made (see also Section 5.1).

Exact SVP	⟵ **GaussSieve** ⟶			⟵ **HashSieve** ⟶		
	LLL	BKZ-10	BKZ-30	LLL	BKZ-10	BKZ-30
Standard sieving	19100	18100	16500	3300	3050	2900
Progressive sieving	595	440	390	165	125	115
Speedup factor	**32**×	**41**×	**42**×	**20**×	**24**×	**25**×

Table 1. Running times (in seconds) for solving exact SVP on 70-dimensional lattices from the SVP challenge [SVP18], using the baseline GaussSieve algorithm [MV10b] and the near-neighbor-based HashSieve extension of the GaussSieve [Laa15]. The lattice bases are pre-reduced with either LLL or BKZ with blocksize 10 or 30, where we used `fplll` [fplll18] to perform the basis reduction. The timings above are based only on the sieving step, and do not include the time for the basis reduction step.

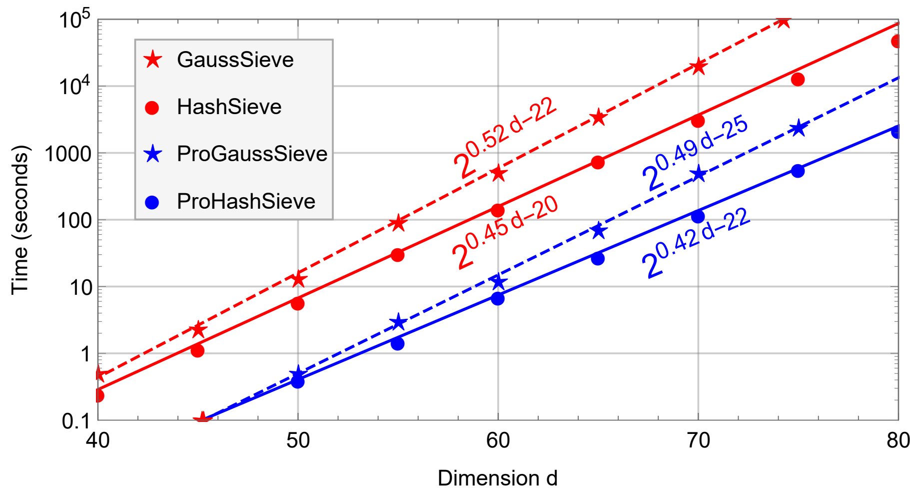


Fig. 1. Time complexities for solving exact SVP on BKZ-10 reduced bases for the SVP challenge lattices, using standard lattice sieving approaches (GaussSieve and HashSieve) and progressive sieving modifications of these algorithms (ProGaussSieve and ProHashSieve). The formulas written in the figure denote the least-squares fits of the four data sets up to two significant digits.

Concurrent work. During the write-up of this paper, we learned similar ideas were independently and concurrently being studied by Ducas, which were later published in [Duc18]. The focus of that work however is on a different improvement to sieving ("a few dimensions for free"), and the idea of progressive sieving is only mentioned in passing, with little explanation where the improvement comes from. In this paper we chose to focus only on the progressive sieving idea, to understand why it works and where further improvements can be found.

2 Preliminaries

Given a set $\mathbf{B} = \{\boldsymbol{b}_1, \dots, \boldsymbol{b}_m\} \subset \mathbb{R}^d$ of m independent d-dimensional vectors, we denote the (rank-m) lattice spanned by $\mathbf{B}$ by $\mathcal{L}(\mathbf{B}) = \{\sum_{i=1}^m c_i \boldsymbol{b}_i : c_i \in \mathbb{Z}\}$. With abuse of notation, we sometimes write $\mathcal{L} = \mathcal{L}(\mathbf{B})$ when the basis $\mathbf{B}$ is implicit. For $k \leq m$, we write $\mathcal{L}_k = \mathcal{L}_k(\mathbf{B})$ for the sublattice $\{\sum_{i=1}^k c_i \boldsymbol{b}_i : c_i \in \mathbb{Z}\} \subseteq \mathcal{L}$ spanned by the first k basis vectors of $\mathbf{B}$. For simplicity, we commonly assume that the original problem concerns a lattice problem in a lattice of full rank ($m = d$); however, using rank reduction we sometimes consider problems in sublattices as well. We denote by $\mathrm{Vol}(\mathcal{L}) = \sqrt{\det(\mathbf{B}^T\mathbf{B})}$ the volume of a lattice $\mathcal{L}$, and by $\mathrm{Vol}(\mathcal{A})$ the volume of a set $\mathcal{A} \subset \mathbb{R}^d$.

2.1 Heuristic assumptions

To analyze lattice algorithms, various heuristic assumptions are commonly used. These are not proven and may well be false for certain extreme lattices/bases, but commonly hold for *random* lattices encountered in cryptography. Analyzing algorithms under these (average-case) assumptions therefore commonly leads to more accurate security assessments than analyses based on provable (worst-case) bounds, which are commonly not tight for random lattices.

The *Gaussian heuristic* states that, given a subset $\mathcal{A} \subset \mathbb{R}^d$, the number of lattice points in $\mathcal{A}$ is roughly $|\mathcal{A} \cap \mathcal{L}| \approx \mathrm{Vol}(\mathcal{A}) / \mathrm{Vol}(\mathcal{L})$. As a consequence, we expect a ball of radius r around a random point in space to contain a lattice point iff $r \geq r_0 \sim \sqrt{d/(2\pi e)}$, and heuristically we expect $\lambda_1(\mathcal{L}) \approx \sqrt{d/(2\pi e)}$.

The *Geometric Series Assumption (GSA)* states that, after performing BKZ reduction [Sch87] with block-size β on a full-rank lattice basis, the Gram-Schmidt orthogonalization $\boldsymbol{b}_1^* \dots \boldsymbol{b}_d^*$ of the output basis satisfies $\|\boldsymbol{b}_i^*\| \approx \delta^{d-2i+1} \cdot \mathrm{Vol}(\mathcal{L})^{1/d}$ for all i. Here $\delta = \delta(\beta)$ determines the quality of the basis; the smaller δ, the more orthogonal the basis and the shorter $\boldsymbol{b}_1$. Note that $\prod_{i=1}^d \|\boldsymbol{b}_i^*\| = \mathrm{Vol}(\mathcal{L})$.

2.2 Lattice sieving algorithms

The Nguyen–Vidick sieve. When the idea of lattice sieving was introduced by Ajtai–Kumar–Sivakumar [AKS01], it was still a purely theoretical idea, and no practical instantiations existed until Nguyen–Vidick described a heuristic version of this idea in [NV08]. This sieve starts by sampling many long lattice vectors (e.g. from a discrete Gaussian over the lattice), and then iteratively applies a sieve to this list of vectors to produce a list with fewer, shorter lattice vectors. This sieve essentially works by considering all pairwise combinations of vectors in the input list, and seeing if any of the sums/differences are shorter than the original vectors. Note that any such combination of lattice vectors again forms a vector which is in the lattice. These pairwise combinations are then used in the next iteration, while the input vectors to the sieve are discarded. This process of throwing away many short lattice vectors is unnaturally wasteful, and the Nguyen–Vidick sieve is therefore not commonly used in practice.

Algorithm 1 The **standard** GaussSieve algorithm – **GaussSieve**

```
1: Initialize an empty list L ← ∅ and an empty stack S ← ∅
2: Initialize COLLISIONS ← 0
3: while true do
4:     Get a vector v from the stack S or sample a new one from ℒ(b_1, ..., b_d)
5:     Reduce v with all w ∈ L                 // this naively takes O(|L|) time
6:     Reduce all w ∈ L with v                 // NNS can speed up these searches
7:     Move reduced vectors w ∈ L from the list L to the stack S
8:     if v has not changed then
9:         Add v to the list L
10:    else
11:        if v ≠ 0 then
12:            Add v to the stack S
13:        else
14:            COLLISIONS++
15:            if COLLISIONS = 100 then        // a target norm may be used instead
16:                return argmin_{v∈L} ‖v‖     // the shortest vector found so far
```

The GaussSieve and ListSieve. In 2010, Micciancio–Voulgaris [MV10b] introduced two new (heuristic) lattice sieving algorithms, which unlike the Nguyen–Vidick sieve both start from an empty list L, and gradually grow this list by adding more (short) lattice vectors to the list, until this list contains a solution.

In the *ListSieve*, before adding a randomly sampled lattice vector $\boldsymbol{v}$ to the list, the vector is reduced with all list vectors $\boldsymbol{w} \in L$ by checking whether either of the sum/difference vectors $\boldsymbol{v} \pm \boldsymbol{w}$ are shorter than $\boldsymbol{v}$ – if so, $\boldsymbol{v}$ is replaced by this shorter vector. Once $\boldsymbol{v}$ has been reduced with all list vectors, and no pairwise sums/differences with list vectors lead to shorter vectors anymore, $\boldsymbol{v}$ is added to the list. This process is repeated until L contains a shortest vector.

In the *GaussSieve* this process is essentially the same, except that list vectors $\boldsymbol{w} \in L$ are also reduced with newly sampled vectors $\boldsymbol{v}$, before adding $\boldsymbol{v}$ to the list; in contrast, in the ListSieve vectors which have been added to the list L are never modified again. Algorithm 1 describes the GaussSieve, while removing Lines 6–7 leads to the ListSieve algorithm.

Extensions and variants. Over the last few years, various extensions and variants of these heuristic sieving algorithms have been studied, to make them even more efficient in practice. High-dimensional nearest neighbor search algorithms have been used to obtain speed-ups in the searches in Lines 5–6 of Algorithm 1, which naively take time $O(|L|)$ but can be done in sublinear time $O(|L|^{\rho})$ with $\rho < 1$ with more advanced methods of indexing and querying the list L [Laa15, LdW15, BGJ15, BL16, BDGL16]. Recent work on tuple sieving [BLS16, HK17, HKL18] focused on reducing the memory of lattice sieving, by considering a larger number of vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_k \in L$ from the list to form short combinations $\boldsymbol{v} \pm \boldsymbol{w}_1 \pm \cdots \pm \boldsymbol{w}_k$ with $\boldsymbol{v}$. Other work has shown that the same techniques can be used to solve the Closest Vector Problem (CVP), and how near neighbor techniques can be used to obtain fast CVP algorithms with preprocessing [Laa16].

3 Progressive lattice sieving

When given a (full-rank) d-dimensional lattice basis, standard sieving approaches attempt to saturate this entire d-dimensional space from the start, always working with vectors from the full-dimensional space. With pairwise reductions between lattice vectors, one needs roughly $2^{0.21d+o(d)}$ vectors to saturate this space and make significant progress in obtaining shorter vectors. This means that also approximate shorter vectors are not found until the list size approaches $2^{0.21d+o(d)}$, which means a lot of effort is spent even before any progress is made at all. Naively this means the time complexity to find any shorter vectors at all is at least $2^{0.42d+o(d)}$ (or $2^{0.29d+o(d)}$ using near neighbor techniques [BDGL16]).

Using BKZ, finding approximate shortest vectors does not necessarily require the same amount of effort as finding exact solutions. More specifically, BKZ only solves SVP instances in lower-dimensional lattices, thereby finding short lattice vectors in (projected) sublattices sooner. The idea of progressive lattice sieving is similar: by first considering low-dimensional sublattices of the full lattice, which already contain many short lattice vectors, we will find approximate solutions faster, which will eventually contribute to finding exact solutions faster as well.

The GaussSieve. Applying progressive sieving to the GaussSieve means making the following modifications. First, we start with a sublattice of the original lattice, spanned by the first few basis vectors. We then run a sieve on this sublattice, until we reach the stopping criterion (e.g. a certain number of collisions). We then add a new basis vector to the search space, and continue sieving in this sublattice of slightly higher rank. We repeat this procedure until we reach the complete lattice, where we expect to find an exact solution to SVP. The result of applying progressive sieving to the GaussSieve is sketched in Algorithm 2, where the main modifications are (1) vectors are sampled from sublattices, and (2) the rank counter is increased and the collision counter is reset when we reach 100 collisions. For simplicity we start with a sublattice of rank 10 – everything below rank 30 or 40 generally finishes within a second anyway, so setting the initial rank to any rank below 40 will lead to similar results in higher dimensions.

Other sieving algorithms. Naturally the same idea can be trivially applied to most other sieving algorithms as well. Applying the same idea to the List-Sieve [MV10b] would simply mean we do not reduce list vectors with sampled vectors in Lines 6–7 of Algorithm 2. Near neighbor extensions to sieving [Laa15,BDGL16] only affect the search procedure for finding reducing pairs of vectors, and this can naturally be combined with progressive sieving as well. For tuple sieving [BLS16, HK17, HKL18] we simply run tuple sieving on sublattices until the corresponding sublattice has been saturated, after which we increase the rank as before. Applying progressive sieving to the CVP sieve of [Laa16] is also straightforward. The same idea can also be applied to the Nguyen–Vidick sieve, but in that case the integration of progressive sieving is slightly more contrived; since the practical results with the Nguyen–Vidick sieve will likely be worse than when using the GaussSieve, we have chosen to omit this application.

Algorithm 2 The **progressive** GaussSieve algorithm – **ProGaussSieve**

```
1: Initialize an empty list L ← ∅ and an empty stack S ← ∅
2: Initialize COLLISIONS ← 0, RANK ← min{10, d}
3: while true do
4:     Get a vector v from the stack S or sample a new one from L(b_1, ..., b_RANK)
5:     Reduce v with all w ∈ L                 // this naively takes O(|L|) time
6:     Reduce all w ∈ L with v                 // NNS can speed up these searches
7:     Move reduced vectors w ∈ L from the list L to the stack S
8:     if v has not changed then
9:         Add v to the list L
10:    else
11:        if v ≠ 0 then
12:            Add v to the stack S
13:        else
14:            COLLISIONS++
15:            if COLLISIONS = 100 then
16:                if RANK = d then
17:                    return argmin_{v∈L} ||v||  // the shortest vector found so far
18:                else
19:                    RANK++                   // continue with the next sublattice
20:                    COLLISIONS ← 0           // reset the collisions counter
```

4 Experiments

Heuristically, sieving naively requires a search over all pairs of vectors in the list (when not using near neighbor techniques), leading to a quadratic time complexity in the list size. With progressive sieving, these arguments still hold, and so the asymptotic improvement of progressive sieving will not be visible in the leading time/memory exponents. Experimentally however there are rather large hidden order terms, and these may become smaller with progressive sieving. To get an insight into this improvement, we will experimentally evaluate progressive sieving and compare it to standard sieving approaches. As a baseline we will use the GaussSieve [MV10b], the baseline sieve without near neighbor searching, and the HashSieve [Laa15], which is perhaps the most practical near neighbor extension of the GaussSieve to date [MLB15, MB16, MLB17]. In Section 5 we will give a more in-depth discussion of various aspects of these results.

Experiment setting. All experiments were performed on a Medion Erazer P6661 laptop with an Intel Core i7-6500 CPU (2.50 GHz) and 8 GB of RAM. Except for approximate SVP, where the termination condition was finding a vector of a certain length, all experiments used a bound on the number of "collisions" (reducing to the all-zero vector) as the termination condition. Similar to e.g. [MV10b, IKMT14] we used (an unoptimized version of) Klein's sampler [Kle00]. For the HashSieve, we used the same parameters as in [Laa15, MLB15]. All experiments were performed on lattices from the SVP challenge [SVP18], where the LLL/BKZ basis reduction was done using `fplll` [fplll18].

4.1 Profiles

To get an idea how progressive sieving differs from classical sieving in practice, Figures 2–3 depict typical profiles of the classical and progressive HashSieve, when solving SVP on 70-dimensional lattices from the SVP challenge [SVP18]. As described in the captions of the figures, the time complexities are significantly better, the list size increases more steadily (and predictably) than in the original HashSieve, approximate solutions are found considerably faster, and vectors in the list are updated much less frequently in progressive sieving.

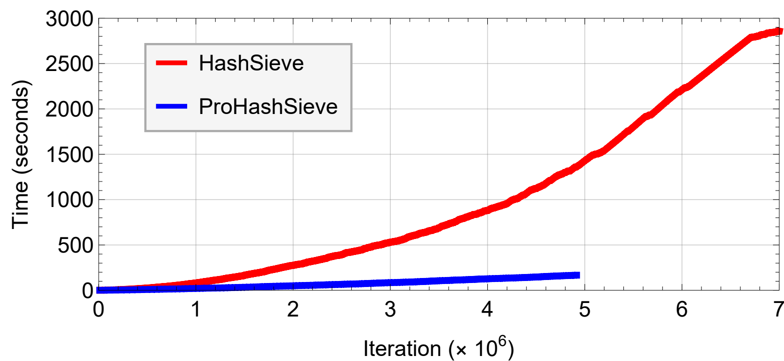
Note that as in Figure 2c, classical sieving can be viewed as a special case of progressive sieving, with the initial value of RANK set to d. Also note that in all figures, the horizontal axis counts the number of *iterations* of the **while**-loop of Algorithms 1–2), which translates to actual time complexities as in Figure 2a – this means that e.g. a vector of Euclidean norm 2400 in this 70-dimensional lattice is not found a factor $5\times$ faster (as one might guess from Figure 3a), but close to a factor $5 \cdot 20 \approx 1000\times$ faster (taking into account Figure 2a).

4.2 Results

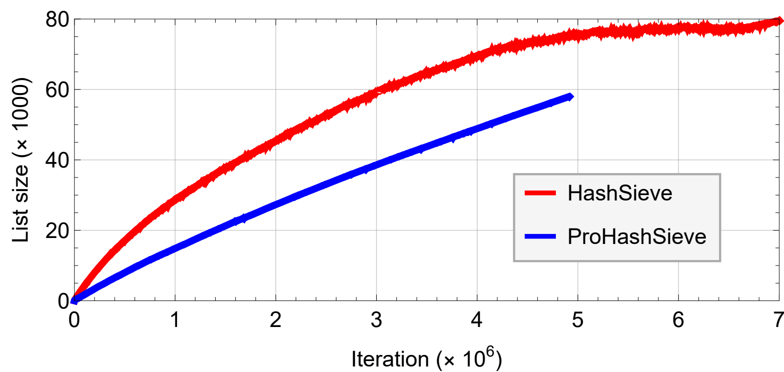
To get reliable complexity estimates for arbitrary dimensions, we ran several experiments of both standard and progressive sieving on lattices of dimensions 40 to 80, the results of which are displayed in Figure 1. These results clearly show a large decrease in the time complexities for sieving in arbitrary dimensions, both for the classical GaussSieve and the near-neighbor-optimized HashSieve algorithm [Laa15]. For both algorithms the improvement is more than a factor $20\times$ for all tested dimensions, showing that the improvement of progressive sieving can indeed be combined with near neighbor techniques.

Focusing only on 70-dimensional lattices, we also ran experiments with different quality bases, to see how the basis reduction affects the performance of (progressive) sieving. The results in Table 1 are based on at least 10 experiments each on randomized lattice bases (where only the experiments that took several hours are based on just 10 runs). As can be seen in the table, progressive sieving benefits more from reduced bases, with the speedup factor increasing as the quality of the basis improves. We further observe that the speedup factor for the HashSieve is slightly smaller than for the GaussSieve, which can be explained by the HashSieve spending most of its time working with the near neighbor data structure – costs which are not affected by progressive sieving.

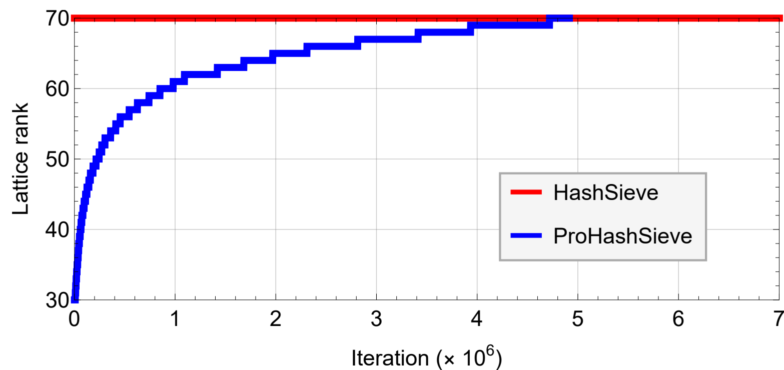
Finally, to illustrate the improvements for finding approximate shortest vectors, Table 2 lists the time complexities for 1.1-SVP on 70-dimensional lattices from the SVP challenge [SVP18]: the measured complexities are based on terminating the algorithm as soon as a vector $\boldsymbol{v} \in \mathcal{L}$ of norm $\|\boldsymbol{v}\| \leq 1.1 \cdot \lambda_1(\mathcal{L})$ has been found. As the results show (and as could already be seen in Figure 3a), standard sieving barely benefits from this relaxed termination requirement, while progressive sieving improves considerably both when an approximate solution suffices, and when the input bases is more reduced, leading to extreme speedup factors as high as $5000\times$. This highlights a weakness of existing lattice sieving methods.



(a) **Time complexities.** Iterations (Line 3) take significantly less time in the ProHashSieve, leading to finding short lattice vectors much faster.

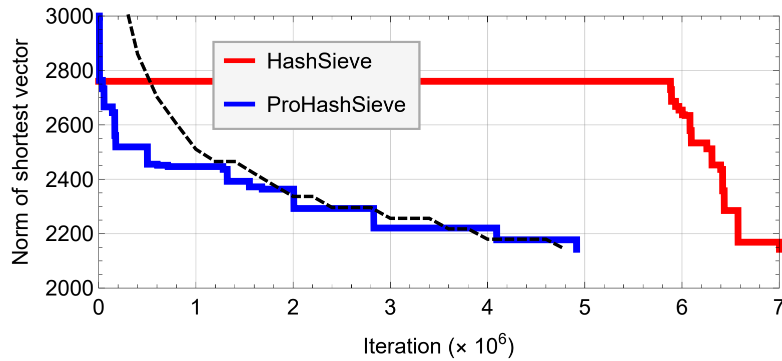


(b) **List sizes.** Whereas the list size behaves rather erratically in the original HashSieve, the ProHashSieve shows a much more steady behavior.

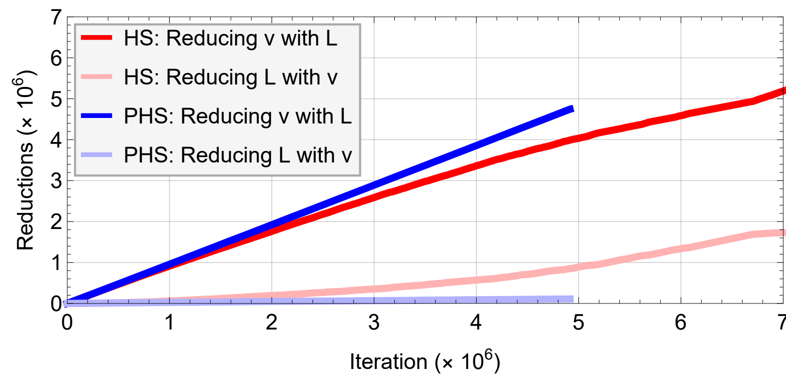


(c) **Lattice rank.** For the ProHashSieve, the rank of the sublattice (roughly) follows a square-root law, with more time spent on higher ranks.

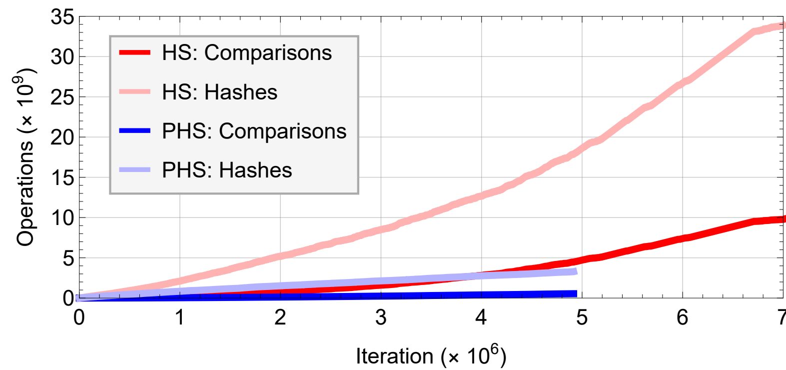
Fig. 2. Sieving profiles for the HashSieve (red) and ProHashSieve (blue), when given as input a BKZ-30 reduced lattice basis of a 70-dimensional lattice.



(a) **Norm of the shortest vector.** The dashed black curve shows the theoretical prediction for the ProHashSieve based on the GSA and rank.



(b) **Reductions ($\times 10^6$).** Progressive sieving rarely sees reductions of the list with new vectors, making the GaussSieve similar to the ListSieve.



(c) **Operations ($\times 10^9$).** Both algorithms roughly have the same ratio of hashing operations (finding near neighbors) and vector comparisons.

Fig. 3. Sieving profiles for the HashSieve (red) and ProHashSieve (blue), when given as input a BKZ-30 reduced lattice basis of a 70-dimensional lattice.

Approximate SVP	⟵ **GaussSieve** ⟶			⟵ **HashSieve** ⟶		
($\gamma = \mathbf{1.1}$)	LLL	BKZ-10	BKZ-30	LLL	BKZ-10	BKZ-30
Standard sieving	18500	17200	15600	3180	2960	2700
Progressive sieving	120	40	3	65	20	2
Speedup factor	**150**×	**400**×	**5000**×	**50**×	**150**×	**1000**×

Table 2. Running times (in seconds) for solving approximate SVP with approximation factor $\gamma = 1.1$ on 70-dimensional lattice bases from the SVP challenge [SVP18]. When the bases are sufficiently well-reduced, low-rank sublattices already contain very short lattice vectors, leading to substantially faster convergence for progressive sieving.

5 Discussion

In this section we will briefly discuss different aspects of lattice sieving, and how various design choices affect the performance of (progressive) sieving in practice.

5.1 Approximate SVP

As shown in Table 2, progressive sieving benefits greatly from being given a relaxed termination condition, in the sense that an approximate solution to SVP suffices. To quantify where this improvement comes from, recall that by the Geometric Series Assumption (GSA), BKZ gives us a reduced lattice basis $\{\boldsymbol{b}_1, \dots, \boldsymbol{b}_d\}$ where the Gram-Schmidt orthogonalization vectors satisfy $\|\boldsymbol{b}_i^*\| \approx \delta^{d-2i+1} \cdot \mathrm{Vol}(\mathcal{L})^{1/d}$ for all $i = 1, \dots, d$. As a result, the first k basis vectors together form a rank-k sublattice $\mathcal{L}_k = \mathcal{L}(\boldsymbol{b}_1, \dots, \boldsymbol{b}_k) \subset \mathcal{L}$ of volume:

$$\mathrm{Vol}(\mathcal{L}_k) = \prod_{i=1}^{k} \|\boldsymbol{b}_i^*\| = \mathrm{Vol}(\mathcal{L})^{k/d} \cdot \delta^{\sum_{i=1}^{k}(d-2i+1)} = \mathrm{Vol}(\mathcal{L})^{k/d} \cdot \delta^{k(d-k)} \tag{1}$$

By the Gaussian heuristic we have $\lambda_1(\mathcal{L}) \approx \mathrm{Vol}(\mathcal{L})^{1/d}/\sqrt{2e\pi d}$, and therefore the shortest vector in this rank-k sublattice $\mathcal{L}_k$ has relative norm:

$$\lambda_1(\mathcal{L}_k) \approx \frac{\mathrm{Vol}(\mathcal{L}_k)^{1/k}}{\sqrt{2e\pi k}} = \frac{\delta^{d-k}\,\mathrm{Vol}(\mathcal{L})^{1/d}}{\sqrt{2e\pi k}} \approx \delta^{d-k}\sqrt{\frac{d}{k}} \cdot \lambda_1(\mathcal{L}) \tag{2}$$

To obtain a solution to γ-SVP, i.e. approximate SVP with approximation factor γ, we need $\delta^{d-k}\sqrt{d/k} \approx \gamma$. Depending on the basis quality δ, dimension d, and approximation factor γ, this tells us how many dimensions we can essentially "skip" at the end when looking for approximate solutions: already at rank k, we will then expect to find a vector of sufficiently short length.

One could argue that the picture sketched by Table 2 is painted too rosily in favor of progressive sieving, since classical sieving approaches could also find approximate solutions faster by (a) computing a similar heuristic on what rank

k is required to find an approximate solution in a suitable sublattice, and then (b) simply running sieving on such a sublattice until an approximate solution has been found. Even in that case however, progressive sieving has the benefits of not requiring an a priori choice of k (it automatically finds the right rank k required to find a sufficiently short vector), and having a faster convergence for exact SVP in such sublattices as well (as sketched in Table 1).

5.2 Effects of basis reduction

The experiments in Section 4 further suggest that progressive sieving benefits more from being given a more reduced basis than classical sieving approaches. To explain this, we observe that there are two sides to having better bases, and previous sieving approaches only benefited from one of these advantages.

Standard sieving. When given a better basis, classical sieving methods will be able to sample shorter lattice vectors in Line 4 more easily. Being able to sample shorter vectors means that vectors will need to be reduced less frequently before "stabilizing" in the list, thus reducing the overall cost of the algorithm as for each vector, we might save some searches and reductions.

Progressive sieving. Besides the benefit stated above, which also holds for progressive sieving, there is a second advantage to being given a nice lattice basis. This is closely related to the heuristic analysis for approximate SVP above, as having a reduced basis means that δ is smaller and therefore low-rank sublattices will already contain shorter lattice vectors. And being able to find very short lattice vectors early in low ranks, which will then be contained in the list L when moving to higher-rank sublattices, means that reductions will proceed faster in those higher ranks as well.

Formally, let $\mathcal{S}^{d-1} = \{\boldsymbol{x} \in \mathbb{R}^d : \|\boldsymbol{x}\| = 1\}$ denote the unit sphere, and let $\boldsymbol{v} \sim \mathcal{S}^{d-1}$ be sampled uniformly at random from this sphere. Then, using spherical cap arguments similar to e.g. [BDGL16], one can show that the probability that $\boldsymbol{v}$ can be reduced with a random vector $\boldsymbol{w} \in \mathcal{S}^{d-1}$ (i.e. $\|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \|\boldsymbol{v}\|$) is $(3/4)^{d+o(d)} \approx 2^{-0.21d+o(d)}$. However, when the norms of list vectors $\boldsymbol{w}$ differ from 1, so that the norms of $\boldsymbol{v}$ and $\boldsymbol{w}$ are different, this probability changes. Intuitively, for small $\|\boldsymbol{w}\| \to 0$ the probability of reducing $\boldsymbol{v}$ with $\boldsymbol{w}$ approaches 1, as either $\boldsymbol{v} + \boldsymbol{w}$ or $\boldsymbol{v} - \boldsymbol{w}$ is likely to be shorter than $\boldsymbol{v}$. For large $\|\boldsymbol{w}\| \to \sqrt{2}\|\boldsymbol{v}\|$ the probability of being able to reduce $\boldsymbol{v}$ with $\boldsymbol{w}$ approaches 0, and for $\|\boldsymbol{w}\| \geq 2\|\boldsymbol{v}\|$ it is even impossible for such a reduction to take place. For arbitrary $\alpha \in (0, \sqrt{2})$ and random $\boldsymbol{w} \in \alpha\mathcal{S}^{d-1}$, a geometric exercise shows that this probability scales as $(1 - \alpha^2/2)^{d/2+o(d)}$, showing that if list vectors $\boldsymbol{w} \in L$ are a factor α shorter than the sampled vectors $\boldsymbol{v}$, the likelihood that $\boldsymbol{v}$ can be reduced with the list is exponentially larger (in α) than when vectors in the list have equal norm as $\boldsymbol{v}$.

Summarizing, if the input basis to the sieve is more reduced, then in lower ranks the algorithm will already find shorter lattice vectors to add to the list L, and having shorter vectors in the list means that new vectors can be reduced

faster and more easily. Since in classical sieving approaches it takes much longer to find (approximate) short lattice vectors, previous approaches to sieving do not benefit from this accelerated reductions when the list vectors are short. We believe this mostly explains the bigger improvements to progressive sieving when using more reduced bases.

Pruning. Note that when the lattice basis is very reduced, the last coefficients of the shortest lattice vector in terms of these basis vectors are commonly small or zero. If the search space in the last few coefficients is small, one could simply guess or enumerate these potential coefficients, while doing sieving on the first coefficients. This suggests an approach where we only do sieving up to a certain rank, and for the last few basis vectors we use a different procedure of either guessing the coefficients and randomizing/restarting when we fail (similar to pruned enumeration with randomized bases), or using e.g. enumeration or Babai rounding to deal with these latter dimensions. A somewhat similar idea of dealing with these last dimensions faster was analyzed in [Duc18].

5.3 Effects of the sampler

By the sampler, we are referring to the procedure in Line 4 of Algorithms 1–2 of sampling new lattice vectors, in case the stack is empty: a new vector is sampled from a certain distribution over the lattice, using some efficient sampling algorithm. This sampling is often done through a randomized enumeration procedure [Kle00], and the exact specification of this procedure determines how short the sampled lattice vectors will be, how often similar vectors (in particular, the vector $\mathbf{0}$) are sampled, and how long the procedure takes to produce a new sample.

Standard sieving. Choosing the best parameters for the sampler, to get the best performance for sieving, is a rather cumbersome procedure: there are several parameters to tune in the sampler alone, and the best choice varies per dimension and per lattice sieve method (GaussSieve, HashSieve, etc.). Experiments in the past have shown that for previous sieving approaches, this choice of parameters may also greatly influence the practical performance of the algorithm [IKMT14, FBB+14, MLB15, MB16], which makes this aspect of sieving hard to deal with properly – choosing parameters optimally is both difficult and important.

Progressive sieving. Although choosing parameters accurately still influences the performance of progressive sieving, sampling longer vectors causes less of a slowdown due to the list having many short vectors early on (as discussed before, in the context of the input basis quality). Long sampled vectors are reduced in length faster and more easily, so that sampling longer vectors is less of an issue. Therefore fine-tuning the sampler will likely not lead to as big improvements for progressive sieving as for standard sieving. However, this also means that the

numbers in Tables 1–2 and Figure 1 may not be entirely accurate when using optimized samplers – our numbers are based on a simple, straightforward proof-of-concept implementation of Klein's sampler, rather than having optimized all aspects of the algorithm (and in particular this sampling routine).[4]

5.4 Effects of list updates (GaussSieve vs. ListSieve)

As mentioned, the main difference between Micciancio–Voulgaris' GaussSieve and ListSieve is that in the ListSieve, vectors that are added to the list are never modified again. By "list updates", we are referring to either making updates to the list vectors as in the GaussSieve, or never updating list vectors as in the ListSieve algorithm.

Standard sieving. In classical sieving approaches, as can for instance be seen in Figure 3b, lattice vectors in the sieved list are quite often reduced and moved back to the stack, after which they are processed again and pushed back to the list. In the ListSieve-variant of such algorithms, where list vectors are never touched again, one would miss all these reductions, leading to a significantly worse performance overall [MODB14]. Although the ListSieve is conceptually slightly simpler, the performance loss would not be worth it to ever consider using the ListSieve instead of the GaussSieve in high-performance environments.

Progressive sieving. For progressive sieving, reductions of list vectors almost never happen, as can also be seen in Figure 3b. Recall that the experiments described in Figures 2–3 are based on the GaussSieve-based HashSieve, where such list updates are allowed if they produce shorter lattice vectors. For progressive sieving, the execution times of the GaussSieve and ListSieve variants are much closer, which suggests using ListSieve-like sieving may be practical as well. Note that the ListSieve does not save any inner product computations between sampled and list vectors, since we already need to compute $\boldsymbol{v} \cdot \boldsymbol{w}$ to see if $\boldsymbol{v}$ can be reduced with $\boldsymbol{w}$ in Line 5 (before reducing $\boldsymbol{w}$ with $\boldsymbol{v}$ in Line 6), and so the performance is still worse when doing the ListSieve instead of the GaussSieve: skipping potential reductions which are "free" is almost never a wise choice.

A potential application of ListSieve-style sieving, knowing that its performance is closer to the GaussSieve when using progressive sieving, is in parallel implementations. For shared memory systems, current approaches use (probable) lock-free lists [MTB14, MLB15, MLB17], and these lock-free lists may no longer be needed when we know that once a vector is added to the list, it is never updated again and it becomes read-only memory. For distributed-memory implementations [IKMT14, BNvdP16, YKYC17], using a ListSieve as a baseline

[4] Concurrent work [Duc18] suggests that ideas similar to progressive sieving only lead to a factor 5× speed-up, i.e. roughly a factor 4× less than described here. We conjecture that this difference is mainly caused by Ducas using a more optimized implementation of the baseline approach, and in particular using a better sampler.

may also be beneficial in terms of performance and simplicity – synchronization between nodes only has to be done on local sampled vectors, and not on list vectors which are modified by different nodes.

6 Open problems

As we have seen, progressive sieving has a better (experimental) performance than current sieving approaches in many ways, making sieving slightly smoother, more predictable, less dependent on parameter choices for the sampler, and more dependent on the quality of the input basis. Below we state some open problems for future work, related to the ideas presented in this paper.

6.1 BKZ and sieving on sliding windows

A long-standing open problem is to efficiently implement sieving as the SVP subroutine for BKZ, which now commonly uses enumeration as its SVP subroutine instead. Only recently has sieving become rather competitive with enumeration, and projects in this direction are likely ongoing.

When running BKZ on d-dimensional bases, there is often a sliding window of k indices $i+1, \dots, i+k$, for which SVP in k dimensions needs to be solved to form an HKZ-reduced basis in this block. Then, when this block has been properly reduced, the index i is increased by 1, and the same procedure is applied to the window $i+2, \dots, i+k+1$. Since there is a large overlap between different windows, a natural question is whether dealing with this new block can be done more efficiently than starting from scratch.

This is somewhat similar to progressive sieving, where as an abstraction we start with k basis vectors $\boldsymbol{b}_1, \dots, \boldsymbol{b}_k$, and after reducing this basis (running sieving on this block), we switch to a basis $\pi_1(\boldsymbol{b}_2), \dots, \pi_1(\boldsymbol{b}_{k+1})$, with $\pi_1(\boldsymbol{x})$ being the projection of $\boldsymbol{x}$ orthogonal to $\boldsymbol{b}_1$, and $\boldsymbol{b}_{k+1}$ being linearly independent of the previous basis vectors. So (besides the projections) not only do we add a new basis vector $\boldsymbol{b}_{k+1}$ to the system as in progressive sieving, we also remove one vector $\boldsymbol{b}_1$, making all vectors currently in our list with a non-zero coefficient of $\boldsymbol{b}_1$ "unusable" in the next iteration.

One potential way of dealing with shifting windows in sieving is to discard all vectors $\boldsymbol{w}$ with non-zero coefficient of $\boldsymbol{b}_1$ when moving from one window to the next – the contribution of $\boldsymbol{b}_1$ may be crucial for this vector to be short, and removing this contribution may result in a long lattice vector. By the GSA, the number of vectors of norm less than $\frac{4}{3} \cdot \lambda_1(\mathcal{L}(\boldsymbol{b}_1, \dots, \boldsymbol{b}_k))$ in $\mathcal{L}(\boldsymbol{b}_1, \dots, \boldsymbol{b}_k)$ is approximately $[\frac{4}{3} \cdot \lambda_1(\mathcal{L}(\boldsymbol{b}_1, \dots, \boldsymbol{b}_k))/\lambda_1(\mathcal{L}(\boldsymbol{b}_2, \dots, \boldsymbol{b}_k))]^d$, which is roughly a fraction $[\lambda_1(\mathcal{L}(\boldsymbol{b}_1, \dots, \boldsymbol{b}_k))/\lambda_1(\mathcal{L}(\boldsymbol{b}_2, \dots, \boldsymbol{b}_k))]^d$ of all vectors in the sieved list $L \subset \mathcal{L}(\boldsymbol{b}_1, \dots, \boldsymbol{b}_k)$. Although this fraction may be small depending on the shape of the lattice and the block size k, a non-negligible fraction of vectors can thus be reused in the next iteration without difficulties.

Besides this straightforward approach, one could also imagine vectors with non-zero coefficients of $\boldsymbol{b}_1$ being reused in the next iteration by just removing this

contribution of $\boldsymbol{b}_1$, and hoping that the resulting vector is still short. This might make the vectors slightly longer on average, but makes sure that all vectors can be reused, potentially saving even more work on the sieve in the next window.

6.2 Enumeration with sieving

Another long-standing open problem in lattice algorithms is to consider approaches based on enumeration and sieving, and combining the "best of both worlds" to construct even better SVP algorithms. As suggested in [Laa16], one potential such combination would consist of running sieving on a subset of the basis (say $\boldsymbol{b}_1, \dots, \boldsymbol{b}_k$), and then using the sieved list as an approximate Voronoi cell for faster enumeration. This enumeration procedure would then consist of considering combinations of the vectors $\boldsymbol{b}_{k+1}, \dots, \boldsymbol{b}_d$ as in state-of-the-art enumeration algorithms, and then using the sieved list $L \subset \mathcal{L}_k$ to see whether those enumerated vectors are close to a vector in the sublattice $\mathcal{L}_k$. If it is close to such a vector, the difference with that vector is likely a short vector in the full lattice. In this case sieving would be used as a batch-CVP/CVPP oracle.

While this idea also works with classical sieving methods, it becomes even more natural with progressive sieving, which already considers increasingly large sublattices to make progress. Modifying progressive sieving to the above application would simply mean changing the upper bound in Line 16 from the full rank d to some bound d_0. Similar ideas of combining sieving on a sublattice with enumerating the "last few dimensions" have been studied in [Duc18], but more work is needed to understand the full potential of such a hybrid approach.

Acknowledgments

The authors thank Léo Ducas for discussions and comments on this topic, and for sharing an early draft of [Duc18]. The first author is supported by the ERC consolidator grant 617951. The second author was partially supported by Fundação para a Ciência e a Tecnologia (FCT) and Instituto de Telecomunicações under grant UID/EEA/50008/2013.

References

ADPS16. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *USENIX Security Symposium*, pages 327–343, 2016.

AKS01. Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *STOC*, pages 601–610, 2001.

AN17. Yoshinori Aono and Phong Q. Nguyên. Random sampling revisited: lattice enumeration with discrete pruning. In *EUROCRYPT*, 2017.

BDGL16. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA*, pages 10–24, 2016.

BDK+18. Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. In *EuroS&P*, 2018.

BGJ14. Anja Becker, Nicolas Gama, and Antoine Joux. A sieve algorithm based on overlattices. In *ANTS*, pages 49–70, 2014.

BGJ15. Anja Becker, Nicolas Gama, and Antoine Joux. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. *Cryptology ePrint Archive, Report 2015/522*, pages 1–14, 2015.

BL16. Anja Becker and Thijs Laarhoven. Efficient (ideal) lattice sieving using cross-polytope LSH. In *AFRICACRYPT*, pages 3–23, 2016.

BLS16. Shi Bai, Thijs Laarhoven, and Damien Stehlé. Tuple lattice sieving. In *ANTS*, pages 146–162, 2016.

BNvdP16. Joppe W. Bos, Michael Naehrig, and Joop van de Pol. Sieving for shortest vectors in ideal lattices: a practical perspective. *International Journal of Applied Cryptography*, pages 1–23, 2016.

CN11. Yuanmi Chen and Phong Q. Nguyên. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, pages 1–20, 2011.

DLL+18. Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Dilithium: Digital signatures from module lattices. In *CHES*, 2018.

Duc18. Léo Ducas. Shortest vector from lattice sieving: a few dimensions for free. In *EUROCRYPT*, 2018.

FBB+14. Robert Fitzpatrick, Christian Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang. Tuning GaussSieve for speed. In *LATINCRYPT*, pages 288–305, 2014.

FP85. Ulrich Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice. *Mathematics of Computation*, 44(170):463–471, 1985.

fplll18. The FPLLL development team. fplll, a lattice reduction library. Available at `https://github.com/fplll/fplll`, 2016.

GNR10. Nicolas Gama, Phong Q. Nguyên, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, pages 257–278, 2010.

HK17. Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate k-list problem in Euclidean norm. In *PKC*, pages 327–343, 2017.

HKL18. Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven. Speed-ups and time-memory trade-offs for tuple lattice sieving. In *PKC*, 2018.

HPS11. Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the shortest and closest lattice vector problems. In *IWCC*, pages 159–190, 2011.

IKMT14. Tsukasa Ishiguro, Shinsaku Kiyomoto, Yutaka Miyake, and Tsuyoshi Takagi. Parallel Gauss Sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice. In *PKC*, pages 411–428, 2014.

Kan83. Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *STOC*, pages 193–206, 1983.

Kle00. Philip Klein. Finding the closest lattice vector when it's unusually close. In *SODA*, pages 937–941, 2000.

Laa15. Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *CRYPTO*, pages 3–22, 2015.

Laa16. Thijs Laarhoven. Finding closest lattice vectors using approximate Voronoi cells. *Cryptology ePrint Archive, Report 2016/888*, 2016.

LdW15. Thijs Laarhoven and Benne de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In *LATINCRYPT*, pages 101–118, 2015.

MB16. Artur Mariano and Christian Bischof. Enhancing the scalability and memory usage of HashSieve on multi-core CPUs. In *PDP*, pages 545–552, 2016.

MLB15. Artur Mariano, Thijs Laarhoven, and Christian Bischof. Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In *ICPP*, pages 590–599, 2015.

MLB17. Artur Mariano, Thijs Laarhoven, and Christian Bischof. A parallel variant of LDSieve for the SVP on lattices. *PDP*, 2017.

MLC^{+}17. Artur Mariano, Thijs Laarhoven, Fábio Correia, Manuel Rodrigues, and Gabriel Falcao. A practical view of the state-of-the-art of lattice-based cryptanalysis. *IEEE Access*, 5:24184–24202, 2017.

MODB14. Artur Mariano, Özgür Dagdelen, and Christian Bischof. A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In *Euro-Par 2014*, pages 48–59, 2014.

MS11. Benjamin Milde and Michael Schneider. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In *PACT*, pages 452–458, 2011.

MTB14. Artur Mariano, Shahar Timnat, and Christian Bischof. Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. In *SBAC-PAD*, pages 278–285, 2014.

MV10a. Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations. In *STOC*, pages 351–358, 2010.

MV10b. Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA*, pages 1468–1480, 2010.

NV08. Phong Q. Nguyên and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.

Sch87. Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2–3):201–224, 1987.

Sch11. Michael Schneider. Analysis of Gauss-Sieve for solving the shortest vector problem in lattices. In *WALCOM*, pages 89–97, 2011.

Sch13. Michael Schneider. Sieving for short vectors in ideal lattices. In *AFRICACRYPT*, pages 375–391, 2013.

SE94. Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(2–3):181–199, 1994.

SFS09. Naftali Sommer, Meir Feder, and Ofir Shalvi. Finding the closest lattice point by iterative slicing. *SIAM Journal of Discrete Mathematics*, 23(2):715–731, 2009.

SVP18. SVP challenge, 2018. `https://latticechallenge.org/svp-challenge/`.

WLTB11. Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In *ASIACCS*, pages 1–9, 2011.

YKYC17. Shang-Yi Yang, Po-Chun Kuo, Bo-Yin Yang, and Chen-Mou Cheng. Gauss sieve algorithm on GPUs. In *CT-RSA*, pages 39–57, 2017.

ZPH13. Feng Zhang, Yanbin Pan, and Gengran Hu. A three-level sieve algorithm for the shortest vector problem. In *SAC*, pages 29–47, 2013.