Finding closest lattice vectors using approximate Voronoi cells

Emmanouil Doulgerakis, Thijs Laarhoven, and Benne de Weger

Eindhoven University of Technology
Eindhoven, The Netherlands
{e.doulgerakis,b.m.m.d.weger}@tue.nl, mail@thijs.com

Abstract. The two traditional hard problems underlying the security of lattice-based cryptography are the shortest vector problem (SVP) and the closest vector problem (CVP). For a long time, lattice enumeration was considered the fastest method for solving these problems in high dimensions, but recent work on memory-intensive methods has resulted in lattice sieving overtaking enumeration both in theory and in practice. Some of the recent improvements [Ducas, Eurocrypt 2018; Laarhoven–Mariano, PQCrypto 2018; Albrecht–Ducas–Herold–Kirshanova–Postlethwaite–Stevens, Eurocrypt 2019] are based on the fact that these methods find more than just one short lattice vector, and this additional data can be reused effectively later on to solve other, closely related problems faster. Similarly, results for the preprocessing version of CVP (CVPP) have demonstrated that once this initial data has been generated, instances of CVP can be solved faster than when solving them directly, albeit with worse memory complexities [Laarhoven, SAC 2016].
In this work we study CVPP in terms of approximate Voronoi cells, and obtain better time and space complexities using randomized slicing, which is similar in spirit to using randomized bases in lattice enumeration [Gama–Nguyen–Regev, Eurocrypt 2010]. With this approach, we improve upon the state-of-the-art complexities for CVPP, both theoretically and experimentally, with a practical speedup of several orders of magnitude compared to non-preprocessed SVP or CVP. Such a fast CVPP solver may give rise to faster enumeration methods, where the CVPP solver is used to replace the bottom part of the enumeration tree, consisting of a batch of CVP instances in the same lattice.
Asymptotically, we further show that we can solve an exponential number of instances of CVP in a lattice in essentially the same amount of time and space as the fastest method for solving just one CVP instance. This is in line with various recent results, showing that perhaps the biggest strength of memory-intensive methods lies in being able to reuse the generated data several times. Similar to [Ducas, Eurocrypt 2018], this further means that we can achieve a "few dimensions for free" for sieving for SVP or CVP, by doing $\Theta(d/\log d)$ levels of enumeration on top of a CVPP solver based on approximate Voronoi cells.

Keywords: lattices, preprocessing, Voronoi cells, sieving algorithms, shortest vector problem (SVP), closest vector problem (CVP)

1 Introduction

Lattice problems. Lattices are discrete subgroups of $\mathbb{R}^d$: given a basis $\boldsymbol{B} = \{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_d\} \subset \mathbb{R}^d$, the lattice generated by $\boldsymbol{B}$ is defined as $\mathcal{L} = \mathcal{L}(\boldsymbol{B}) := \{\sum_{i=1}^{d} \lambda_i \boldsymbol{b}_i : \lambda_i \in \mathbb{Z}\}$. Given a basis of $\mathcal{L}$, the shortest vector problem (SVP) is to find a (non-zero) lattice vector $\boldsymbol{s}$ of Euclidean norm $\|\boldsymbol{s}\| = \lambda_1(\mathcal{L}) := \min_{\boldsymbol{v} \in \mathcal{L} \setminus \{\boldsymbol{0}\}} \|\boldsymbol{v}\|$. Given a basis of a lattice and a target vector $\boldsymbol{t} \in \mathbb{R}^d$, the closest vector problem (CVP) is to find a lattice vector $\boldsymbol{s} \in \mathcal{L}$ closest to $\boldsymbol{t}$. The preprocessing variant of CVP (CVPP) asks to preprocess the lattice $\mathcal{L}$ such that, when later given a target vector $\boldsymbol{t}$, one can quickly find a closest lattice vector to $\boldsymbol{t}$.

SVP and CVP are fundamental in the study of lattice-based cryptography, as the security of many schemes is directly related to their hardness. Various other hard lattice problems, such as Learning With Errors (LWE), are closely related to SVP and CVP; see, e.g., [63,74,75] for reductions among lattice problems. These reductions show that understanding the hardness of SVP and CVP is crucial for accurately estimating the security of lattice-based cryptographic schemes.

1.1 Related work

Worst-case SVP/CVP analyses. Although SVP and CVP are both central in the study of lattice-based cryptography, algorithms for SVP have received somewhat more attention, including a benchmarking website to compare different methods [1]. Various SVP algorithms have been studied which can solve CVP as well, such as the polynomial-space, superexponential-time lattice enumeration studied in [14, 32, 38, 40, 47, 66]. More recently, methods have been proposed which solve SVP/CVP in only single exponential time, but which also require exponential-sized memory [2, 6, 64]. By constructing the Voronoi cell of the lattice [4,25,64,73], Micciancio–Voulgaris showed that SVP and CVP(P) can provably be solved in time $2^{2d+o(d)}$, and Bonifas–Dadush reduced the complexity for CVPP to only $2^{d+o(d)}$. In high dimensions the best provable complexities for SVP and CVP are currently due to discrete Gaussian sampling [2, 3], solving both problems in $2^{d+o(d)}$ time and space in the worst case on arbitrary lattices.

Average-case SVP/CVP algorithms. When considering and comparing these methods in practice on random lattices, we get a completely different picture. Currently the fastest heuristic methods for SVP and CVP in high dimensions are based on lattice sieving. After a long series of theoretical works on constructing efficient heuristic sieving algorithms [18–21, 50, 53, 65, 68, 78, 80] as well as applied papers studying how to further speed up these algorithms in practice [28, 35, 39, 46, 54, 57–61, 67, 71, 72], the best heuristic time complexity for solving SVP (and CVP [52]) currently stands at $2^{0.292d+o(d)}$ [18, 59], using $2^{0.208d+o(d)}$ memory. The highest records in the SVP challenge [1] were recently obtained using a BKZ-sieving hybrid [7]. These recent improvements have resulted in a major shift in security estimates for lattice-based cryptography, from estimating the hardness of SVP/CVP using the best enumeration methods, to estimating this hardness based on state-of-the-art sieving results [9,24,26,27,36].

Hybrid algorithms and batch-CVP. In moderate dimensions, enumeration-based methods dominated for a long time, and the cross-over point with single-exponential time algorithms like sieving seemed to be far out of reach [66]. Moreover, the exponential memory of, e.g., lattice sieving will ultimately also significantly slow down these algorithms due to the large number of random memory accesses [23], and parallelizing sieving efficiently is less trivial than parallelizing enumeration [7, 23, 28, 46, 59, 67, 79]. Some previous work focused on obtaining a trade-off between enumeration and sieving, using less memory for sieving [17, 43, 44] or using more memory for enumeration [48].

Another well-known direction for a hybrid between memory-intensive methods and enumeration is to use a fast CVP(P) algorithm as a subroutine within enumeration. As described in, e.g., [40,66], at any given level in the enumeration tree, one is attempting to solve a CVP instance in a lower-rank sublattice, where the target vector is determined by the path from the root to the current node in the tree. Each node at this level in the tree corresponds to a CVP instance in the same sublattice, but with a different target. If we can preprocess this low-dimensional sublattice such that the amortized time complexity of solving a batch of CVP-instances in this sublattice is small, then this may speed up processing the bottom part of the enumeration tree.

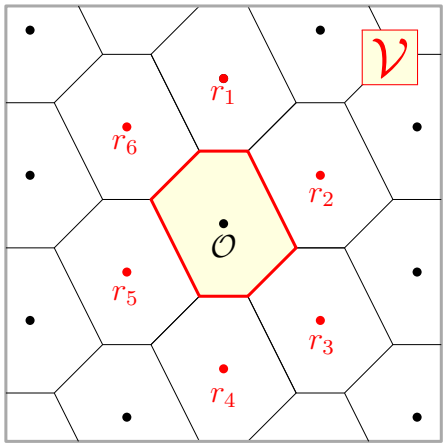
A first step in this direction was taken in [52], where it was shown that with a sufficient amount of preprocessing and space, one can achieve better amortized time complexities for batch-CVP than when solving just one instance. The large memory requirement (at least $2^{d/2+o(d)}$ memory is required to improve upon direct CVP approaches) as well as the large number of CVP instances required to get a lower amortized complexity made this approach impractical to date.

1.2 Contributions: Approximate Voronoi cells

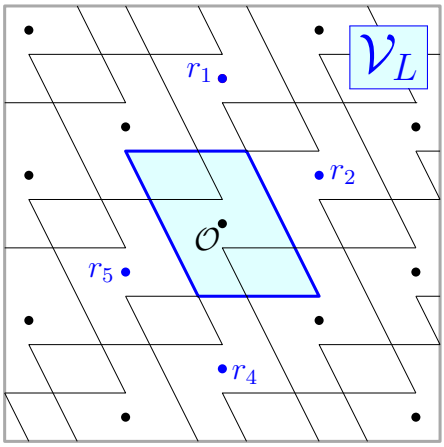
In this paper we revisit the preprocessing approach to CVP of [52], as well as the recent trend of speeding up these algorithms using nearest neighbor searching, and we show how to obtain significantly improved time and space complexities. These results can be viewed as a first step towards a practical, heuristic alternative to the Voronoi cell approach of Micciancio–Voulgaris [66], where instead of constructing the exact Voronoi cell, the preprocessing computes an approximation of it, requiring less time and space to compute and store.

First, our preprocessing step consists of computing a list L of most lattice vectors below a given norm.[1] This preprocessing can be done using either enumeration or sieving. The preprocessed data can best be understood as representing an *approximate* Voronoi cell $\mathcal{V}_L$ of the lattice, where the size of L determines how well $\mathcal{V}_L$ approximates the true Voronoi cell $\mathcal{V}$ of the lattice; see Figure 1 for an example. Using this approximate Voronoi cell, we then attempt to solve CVP instances by applying the iterative slicing procedure of Sommer–Feder–Shalvi [73], with nearest neighbor optimizations to reduce the search costs [12, 18].

[1] Heuristically, finding a large fraction of all lattice vectors below a given norm will suffice – one does not necessarily need to run a deterministic preprocessing algorithm to ensure all short lattice vectors are found.



(a) A tiling of $\mathbb{R}^2$ with exact Voronoi cells $\mathcal{V}$ of a lattice $\mathcal{L}$ (red/black points), generated by the set $\mathcal{R} = \{r_1, \ldots, r_6\}$ of all *relevant vectors* of $\mathcal{L}$. Here $\mathrm{vol}(\mathcal{V}) = \det(\mathcal{L})$.



(b) An overlapping tiling of $\mathbb{R}^2$ with approximate Voronoi cells $\mathcal{V}_L$ of the same lattice $\mathcal{L}$, generated by a subset of the relevant vectors, $L = \{r_1, r_2, r_4, r_5\} \subset \mathcal{R}$.

Fig. 1. Exact and approximate Voronoi cells of the same two-dimensional lattice $\mathcal{L}$. For the **exact** Voronoi cell $\mathcal{V}$ (Figure 1a), the cells around the lattice points form a tiling of $\mathbb{R}^2$, covering each point in space exactly once. Given that a point $\boldsymbol{t}$ lies in the Voronoi cell around $\boldsymbol{s} \in \mathcal{L}$, we know that $\boldsymbol{s}$ is the closest lattice point to $\boldsymbol{t}$.
For the **approximate** Voronoi cell $\mathcal{V}_L$ (Figure 1b), the cells around the lattice points overlap, and cover a non-empty fraction of the space by multiple cells. Given that a vector $\boldsymbol{t}$ lies in an approximate Voronoi cell around a lattice point $\boldsymbol{s}$, we further do not have the definite guarantee that $\boldsymbol{s}$ is the closest lattice point to $\boldsymbol{t}$.

The main difference in our work over [52] lies in generalizing how similar $\mathcal{V}_L$ (generated by the list L) needs to be to $\mathcal{V}$. We distinguish two cases below. As sketched in Figure 1, a worse approximation leads to a larger approximate Voronoi cell, so $\mathrm{vol}(\mathcal{V}_L) \geq \mathrm{vol}(\mathcal{V})$ with equality iff $\mathcal{V} = \mathcal{V}_L$.

Good approximations: If $\mathcal{V}_L$ is a good approximation of $\mathcal{V}$ (i.e., $\mathrm{vol}(\mathcal{V}_L) \approx \mathrm{vol}(\mathcal{V})$), then with high probability over the randomness of the target vectors, the iterative slicer returns the closest lattice vector to random targets. To guarantee $\mathrm{vol}(\mathcal{V}_L) \approx \mathrm{vol}(\mathcal{V})$ we need $|L| \geq 2^{d/2+o(d)}$, where additional memory can be used to speed up the nearest neighbor part of the iterative slicer. The resulting query complexities are sketched in red in Figure 2.

Arbitrary approximations: If the preprocessed list contains fewer than $2^{d/2}$ vectors, then $\mathrm{vol}(\mathcal{V}_L) \gg \mathrm{vol}(\mathcal{V})$ and with overwhelming probability the iterative slicer will not return the closest lattice point to a random target vector. However, similar to [40], the running time of this method is decreased by a much more significant factor than the success probability. So if we are able to *rerandomize* the problem instance and try several times, we may still be faster (and more memory-efficient) than when using a larger list L.

1.3 Contributions: Randomized slicing

To actually find solutions to CVP instances with a "bad" approximation $\mathcal{V}_L$ to the real Voronoi cell $\mathcal{V}$, we need to be able to suitably rerandomize the iterative slicing procedure, so that if the success probability in a single run of the slicer is small, we can repeat the method several times for a high success probability. To do this, we will run the iterative slicer on randomly perturbed vectors $\boldsymbol{t}' \sim D_{\boldsymbol{t}+\mathcal{L},s}$, sampled from a discrete Gaussian over the coset $\boldsymbol{t}+\mathcal{L}$. Here the standard deviation s needs to be sufficiently large to make sampling from $D_{\boldsymbol{t}+\mathcal{L},s}$ efficient and the results of the slicer to be almost independent, and s needs to be sufficiently small to guarantee that the slicer will terminate in a limited number of steps. Algorithm 1 explicitly describes this procedure, given as input an approximate Voronoi cell $\mathcal{V}_L$ (i.e., a list $L \subset \mathcal{L}$ of short lattice vectors defining the facets of this approximate Voronoi cell).

Algorithm 1 The randomized heuristic slicer for finding closest vectors

Require: A list $L \subset \mathcal{L}$ and a target $\boldsymbol{t} \in \mathbb{R}^d$
Ensure: The algorithm outputs a closest lattice vector $\boldsymbol{s} \in \mathcal{L}$ to $\boldsymbol{t}$

$\boldsymbol{s} \leftarrow \boldsymbol{0}$ ▷ Initial guess $\boldsymbol{s}$ for closest vector to $\boldsymbol{t}$
repeat
  Sample $\boldsymbol{t}' \sim D_{\boldsymbol{t}+\mathcal{L},s}$ ▷ Randomly shift $\boldsymbol{t}$ by a vector $\boldsymbol{v} \in \mathcal{L}$
  for each $\boldsymbol{r} \in L$ **do**
    if $\|\boldsymbol{t}'-\boldsymbol{r}\| < \|\boldsymbol{t}'\|$ **then** ▷ New shorter vector $\boldsymbol{t}' \in \boldsymbol{t}+\mathcal{L}$
      Replace $\boldsymbol{t}' \leftarrow \boldsymbol{t}'-\boldsymbol{r}$ and restart the **for**-loop
  if $\|\boldsymbol{t}'\| < \|\boldsymbol{t}-\boldsymbol{s}\|$ **then**
    $\boldsymbol{s} \leftarrow \boldsymbol{t}-\boldsymbol{t}'$ ▷ New lattice vector $\boldsymbol{s}$ closer to $\boldsymbol{t}$
until $\boldsymbol{s}$ is a closest lattice vector to $\boldsymbol{t}$
return $\boldsymbol{s}$

Even though this algorithm requires sampling many vectors from the coset $\boldsymbol{t}+\mathcal{L}$ and running the iterative slicer on all of these, the overall time complexity of this procedure will still be lower, since the iterative slicer needs less time to complete when the input list L is shorter. To estimate the number of iterations necessary to guarantee that the algorithm returns the actual closest vector, we make the following assumption, stating that the probability that the iterative slicer terminates with a vector $\boldsymbol{t}' \in (\boldsymbol{t}+\mathcal{L}) \cap \mathcal{V}$, given that it must terminate to some vector $\boldsymbol{t}' \in (\boldsymbol{t}+\mathcal{L}) \cap \mathcal{V}_L$, is proportional to the ratio of the volumes of these (approximate) Voronoi cells $\mathcal{V}$ and $\mathcal{V}_L$.

Heuristic assumption 1 (Randomized slicing) *For $L \subset \mathcal{L}$ and large s,*

$$\Pr_{\boldsymbol{t}' \sim D_{\boldsymbol{t}+\mathcal{L},s}}\left[\mathrm{Slice}_L(\boldsymbol{t}') \in \mathcal{V}\right] \approx \frac{\mathrm{vol}(\mathcal{V})}{\mathrm{vol}(\mathcal{V}_L)}\,. \tag{1}$$

This is a new and critical assumption to guarantee that the claimed asymptotic complexities are correct, and we will therefore come back to this assumption later on, to show that experiments indeed suggest this assumption is justified.

1.4 Contributions: Improved CVPP complexities

For the exact closest vector problem with preprocessing, our improved complexities over [52] mainly come from the aforementioned randomizations. To illustrate this with a simple example, suppose we run an optimized (GaussSieve-based [65]) LDSieve [18], ultimately resulting in a list of $(4/3)^{d/2+o(d)}$ of the shortest vectors in the lattice, indexed in a nearest neighbor data structure of size $(3/2)^{d/2+o(d)}$. Asymptotically, using this list as our approximate Voronoi cell, the iterative slicer succeeds with probability $p = (13/16)^{d/2+o(d)}$ (as shown in the analysis later on), while processing a query with this data structure takes time $(9/8)^{d/2+o(d)}$. By repeating a query $1/p$ times with rerandomizations of the same CVP instance, we obtain the following heuristic complexities for CVPP.

Proposition 1 (Standard sieve preprocessing). *Using the output of the LDSieve [18] as the preprocessed list and encompassing data structure, we can heuristically solve CVPP with the following query space and time complexities:*

$$\mathrm{S} = (3/2)^{d/2+o(d)} \approx 2^{0.292d+o(d)}, \qquad \mathrm{T} = (18/13)^{d/2+o(d)} \approx 2^{0.235d+o(d)}.$$

This point (S, T) *is highlighted in light blue in Figure 2.*

If we use a more general analysis of the approximate Voronoi cell approach, varying over both the nearest neighbor parameters and the size of the preprocessed list, we can obtain even better query complexities. For a memory complexity of $(3/2)^{d/2+o(d)} \approx 2^{0.292d+o(d)}$, we can achieve a query time complexity of approximately $2^{0.220d+o(d)}$ by using a shorter list of lattice vectors, and a more memory-intensive parameter setting for the nearest neighbor data structure. The following main result summarizes all the asymptotic time–space trade-offs we can obtain for heuristically solving CVPP in the average case.

Theorem 1 (Optimized CVPP complexities). *Let* $\alpha \in (1.03396, \sqrt{2})$ *and* $u \in (\sqrt{\frac{\alpha^2-1}{\alpha^2}}, \sqrt{\frac{\alpha^2}{\alpha^2-1}})$. *With approximate Voronoi cells we can heuristically solve CVPP with preprocessing space and time* S_1 *and* T_1*, and query space and time* S_2 *and* T_2*, where:*

$$\mathrm{S}_1 = \max\left\{\mathrm{S}_2, \left(\frac{4}{3}\right)^{d/2+o(d)}\right\}, \qquad \mathrm{T}_1 = \max\left\{\mathrm{S}_2, \left(\frac{3}{2}\right)^{d/2+o(d)}\right\}, \tag{2}$$

$$\mathrm{S}_2 = \left(\frac{\alpha}{\alpha - (\alpha^2-1)(\alpha u^2 - 2u\sqrt{\alpha^2-1} + \alpha)}\right)^{d/2+o(d)}, \tag{3}$$

$$\mathrm{T}_2 = \left(\frac{16\alpha^4\left(\alpha^2-1\right)}{-9\alpha^8+64\alpha^6-104\alpha^4+64\alpha^2-16} \cdot \frac{\alpha + u\sqrt{\alpha^2-1}}{-\alpha^3 + \alpha^2 u\sqrt{\alpha^2-1} + 2\alpha}\right)^{d/2+o(d)}. \tag{4}$$

The best query complexities $(\mathrm{S}_2, \mathrm{T}_2)$ *together form the blue curve in Figure 2.*

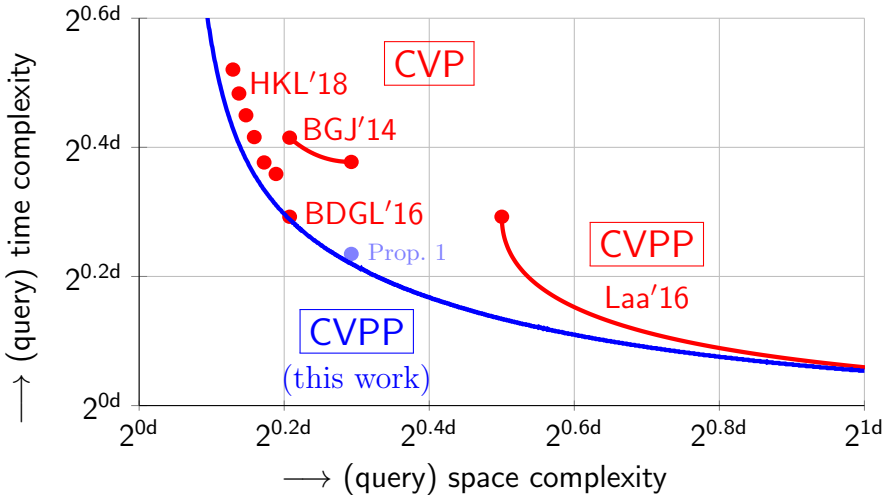


Fig. 2. Query complexities for finding closest vectors, directly (CVP) and with preprocessing (CVPP). The leftmost red points/curve show the best asymptotic SVP/CVP complexities of Becker–Gama–Joux [19], Becker–Ducas–Gama–Laarhoven [18], and Herold–Kirshanova–Laarhoven [44]. The rightmost red point and curve are the previous best CVPP complexities of [52]. The blue curve shows our new CVPP complexities.

Compared to [52], we obtain trade-offs for much lower memory complexities, and we improve upon both the best CVPP complexities of [52] and the best SVP/CVP complexities of [18, 44].[2] Observe that our trade-off passes *below* all the best CVP results, i.e., we can always solve an exponentially large batch of $2^{\varepsilon d}$ CVP instances for small $\varepsilon > 0$ in the same amount of time as the current best complexities for solving just one instance, for any memory bound.

Due to the condition that $\alpha > 1.0339\ldots$ (which follows from the fact that the denominator in T_2 needs to remain positive), the blue curve in Figure 2 terminates on the left side at a minimum query space complexity of $1.03396^{d+o(d)} \approx 2^{0.0482d+o(d)}$. One might wonder whether we can obtain a continuous trade-off between the query time and space complexities reaching all the way to $2^{o(d)}$ memory and $2^{\omega(d)}$ query time. The lower bound on α might be a consequence of our analysis, and perhaps a different approach would show this algorithm solves CVPP in $2^{O(d)}$ time even with less memory.

As for the other end of the blue curve, as the available space increases, one can achieve an amortized time complexity for CVP of $2^{\varepsilon d+o(d)}$ at the cost of $(1/\varepsilon)^{O(d)}$ preprocessed space for arbitrary $\varepsilon > 0$. For large query space complexities, i.e., when a lot of memory and preprocessing power is available for speeding up the queries, the blue and red curve converge, and the best parameter choice is to set $\alpha \approx \sqrt{2}$ such that $\mathcal{V}_L \approx \mathcal{V}$, as explained in Section 1.2.

[2] As detailed in [52], by modifying sieve algorithms for SVP, one can also solve CVP with essentially equivalent heuristic time and space complexities as for SVP.

Concrete complexities. Although Theorem 1 and Figure 2 illustrate how well we expect these methods to scale in high dimensions d, we would like to stress that Theorem 1 is a purely asymptotic result, with potentially large order terms hidden by the $o(d)$ in the exponents for the time and space complexities. To obtain security estimates for real-world applications, and to assess how fast this algorithm actually solves problems appearing in the cryptanalysis of lattice-based cryptosystems, it therefore remains necessary to perform extensive experiments, and to cautiously try to extrapolate from these results what the real attack costs might be for high dimensions d, necessary to attack actual instantiations of cryptosystems. Later on we will describe some preliminary experiments we performed to test the practicality of this approach, but further work is still necessary to assess the impact of these results on the concrete hardness of CVPP.

1.5 High-level proof description

To prove the main results regarding the improved asymptotic CVPP complexities compared to [52], we first prove that under certain natural heuristic assumptions, we obtain the following upper bound on the volume of approximate Voronoi cells generated by the $\alpha^{d+o(d)}$ shortest vectors of a lattice. The preprocessing will consist of exactly this: generate the $\alpha^{d+o(d)}$ shortest vectors in the lattice, and store them in a nearest neighbor data structure that allows for fast look-ups of nearby points in space.

Lemma 1 (Relative volume of approximate Voronoi cells). *Let $L \subset \mathcal{L}$ consist of the $\alpha^{d+o(d)}$ shortest vectors of a lattice $\mathcal{L}$, with $\alpha \in (1.03396, \sqrt{2})$. Then heuristically,*

$$\frac{\mathrm{vol}(\mathcal{V}_L)}{\mathrm{vol}(\mathcal{V})} \leq \left(\frac{16\alpha^4 \left(\alpha^2 - 1\right)}{-9\alpha^8 + 64\alpha^6 - 104\alpha^4 + 64\alpha^2 - 16} \right)^{d/2+o(d)} . \tag{5}$$

Using this lemma and the heuristic assumption stated previously, relating the success probability of the slicer to the volume of the approximate Voronoi cell, this immediately gives us a (heuristic) lower bound on the success probability p_α of the randomized slicing procedure, given as input a preprocessed list of the $\alpha^{d+o(d)}$ shortest vectors in the lattice. Then, similar to [52], the complexity analysis is a matter of combining the costs for the preprocessing phase, the costs of the nearest neighbor data structure, and the cost of the query phase, where now we need to repeat the randomized slicing of the order $1/p_\alpha$ times – the difference in the formulas for the complexities compared to [52] comes exactly from this additional factor $1/p_\alpha \approx \mathrm{vol}(\mathcal{V}_L)/\mathrm{vol}(\mathcal{V})$.

To prove the above lemma regarding the volume of approximate Voronoi cells, we will prove the following statements. First, we show that if the list L contains the $\alpha^{d+o(d)}$ shortest vectors of a random lattice $\mathcal{L}$, then on input a target vector $\boldsymbol{t}$, we heuristically expect the slicer to terminate on a reduced vector $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$ of norm at most $\|\boldsymbol{t}'\| \leq \beta \cdot \lambda_1(\mathcal{L})$, where β is determined by the parameter α. The

relation between α and β can be succinctly described by the following relation

$$\beta = \alpha^2/\sqrt{4\alpha^2 - 4}. \tag{6}$$

More precisely, we show that as long as $\|\boldsymbol{t}'\| \gg \beta \cdot \lambda_1(\mathcal{L})$, then with high probability we expect to be able to combine $\boldsymbol{t}'$ with vectors in L to form a shorter vector $\boldsymbol{t}'' \in \boldsymbol{t} + \mathcal{L}$ with $\|\boldsymbol{t}''\| < \|\boldsymbol{t}'\|$. On the other hand, if we have a vector $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$ of norm less than $\beta \cdot \lambda_1(\mathcal{L})$, then we only expect to be able to combine $\boldsymbol{t}'$ with a vector in L to form a shorter vector with exponentially small probability $2^{-\Theta(d)}$. In other words, reducing to a vector of norm $\beta \cdot \lambda_1(\mathcal{L})$ can be done almost "effortlessly", while after that even making small progress in reducing the length of $\boldsymbol{t}'$ comes at an exponential loss in the success probability.

Good approximations. Next, from the above relation between the size of the input list, $|L|$ (or α), and the reduced norm of the shifted target vector, $\|\boldsymbol{t}'\|$ (or β), the previous result of [52] immediately follows – to achieve $\boldsymbol{t}' \in \mathcal{V}$ we heuristically need $\beta = 1 + o(1)$. This implies that $\alpha = \sqrt{2}$ is the minimal parameter that guarantees we will be able to effortlessly reduce to the exact Voronoi cell, and so L must contain the $\alpha^{d+o(d)} = 2^{d/2+o(d)}$ shortest vectors in the lattice. In that case the success probability is constant, and the costs of the query phase are determined by a single reduction of $\boldsymbol{t}$ with the iterative slicer.

Arbitrary approximations. However, even if $\alpha < \sqrt{2}$ is smaller, and the corresponding β is therefore larger than 1, the slicer might still succeed with (exponentially) small probability. To analyze the success probability, note that from the Gaussian heuristic we may assume that the closest vector to our target $\boldsymbol{t}$ lies uniformly at random in a ball (or sphere) of radius $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}$. Then, also for the reduced vector $\boldsymbol{t}'$ of norm at most $\beta \cdot \lambda_1(\mathcal{L})$, the closest lattice vector lies in a ball of radius $\lambda_1(\mathcal{L})$ around it. Since our list L contains all vectors of norm less than $\alpha \cdot \lambda_1(\mathcal{L})$, we will clearly find the closest lattice vector in the list L if the closest lattice vector lies in the intersection of two balls of radii $\lambda_1(\mathcal{L})$ (resp. $\alpha \cdot \lambda_1(\mathcal{L})$) around $\boldsymbol{t}'$ (resp. $\mathbf{0}$). Estimating the volume of this intersection of balls, relative to the volume of the ball of radius $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}'$, then gives us a lower bound on the success probability of the slicer, and a heuristic upper bound on the volume of the corresponding approximate Voronoi cell. This analysis ultimately leads to the aforementioned lemma.

Tightness of the proof. Note that the above proof technique only gives us a *lower bound* on the success probability, and an *upper bound* on the volume of the approximate Voronoi cell: when the target vector has been reduced to a vector of norm at most $\beta \cdot \lambda_1(\mathcal{L})$, we bound the success probability of the slicer by the probability that the slicer now terminates successfully in a *single* iteration. Since the algorithm might also succeed in more than one additional iteration, the actual success probability may be higher. A tighter analysis, perhaps showing that the given heuristic bound can be improved upon, is left for future work.

1.6 Intermezzo: Another few dimensions for free

Recently, Ducas [35] showed that in practice, one can effectively use the additional vectors found by lattice sieving to solve a few extra dimensions of SVP "for free". More precisely, by running a lattice sieve in a base dimension d, one can solve SVP in dimension $d' = d + \Theta(d/\log d)$ at little additional cost. This is done by taking all vectors returned by a d-dimensional lattice sieve, and running Babai's nearest plane algorithm [16] on all these vectors in the d'-dimensional lattice to find short vectors in the full lattice. If d' is close enough to d, one of these vectors will then be "rounded" to a shortest vector of the full lattice.

On a high level, Ducas' approach can be viewed as a sieving/enumeration hybrid, where the *top* part of enumeration is replaced with sieving, and the bottom part is done regularly as in enumeration, which is essentially equivalent to doing Babai rounding [16]. The approach of using a CVPP-solver inside enumeration is in a sense dual to Ducas' idea, as here the *bottom* part of the enumeration tree is replaced with a (sieving-like) CVPP routine. Since our CVPP complexities are strictly better than the best SVP/CVP complexities, we can also gain up to $\Theta(d/\log d)$ dimensions for free as follows:

1. First, we initialize an enumeration tree in the full lattice $\mathcal{L}$ of dimension $d' = d + k$, and we process the top $k = \varepsilon \cdot d/\log d$ levels as usual in enumeration. This will result in $2^{\Theta(k \log k)} = 2^{\Theta(d)}$ target vectors at level k, and this requires a similar time complexity of $2^{\Theta(d)}$ to generate all these target vectors.
2. Then, we run the CVPP preprocessing on the d-dimensional sublattice of $\mathcal{L}$ corresponding to the bottom part of the enumeration tree. This may for instance take time $2^{0.292d+o(d)}$ and space $2^{0.208d+o(d)}$ using the sieve of [18].
3. Finally, we take the batch of $2^{\Theta(d)}$ target vectors at level k in the enumeration tree, and we solve CVP for each of them with our approximate Voronoi cell and randomized slicing algorithm, with query time $2^{0.220d+o(d)}$ each.

By setting $k = \varepsilon \cdot d/\log d$ as above with small, constant $\varepsilon > 0$, the costs for solving SVP or CVP in dimension d' are asymptotically dominated by the costs of the preprocessing step, which is as costly as solving SVP or CVP in dimension d. So similar to [35], asymptotically we also get $\Theta(d/\log d)$ dimensions "for free". However, unlike for Ducas' idea, in practice the dimensions are likely not quite as free here, as there is more overhead for doing the CVPP-version of sieving than for Ducas' additional batch of Babai nearest plane calls.

Even more dimensions for free. A natural question one might ask now is: can both ideas be combined to get even more dimensions "for free"? At first sight, this seems hard to accomplish, as Ducas' idea speeds up SVP rather than CVPP. Furthermore, note that when solving SVP without Ducas' trick, one gets $2^{0.208d+o(d)}$ short lattice vectors when only one shortest vector is needed, and so in a sense one might "naturally" hope to gain something by going for only one short output vector. Here the analysis of the iterative slicer is already based on the fact that ultimately, we hope to reduce a single target vector to its closest neighbor in the lattice. There might be a way of combining both ideas to get even more dimensions for free, but for now this is left as an open problem.

1.7 Contributions: Experimental results

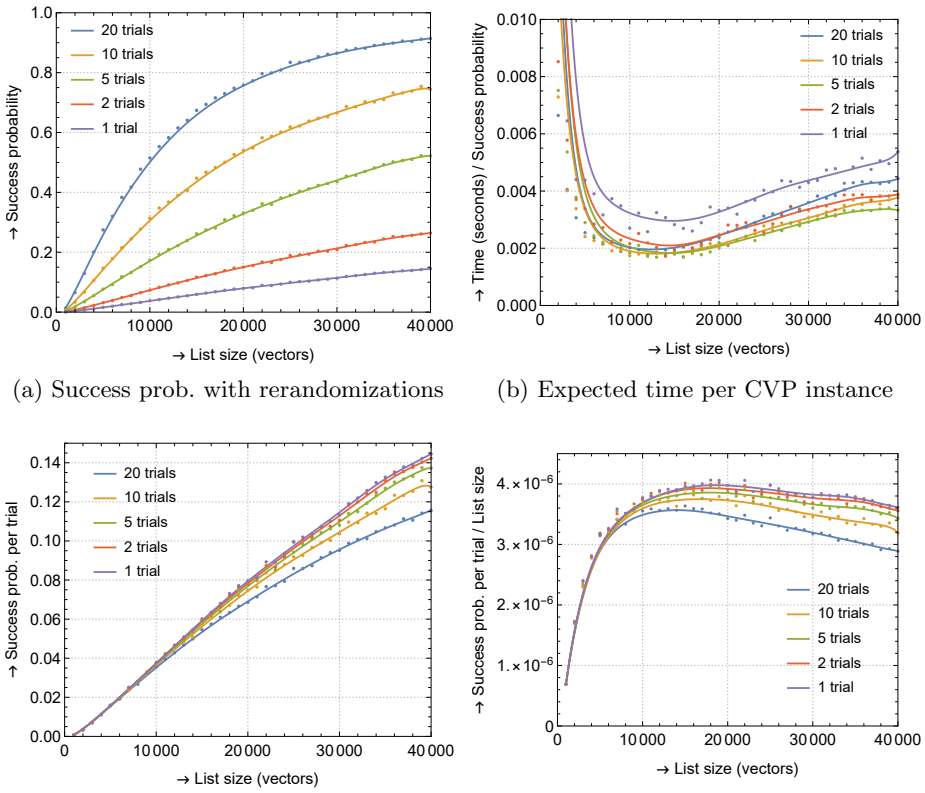
Besides the theoretical contributions mentioned above, with improved heuristic time and space complexities compared to [52], for the first time we also implemented a (sieving-based) CVPP solver using approximate Voronoi cells. For the preprocessing we used a slight modification of a lattice sieve, returning more vectors than a standard sieve, allowing us to vary the list size in our experiments. Our implementations serve two purposes: validating the additional heuristic assumption we make, and to see how well the algorithm performs in practice.

Validation of the randomization assumption. To obtain the aforementioned improved asymptotic complexities for solving CVPP, we required a new heuristic assumption, stating that if the iterative slicer succeeds with some probability p on a CVP instance $\boldsymbol{t}$, then we can repeat it $1/p$ times with perturbations $\boldsymbol{t}' \sim D_{\boldsymbol{t}+\mathcal{L},s}$ to achieve a high success probability for the same target $\boldsymbol{t}$. To verify this assumption, we implemented our method and tested it on lattices of dimension 50 with a range of randomly chosen targets to see whether, if the probability of success is small, repeating the method m times will increase the success rate by a factor m. Figure 3 shows performance metrics for various numbers of repetitions and for varying list sizes. In particular, Figure 3a illustrates the increased success probability as the number of repetitions increases, and Figure 3c shows that the normalized success probability per trial[3] seems independent of the number of repetitions. Therefore, the "expected time" metric as illustrated in Figure 3b appears to be independent of the number of trials.

Experimental performance. Unlike the success probabilities, the time complexity might vary a lot depending on the underlying nearest neighbor data structure. For our experiments we used hyperplane LSH [29] as also used in the HashSieve [50, 58], as it is easy to implement, has few parameters to set, and performs better in low dimensions ($d = 50$) than the LDSieve [18, 59].

To put the complexities of Figure 3b into perspective, let us compare the normalized time complexities for CVPP with the complexities of sieving for SVP, which by [52] are comparable to the costs for CVP. First, we note that the HashSieve algorithm solves SVP in approximately 4 seconds on the same machine. This means that in dimension 50, the expected time complexity for CVPP with the HashSieve (roughly 2 milliseconds) is approximately 2000 times smaller than the time for solving SVP. To explain this gap, observe that the list size for solving SVP is approximately 4000, and so the HashSieve algorithm needs to perform in the order of 4000 reductions of newly sampled vectors with a list of size 4000. For solving CVPP, we only need to reduce 1 target vector, with a slightly larger list of 10 000 to 15 000 vectors. So we save a factor 4000 on the number of reductions, but the searches are more expensive, leading to a speed-up of less than a factor 4000.

[3] As the success prob. q for m trials scales as $q = 1-(1-p)^m$ if each trial independently has success prob. p, we computed the success prob. per trial as $p = 1 - (1-q)^{1/m}$.



(a) Success prob. with rerandomizations

(b) Expected time per CVP instance

(c) Average success prob. per trial

(d) Success prob. per trial, per vector

Fig. 3. Experimental results for solving CVPP with randomized slicing in dimension 50. Each data point corresponds to 10 000 random target vectors for those parameters.

Predictions and extrapolations. For solving SVP or CVP, the HashSieve [50] reports time complexities in dimension d of $2^{0.45d-19}$ seconds, corresponding to 11 seconds in dimension 50, i.e., a factor 3 slower than here. This is based on doing $n \approx 2^{0.21d}$ reductions of vectors with the list. If doing only one of these searches takes a factor $2^{0.21d}$ less time, and we take into account that for SVP the time complexity is now a factor 3 less than in [50], then we obtain an estimated complexity for CVPP in dimension d of $2^{0.24d-19}/3$, which for $d = 50$ corresponds to approximately 2.6 milliseconds. A rough extrapolation would then lead to a time complexity in dimension 100 of only 11 seconds. This however seems to be rather optimistic – preliminary experiments in dimensions 60 and 70 suggest that the overhead of using a lot of memory may be rather high here, as the list size is usually even larger than for standard sieving.

1.8 Contributions: Asymptotics for variants of CVPP

For easier variants of CVP, such as when the target lies closer to the lattice than expected or an approximate solution to CVP suffices as a solution, we obtain considerable gains in both the time and space complexities when using preprocessing. We explicitly consider two variants of CVPP below.

BDDP$_\delta$. For bounded distance decoding with preprocessing (BDDP), we are given a target vector $\boldsymbol{t}$ and a guarantee that $\boldsymbol{t}$ lies within distance $\delta \cdot \lambda_1(\mathcal{L})$ to the nearest lattice vector, for some parameter $\delta > 0$. By the Gaussian heuristic, setting $\delta = 1$ makes this problem as hard as general CVPP without a distance guarantee, while for small $\delta \to 0$ polynomial-time algorithms exist [16].

By adjusting the analysis leading up to Theorem 1 for BDDP, we obtain the same result as Theorem 1 with two modifications: T_2 is replaced by $\mathrm{T}_2^{(\delta)}$ below, and the range of admissable values α changes to (α_0, α_1), with α_0 the smallest root larger than 1 of the denominator of the left-most term in $\mathrm{T}_2^{(\delta)}$, and α_1 the smallest value larger than 1 such that the left-most term in $\mathrm{T}_2^{(\delta)}$ equals 1. The resulting optimized trade-offs for various $\delta \in (0, 1)$ are plotted in Figure 4a.

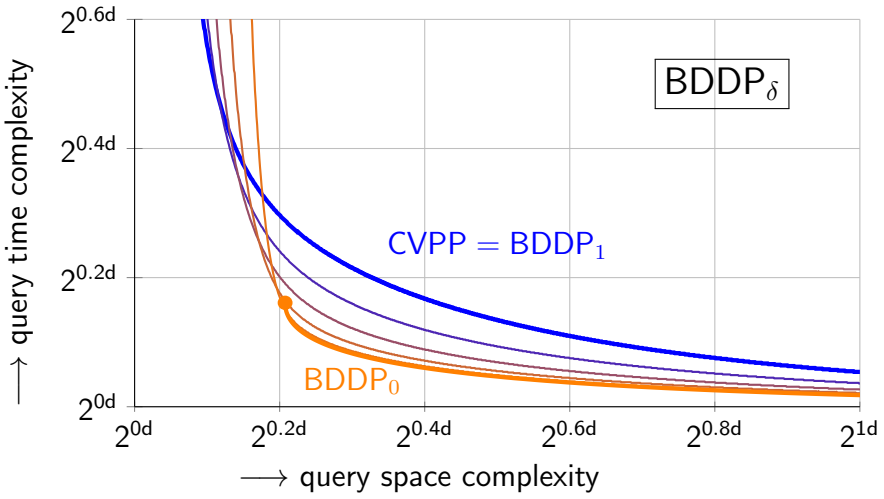
$$\mathrm{T}_2^{(\delta)} = \left(\frac{16\alpha^4 \left(\alpha^2 - 1\right) \delta^2}{-9\alpha^8 + 8\alpha^6(3+5\delta^2) - 8\alpha^4(2+9\delta^2+2\delta^4) + 32\alpha^2(\delta^2+\delta^4) - 16\delta^4} \cdot [\dots] \right)^{d/2+o(d)} . \quad (7)$$

Note that in the limit of $\delta \to 0$, our algorithm tries to reduce a target close to the lattice to the origin. This is similar to reducing a vector to the $\mathbf{0}$-vector in the GaussSieve [65], and even with a long list of all short lattice vectors this does not occur with probability 1. Here also the limiting curve in Figure 4a shows that for $\delta \to 0$ with suitable parameterization we can do better than just with sieving, but we do not get polynomial time and space complexities.

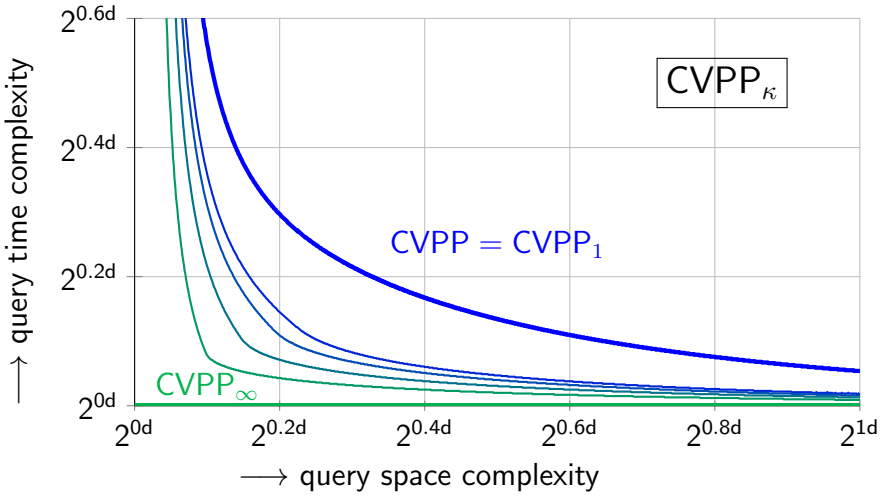
CVPP$_\kappa$. For the approximate version of CVPP, a lattice vector $\boldsymbol{v}$ qualifies as a solution for $\boldsymbol{t}$ if it lies at most a factor κ further from the real distance of $\boldsymbol{t}$ from the lattice, for some $\kappa \geq 1$. Heuristically, this is essentially equivalent to looking for any lattice vector within radius $\kappa \cdot \lambda_1(\mathcal{L})$ of the target, and similar to BDDP the resulting trade-offs can be summarized by Theorem 1 where T_2 is replaced by $\mathrm{T}_2^{(\kappa)}$ below, and the range of admissable values α again changes to (α_0, α_1) as before.

$$\mathrm{T}_2^{(\kappa)} = \left(\frac{16\alpha^4 \left(\alpha^2 - 1\right)}{-9\alpha^8 + 8\alpha^6(3+5\kappa^2) - 8\alpha^4(2+9\kappa^2+2\kappa^4) + 32\alpha^2(\kappa^2+\kappa^4) - 16\kappa^4} \cdot [\dots] \right)^{d/2+o(d)} . \quad (8)$$

For increasing approximation factors $\kappa \to \infty$, our algorithm tries to reduce a target vector to vector of norm less than $\kappa \cdot \lambda_1(\mathcal{L})$. For large κ this is increasingly easy to achieve, and as $\kappa \to \infty$, both the query time and space complexities in our analysis converge to zero as expected. Figure 4b highlights this asymptote, and illustrates the other trade-offs through some examples for small $\kappa > 1$.



(a) Heuristic complexities for BDDP_δ for different values $\delta \in \{0, 0.2, \ldots, 0.8, 1\}$. Smaller δ correspond to easier problems but also to a larger lower bound α_0 on α. The trade-off for $\delta \to 0$ is indicated by the thick orange line.



(b) Heuristic query complexities for CVPP_κ for different approximation factors $\kappa \in \{\sqrt{4/3}, 1.2, 1.3, 1.5, \infty\}$. The thick green line shows the limit as $\kappa \to \infty$.

Fig. 4. Asymptotics for solving variants of CVP(P) with approximate Voronoi cells: (a) BDDP_δ and (b) CVPP_κ. Note that the (tail of the) curve for $\mathrm{CVPP}_{\sqrt{4/3}}$ overlaps with the curve for BDDP_0.

1.9 Open problems

Combination with other techniques. The focus of this work was on the asymptotic complexities we can achieve for high dimensions d, and therefore we focused only on including techniques from the literature that lead to the best asymptotics. In practice however, there may be various other techniques that can help speed up these methods in moderate dimensions. This for instance includes Ducas' dimensions for free [35], progressive sieving [35, 54], the recent sieving-BKZ hybrid [7], and faster NNS techniques [7, 11]. Incorporating such techniques will likely affect the experimental performance as well, and future work may show how well the proposed techniques truly perform in practice when all the state-of-the-art techniques are combined into one.

Faster enumeration with approximate Voronoi cells. As explained above, one potential application of our CVPP algorithm is as a subroutine within enumeration, to speed up the searches in the bottom part of the tree. Such an algorithm can be viewed as a trade-off between enumeration and sieving, where the level at which we insert the CVPP oracle determines whether we are closer to enumeration or to sieving. An open question remains whether this would lead to faster algorithms in practice, or if the preprocessing/query costs are too high. Note that depending on at which level of the tree the CVPP oracle is inserted, and on the amount of pruning in enumeration, the hardness of the CVP instances at these levels also changes. Optimizing all parameters involved in such a combination appears to be a complex task, and is left for future work.

Sieving in the dual lattice. For the application of CVPP within enumeration, observe that a decisional CVPP oracle, deciding whether a vector lies close to the lattice or not, may actually be sufficient; most branches of the enumeration tree will not lead to a solution, and therefore in most cases running an accurate decision-CVPP oracle is enough to determine that this subtree is not the right subtree. For those few subtrees that potentially do contain a solution, one could then run a full CVP(P) algorithm at a slightly higher cost. Improving the complexities for the decision-version of CVPP may therefore be an interesting future direction, and perhaps one approach could be to combine this with ideas from [5], by running a lattice sieve on the dual lattice to find many short vectors in the dual lattice, which can then be used to check if a target vector lies close to the primal lattice or not.

Quantum complexities. As one of the strengths of lattice-based cryptography is its conjectured resistance to quantum attacks [22], it is important to study the potential impact of quantum improvements to SVP and CVP algorithms, so that the parameters can be chosen to be secure in a post-quantum world [15, 55]. For lattice sieving for solving SVP, the time complexity exponent potentially decreases by approximately 25% [55], and for CVPP we expect the exponents may decrease by approximately 25% as well. Studying the exact quantum asymptotics of solving CVPP with approximate Voronoi cells is left for future work.

1.10 Outline

Due to space restrictions for the original publication at PQCrypto 2019, and to maintain the same paper structure as in the published version, the remainder of the paper, including full details on all claims, is given in the appendix. Below we briefly outline the contents of these appendices for the interested reader.

Appendix A – Preliminaries
This section describes preliminary results and notation for the technical contents, formally states the main hard problems discussed in the paper, formalizes the heuristic assumptions made throughout the paper, and describes existing results on nearest neighbor searching, lattice sieving algorithms, Voronoi cells, and Voronoi cell algorithms.

Appendix B – Approximate Voronoi cells
In Appendix B we formalize the CVPP approach considered in this paper in terms of our approximate Voronoi cell framework with randomized slicing, and we derive our main results regarding improved asymptotic complexities for exact CVPP. Approximate Voronoi cells are formally introduced, the main results are stated and proved in terms of this framework, and all corresponding algorithms are given in pseudocode.

Appendix C – Experimental results
Appendix C describes the experiments we performed with these methods in more detail, both to verify the (additional) heuristic assumptions we made for this paper, and to assess the practicality of our CVPP algorithm. Here we also briefly compare our results to various published complexities for SVP or CVP(P), to put these numbers into context.

Appendix D – Asymptotics for variants of CVPP
The last appendix finally discusses asymptotic results for variants of CVPP, namely approximate CVPP and BDDP. This section contains a more formal statement of the results given in Section 1.8, and explains how the analysis changes compared to the analysis for exact CVPP, and how this leads to improved complexities for these slightly easier variants of (exact) CVPP.

Acknowledgments

The authors are indebted to Léo Ducas, whose ideas and suggestions on this topic motivated work on this paper. The authors are further grateful to the reviewers, whose thorough study of the contents (with one review even exceeding the page limit for the conference) significantly helped improve the contents of the paper, as well as improve the presentation of the results. Emmanouil Doulgerakis is supported by the NWO under grant 628.001.028 (FASOR). At the time of writing a preliminary version of this paper, Thijs Laarhoven was supported by the SNSF ERC Transfer Grant CRETP2-166734 FELICITY. At the time of publishing, Thijs Laarhoven is supported by a Veni Innovational Research Grant from NWO under project number 016.Veni.192.005.

References

1. SVP challenge, 2018. `http://latticechallenge.org/svp-challenge/`.
2. Divesh Aggarwal, Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. Solving the shortest vector problem in 2^n time via discrete Gaussian sampling. In *STOC*, pages 733–742, 2015.
3. Divesh Aggarwal, Daniel Dadush, and Noah Stephens-Davidowitz. Solving the closest vector problem in 2^n time – the discrete Gaussian strikes again! In *FOCS*, pages 563–582, 2015.
4. Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.
5. Dorit Aharonov and Oded Regev. Lattice problems in $\mathsf{NP} \cap \mathsf{coNP}$. In *FOCS*, pages 362–371, 2004.
6. Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *STOC*, pages 601–610, 2001.
7. Martin Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In *EUROCRYPT*, 2019.
8. Misha Alekhnovich, Subhash Khot, Guy Kindler, and Nisheeth Vishnoi. Hardness of approximating the closest vector problem with pre-processing. In *FOCS*, pages 216–225, 2005.
9. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *USENIX Security Symposium*, pages 327–343, 2016.
10. Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006.
11. Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. In *NIPS*, pages 1225–1233, 2015.
12. Alexandr Andoni, Thijs Laarhoven, Ilya Razenshteyn, and Erik Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. In *SODA*, pages 47–66, 2017.
13. Alexandr Andoni and Ilya Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *STOC*, pages 793–801, 2015.
14. Yoshinori Aono and Phong Q. Nguyên. Random sampling revisited: lattice enumeration with discrete pruning. In *EUROCRYPT*, pages 65–102, 2017.
15. Yoshinori Aono, Phong Q. Nguyen, and Yixin Shen. Quantum lattice enumeration and tweaking discrete pruning. In *ASIACRYPT*, 2018.
16. László Babai. On lovasz lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.
17. Shi Bai, Thijs Laarhoven, and Damien Stehlé. Tuple lattice sieving. In *ANTS*, pages 146–162, 2016.
18. 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.
19. Anja Becker, Nicolas Gama, and Antoine Joux. A sieve algorithm based on overlattices. In *ANTS*, pages 49–70, 2014.
20. 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.

21. Anja Becker and Thijs Laarhoven. Efficient (ideal) lattice sieving using cross-polytope LSH. In *AFRICACRYPT*, pages 3–23, 2016.
22. Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-quantum cryptography*. Springer, 2009.
23. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: reducing attack surface at low cost. In *SAC*, pages 235–260, 2017.
24. Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. *Cryptology ePrint Archive, Report 2018/725*, 2018.
25. Nicolas Bonifas and Daniel Dadush. Short paths on the Voronoi graph and the closest vector problem with preprocessing. In *SODA*, pages 295–314, 2015.
26. Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *CCS*, pages 1006–1018, 2016.
27. 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 *Euro S&P*, pages 353–367, 2018.
28. 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*, 3(4):313–329, 2016.
29. Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.
30. Tobias Christiani. A framework for similarity search with space-time tradeoffs using locality-sensitive filtering. In *SODA*, pages 31–46, 2017.
31. John H. Conway and Neil J.A. Sloane. *Sphere packings, lattices and groups*. Springer, 1999.
32. Fábio Correia, Artur Mariano, Alberto Proenca, Christian Bischof, and Erik Agrell. Parallel improved Schnorr-Euchner enumeration SE++ for the CVP and SVP. In *PDP*, pages 596–603, 2016.
33. Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. On the closest vector problem with a distance guarantee. In *CCC*, pages 98–109, 2014.
34. The FPLLL development team. fplll, a lattice reduction library. Available at `https://github.com/fplll/fplll`, 2016.
35. Léo Ducas. Shortest vector from lattice sieving: a few dimensions for free. In *EUROCRYPT*, pages 125–145, 2018.
36. Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Dilithium: Digital signatures from module lattices. In *CHES*, volume 2018, pages 238–268, 2018.
37. Ulrich Feige and Daniele Micciancio. The inapproximability of lattice and coding problems with preprocessing. In *CCC*, pages 32–40, 2002.
38. Ulrich Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice. *Mathematics of Computation*, 44(170):463–471, 1985.
39. 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.
40. Nicolas Gama, Phong Q. Nguyên, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, pages 257–278, 2010.

41. Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, pages 197–206, 2008.
42. Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel shortest lattice vector enumeration on graphics cards. In *AFRICACRYPT*, pages 52–68, 2010.
43. Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate k-list problem in Euclidean norm. In *PKC*, pages 16–40, 2017.
44. Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven. Speed-ups and time-memory trade-offs for tuple lattice sieving. In *PKC*, pages 407–436, 2018.
45. Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.
46. 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.
47. Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *STOC*, pages 193–206, 1983.
48. Paul Kirchner and Pierre-Alain Fouque. Time-memory trade-off for lattice enumeration in a ball. *Cryptology ePrint Archive, Report 2016/222*, 2016.
49. Philip Klein. Finding the closest lattice vector when it's unusually close. In *SODA*, pages 937–941, 2000.
50. Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *CRYPTO*, pages 3–22, 2015.
51. Thijs Laarhoven. Tradeoffs for nearest neighbors on the sphere. *arXiv:1511.07527 [cs.DS]*, pages 1–16, 2015.
52. Thijs Laarhoven. Sieving for closest lattice vectors (with preprocessing). In *SAC*, pages 523–542, 2016.
53. Thijs Laarhoven and Benne de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In *LATINCRYPT*, pages 101–118, 2015.
54. Thijs Laarhoven and Artur Mariano. Progressive lattice sieving. In *PQCrypto*, pages 292–311, 2018.
55. Thijs Laarhoven, Michele Mosca, and Joop van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2):375–400, 2015.
56. Jeffrey C. Lagarias, Hendrik W. Lenstra, and Claus-Peter Schnorr. Korkin-Zolotarev bases and successive minima of a lattice and its reciprocal lattice. *Combinatorica*, 10(4):333–348, 1990.
57. Artur Mariano and Christian Bischof. Enhancing the scalability and memory usage of HashSieve on multi-core CPUs. In *PDP*, pages 545–552, 2016.
58. 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.
59. Artur Mariano, Thijs Laarhoven, and Christian Bischof. A parallel variant of LDSieve for the SVP on lattices. In *PDP*, pages 23–30, 2017.
60. Artur Mariano, Özgür Dagdelen, and Christian Bischof. A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In *EURO-PAR*, pages 48–59, 2014.
61. 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.
62. Daniele Micciancio. The hardness of the closest vector problem with preprocessing. *IEEE Transactions on Information Theory*, 47(3):1212–1215, 2001.

63. Daniele Micciancio. Efficient reductions among lattice problems. In *SODA*, pages 84–93, 2008.
64. 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.
65. Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA*, pages 1468–1480, 2010.
66. Daniele Micciancio and Michael Walter. Fast lattice point enumeration with minimal overhead. In *SODA*, pages 276–294, 2015.
67. Benjamin Milde and Michael Schneider. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In *PACT*, pages 452–458, 2011.
68. Phong Q. Nguyên and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.
69. Özgür Dagdelen and Michael Schneider. Parallel enumeration of shortest lattice vectors. In *EURO-PAR*, pages 211–222, 2010.
70. Oded Regev. Improved inapproximability of lattice and coding problems with preprocessing. *IEEE Transactions on Information Theory*, 50(9):2031–2037, 2004.
71. Michael Schneider. Analysis of Gauss-Sieve for solving the shortest vector problem in lattices. In *WALCOM*, pages 89–97, 2011.
72. Michael Schneider. Sieving for short vectors in ideal lattices. In *AFRICACRYPT*, pages 375–391, 2013.
73. 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.
74. Noah Stephens-Davidowitz. Dimension-preserving reductions between lattice problems. *Available at http://noahsd.com/latticeproblems.pdf*, pages 1–6, 2016.
75. Joop van de Pol. Lattice-based cryptography. Master's thesis, Eindhoven University of Technology, 2011.
76. Emanuele Viterbo and Ezio Biglieri. Computing the Voronoi cell of a lattice: the diamond-cutting algorithm. *IEEE Transactions on Information Theory*, 42(1):161–171, 1996.
77. Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *arXiv:1408.2927 [cs.DS]*, pages 1–29, 2014.
78. 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.
79. 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.
80. Feng Zhang, Yanbin Pan, and Gengran Hu. A three-level sieve algorithm for the shortest vector problem. In *SAC*, pages 29–47, 2013.

A Preliminaries

A.1 Notation

We write vectors in boldface (e.g., $\boldsymbol{x}$), and we denote its indices with non-boldface subscripts (e.g., x_i). Throughout the paper we primarily consider problems in the Euclidean norm, hence unless stated otherwise, $\|\boldsymbol{x}\| = \|\boldsymbol{x}\|_2 := (\sum_i x_i^2)^{1/2}$ denotes the Euclidean norm of the vector $\boldsymbol{x}$. We write $\mathcal{S}^{d-1}$ for the Euclidean unit sphere in $\mathbb{R}^d$, i.e., the set of vectors $\boldsymbol{x} \in \mathbb{R}^d$ with $\|\boldsymbol{x}\| = 1$. We denote balls in high-dimensional space by $\mathcal{B}(\boldsymbol{x}, r) := \{\boldsymbol{y} \in \mathbb{R}^d : \|\boldsymbol{y} - \boldsymbol{x}\| \leq r\}$, and we write $\mathcal{H}(\boldsymbol{x}) := \{\boldsymbol{v} \in \mathbb{R}^d : \|\boldsymbol{v}\| \leq \|\boldsymbol{v} - \boldsymbol{x}\|\}$ for half-spaces whose boundaries are the hyperplanes orthogonal to $\boldsymbol{x}$ and passing through $\frac{1}{2}\boldsymbol{x}$. For measurable regions $\mathcal{R} \subset \mathbb{R}^d$, we denote their volume by $\mathrm{vol}(\mathcal{R})$.

Given a basis $\boldsymbol{B} = \{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_d\} \subset \mathbb{R}^d$ of linearly independent vectors, the lattice generated by $\boldsymbol{B}$ is formally defined as $\mathcal{L} = \mathcal{L}(\boldsymbol{B}) := \{\sum_{i=1}^d \lambda_i \boldsymbol{b}_i : \lambda_i \in \mathbb{Z}\}$. We denote the length of a shortest non-zero vector in a lattice by $\lambda_1(\mathcal{L}) := \min_{\boldsymbol{v} \in \mathcal{L} \setminus \{\boldsymbol{0}\}} \|\boldsymbol{v}\|$. We write $\det(\mathcal{L}) := \det(\boldsymbol{B}^T\boldsymbol{B})^{1/2}$ for the determinant of the lattice, which notably is independent of the basis of the lattice: $\det(\boldsymbol{B}^T\boldsymbol{B}) = \det(\boldsymbol{R}^T\boldsymbol{R})$ for any two bases $\boldsymbol{B}, \boldsymbol{R}$ generating the same lattice $\mathcal{L}$. As lattices are groups, we define sublattices as subgroups contained in the full lattice and maintaining the additive group structure.

Given a parameter $s > 0$, we define $\rho_s(\boldsymbol{v}) := \exp(-\pi\|\boldsymbol{v}\|^2/s^2)$, and given a lattice $\mathcal{L}$ we define $\rho_s(\mathcal{L}) := \sum_{\boldsymbol{v} \in \mathcal{L}} \rho_s(\boldsymbol{v})$. We define a discrete probability distribution on this lattice $\mathcal{L}$ by setting $\Pr(\boldsymbol{X} = \boldsymbol{x}) := \rho_s(\boldsymbol{x})/\rho_s(\mathcal{L})$ for $\boldsymbol{x} \in \mathcal{L}$. We refer to this distribution as the discrete Gaussian distribution on $\mathcal{L}$ with parameter s, and we write $\boldsymbol{X} \sim D_{\mathcal{L},s}$ to denote that the random variable $\boldsymbol{X}$ follows this distribution. Sampling from $D_{\mathcal{L},s}$ for large s can efficiently be done in $\mathrm{poly}(d)$ time and space [41,49], still returning lattice vectors of expected norm less than $2^{O(d)} \cdot \lambda_1(\mathcal{L})$. For cosets $\boldsymbol{t}+\mathcal{L}$ of a lattice $\mathcal{L}$, we analogously define $D_{\boldsymbol{t}+\mathcal{L},s}$ with relative density $\rho_{s,\boldsymbol{t}}(\boldsymbol{v}) := \exp(-\pi\|\boldsymbol{v} - \boldsymbol{t}\|^2/s^2)$.

A.2 Problem statements

Problems without preprocessing. Let us first recall formal definitions of some common hard lattice problems, and problems often described in the (approximate) nearest neighbor literature. The problems below are lattice problems where the stated problem needs to be solved directly – there is no preprocessing stage based on partial information for solving the problem.

Definition 1 (Shortest Vector Problem – SVP). *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$, find a non-zero vector $\boldsymbol{s} \in \mathcal{L}$ such that $\|\boldsymbol{s}\| = \lambda_1(\mathcal{L})$.*

Definition 2 (Approximate Shortest Vector Problem – SVP$_\kappa$). *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$ and an approximation factor $\kappa \geq 1$, find a non-zero vector $\boldsymbol{s} \in \mathcal{L}$ such that $\|\boldsymbol{s}\| \leq \kappa \cdot \lambda_1(\mathcal{L})$.*

Definition 3 (Closest Vector Problem – CVP). *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$ and a target vector $\boldsymbol{t} \in \mathbb{R}^d$, find a vector $\boldsymbol{s} \in \mathcal{L}$ with $\|\boldsymbol{s} - \boldsymbol{t}\| = \min_{\boldsymbol{v} \in \mathcal{L}} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

Definition 4 (Approximate Closest Vector Problem – CVP_κ). *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$, a target vector $\boldsymbol{t} \in \mathbb{R}^d$, and an approximation factor $\kappa \geq 1$, find a vector $\boldsymbol{s} \in \mathcal{L}$ with $\|\boldsymbol{s} - \boldsymbol{t}\| \leq \kappa \cdot \min_{\boldsymbol{v} \in \mathcal{L}} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

Definition 5 (Bounded Distance Decoding – BDD_δ). *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$, a target vector $\boldsymbol{t} \in \mathbb{R}^d$, and a distance guarantee $\delta > 0$ with the promise that $\min_{\boldsymbol{v} \in \mathcal{L}} \|\boldsymbol{v} - \boldsymbol{t}\| \leq \delta \cdot \lambda_1(\mathcal{L})$, find a vector $\boldsymbol{s} \in \mathcal{L}$ with $\|\boldsymbol{s} - \boldsymbol{t}\| = \min_{\boldsymbol{v} \in \mathcal{L}} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

Problems with preprocessing. We denote the preprocessing versions of CVP, CVP_κ, and BDD_δ, by CVPP, CVPP_κ, and BDDP_δ respectively. In the preprocessing variants of these problems, the lattice is given in advance and may be preprocessed so that, when given the target vector $\boldsymbol{t}$, a solution to the problem can potentially be provided faster than without preprocessing the lattice. For SVP or SVP_κ such a preprocessing variant clearly does not make sense, as the shortest vector could be precomputed, making "SVPP" trivially solvable with polynomial (query) time and (precomputed) space. Similar results are not true for CVPP, i.e., even with unlimited preprocessing time, it still seems hard to solve random problem instances [62].

Related to nearest neighbor searching, we recall the following problem definitions. These are all problems where preprocessing is essential, and the most general statements of these problems are given below.

Definition 6 (Nearest Neighbor Searching – NNS). *Given a finite set $L \subset \mathbb{R}^d$, preprocess L such that, when given a target vector $\boldsymbol{t} \in \mathbb{R}^d$ later, one can quickly find a vector $\boldsymbol{s} \in L$ such that $\|\boldsymbol{s} - \boldsymbol{t}\| = \min_{\boldsymbol{v} \in L} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

Definition 7 (Approximate Nearest Neighbor Searching – NNS_c). *For a finite set $L \subset \mathbb{R}^d$ and an approximation factor $c \geq 1$, preprocess L such that when given a target vector $\boldsymbol{t} \in \mathbb{R}^d$ later, one can quickly find a vector $\boldsymbol{s} \in L$ such that $\|\boldsymbol{s} - \boldsymbol{t}\| \leq c \cdot \min_{\boldsymbol{v} \in L} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

NNS is essentially equivalent to CVPP, except that (1) the data set in nearest neighbor searching is not assumed to be structured, and (2) the data set is assumed to be of finite cardinality $n < \infty$. Naive brute force algorithms for nearest neighbor searching take $O(n)$ time and $O(n)$ space without any preprocessing costs, and the literature commonly focuses on sublinear time algorithms, running in time $O(n^\rho)$ for $\rho < 1$, commonly with superlinear space and preprocessing costs. Note that for the application of NNS techniques in the context of lattice sieving, one commonly has $n = 2^{\Theta(d)}$, whereas the literature on NNS often focuses on the case $n = 2^{o(d)}$. It is not clear whether lower bounds on the query complexity for (approximate) nearest neighbor searching (e.g., [12, 30]) also apply in the context of lattice sieving.

A.3 Heuristic assumptions

As discussed in the introduction, worst-case analyses of algorithms for SVP and CVP(P) are far off from the best average-case performance we can achieve in practice by just testing these algorithms on random lattices. For purposes in cryptography, where it is in a sense better to be safe than sorry, it therefore makes sense to try to analyze algorithms under "mild" assumptions that allow us to obtain tighter estimates on their performance on average-case lattices. Even if we can no longer formally prove these complexity bounds hold in the worst-case—indeed, these complexities may not be accurate for exotic, dense lattices like the Leech lattice [31]—such estimates may give us a better idea of the actual performance of the best algorithms on random lattices appearing in cryptanalysis.

A commonly made heuristic assumption for analyzing lattice algorithms is the Gaussian heuristic, stating that for a (random) region $\Omega \subset \mathbb{R}^d$, for random lattices the number of lattice points contained in this region is roughly equal to $\mathrm{vol}(\Omega)/\det(\mathcal{L})$. A consequence of this assumption is that for random lattices of high dimension d, the length of the shortest vector can be approximated as:

$$\lambda_1(\mathcal{L}) \approx \mathrm{GH}(\mathcal{L}) := \sqrt{\frac{d}{2\pi e}} \cdot \det(\mathcal{L})^{1/d} \cdot (1 + o(1)). \tag{9}$$

For average-case, random CVP(P) target instances $\boldsymbol{t} \in \mathbb{R}^d$, this further means that we expect the distance to the closest lattice point to be roughly $\lambda_1(\mathcal{L})$: any smaller ball around $\boldsymbol{t}$ of radius $(1-\varepsilon)\mathrm{GH}(\mathcal{L})$ is expected to be empty, and a bigger ball of radius $(1+\varepsilon)\mathrm{GH}(\mathcal{L})$ will likely contain up to $(1+\varepsilon)^{d+o(d)}$ (exponentially many) lattice points for a random lattice $\mathcal{L}$.

When working with lattice vectors $\boldsymbol{v} \in \mathcal{L}$, even if for some algorithm we know the distribution of the "input vectors" over the lattice, after modifying these vectors we quickly lose track of the actual distribution these vectors now follow. A common assumption here is then to simply assume that if at some point in the execution of the algorithm, we are left with a vector $\boldsymbol{v}' \in \mathcal{L}$ of norm $\|\boldsymbol{v}'\|$, then this vector follows a uniform distribution over the sphere of radius $\|\boldsymbol{v}'\|$ around the origin. Clearly this assumption is incorrect and ignores the discrete nature of the lattice (which may play a bigger role as the radius gets smaller), but unless this inaccuracy is exploited and abused in the analysis, this often gives us a better grip on, e.g., the probability that two vectors $\boldsymbol{v}, \boldsymbol{w}$ appearing in a lattice algorithm can be combined to form a shorter vector $\boldsymbol{v} \pm \boldsymbol{w}$.

Finally, observe that although these heuristic assumptions may not be provably accurate for all lattices, extensive experimentation with these algorithms on random lattices has supported these claims for average-case lattices. For new assumptions we might introduce later, we will also provide experimental evidence to back up the theoretical, asymptotic claims.

A.4 The CVPP cost model

For analyzing the performance of CVPP algorithms, we split these methods into two phases: the preprocessing phase (whose input is only the lattice $\mathcal{L}$), and the query phase (where also the target vector $\boldsymbol{t}$ is known). We keep track of four costs of CVPP algorithms.

- **Preprocessing phase**: Preprocess the lattice $\mathcal{L}$ (without the target $\boldsymbol{t}$);
  S_1: The memory used during the preprocessing phase;
  T_1: The time used during the preprocessing phase;
- **Query phase**: Process the query $\boldsymbol{t}$ and output a vector $\boldsymbol{s} \in \mathcal{L}$ near $\boldsymbol{t}$;
  S_2: The memory used during the query phase;
  T_2: The time used during the query phase.

Intuitively the main goal of CVPP algorithms is to reduce the complexities of the query phase $(\mathrm{S}_2, \mathrm{T}_2)$ compared to a non-preprocessed CVP algorithm. That way, a sufficiently large batch of CVP instances on the same lattice can be solved faster than with direct CVP approaches. However, in any practical application we need to perform the preprocessing at least once, and therefore CVPP algorithms with enormous preprocessing costs may be useless even if the query complexities are great. Also note that as we are interested in reducing the query complexity compared to solving CVP, and this usually comes at the cost of a higher preprocessing cost, we generally have $\mathrm{T}_2 \leq \mathrm{T} \leq \mathrm{T}_1$, where T is the corresponding asymptotic time complexity for CVP.

Polynomial vs. exponential advice. Note that in the literature on CVPP, a common assumption is that the output of the preprocessing stage has size *polynomial* in the lattice dimension d [62]. This is partly because with unlimited preprocessing power (time and space), heuristically the "post-processing" stage of CVPP can easily be made polynomial time. As an example, one could cover a sufficiently large ball around the origin with tiny cubes, and precompute/store the centers of these cubes, together with solutions to CVP with these centers as target vectors. Given a target vector for CVPP, one could then size-reduce with an LLL-reduced basis $\boldsymbol{B}$, identify the cube the vector is in, and assuming the net of cubes is sufficiently fine-grained, a solution to CVP for the center of this cube is then likely a solution to CVP for the target vector as well.

Since the costs of the preprocessing stage are usually disregarded when assessing the performance of a CVPP method, this would make CVPP (and CVPP_κ, BDDP_δ) altogether trivial. Throughout we are interested in the practicality of the "total package" of the CVPP algorithm, including the preprocessing. Taking these costs into account, the problem is no longer trivial even when allowing exponential-sized advice from the preprocessing stage. We explicitly do not make the assumption that the output of the preprocessing stage is of polynomial size.

A.5 Nearest neighbor algorithms

A celebrated technique for finding near neighbors in high dimensions is locality-sensitive hashing (LSH) [10,11,13,29,45,77]. Here the idea is to construct many random partitions of the space, and index the data set L in hash tables with buckets corresponding to the regions induced by these partitions. Preprocessing then consists of constructing these hash tables, and a query $\boldsymbol{t}$ is answered by doing a lookup in each of the hash tables, and searching for a(n approximate) nearest neighbor in the hash buckets corresponding to $\boldsymbol{t}$. For a data set of size $|L| = n$, this commonly leads to a sublinear time complexity $O(n^\rho)$ ($\rho < 1$) as long as an approximate solution suffices, or if the majority of data points lie significantly further from the target than the nearest point in the data set. LSH has also been used to speed up lattice sieving, and more details on this can be found in [20,21,50,53], as well as implicitly in [78,80].

Similar to locality-sensitive hash functions, locality-sensitive filters (LSF) [12, 18,30] divide the space into regions, with the added relaxation that these regions do not have to form a proper partition; regions may overlap, and part of the space may not even be covered by any region. This leads to improved results when n is exponential in d [18,51], and allows for more natural time–space trade-offs compared to LSH for arbitrary n [12,30,51]. For the case of $n = 2^{o(d)}$, this leads to optimal trade-offs within certain frameworks [12,30].

Below we restate the main result of [51] for our applications, where n is assumed to be exponential in d. The specific problem considered here is: given a data set L of points sampled uniformly at random from the unit sphere $\mathcal{S}^{d-1}$, and a random query $\boldsymbol{t} \in \mathcal{S}^{d-1}$, return a vector $\boldsymbol{w} \in L$ such that the angle between $\boldsymbol{w}$ and $\boldsymbol{t}$ is at most $\theta \in (0, \frac{\pi}{2})$. The following result further assumes that the list L contains exactly $n = (1/\sin\theta)^{d+o(d)}$ vectors, denoted the *critical density* in [51]. The following is a restatement of [51, Corollary 1].

Lemma 2 (Nearest neighbor costs for spherical data sets). *Let $\theta \in (0, \frac{1}{2}\pi)$, and let $u \in [\cos\theta, 1/\cos\theta]$. Let $L \subset \mathcal{S}^{d-1}$ be a list of $n = (1/\sin\theta)^{d+o(d)}$ vectors sampled uniformly at random from $\mathcal{S}^{d-1}$. Then, using spherical LSF with parameters $\alpha_{\mathrm{q}} = u\cos\theta$ and $\alpha_{\mathrm{u}} = \cos\theta$, one can preprocess L in time $n^{1+\rho_{\mathrm{u}}+o(1)}$, using $n^{1+\rho_{\mathrm{u}}+o(1)}$ space, and with high probability answer a random query $\boldsymbol{t} \in \mathcal{S}^{d-1}$ correctly in time $n^{\rho_{\mathrm{q}}+o(1)}$, where:*

$$n^{\rho_{\mathrm{q}}} = \left(\frac{\sin^2\theta\,(u\cos\theta+1)}{u\cos\theta-\cos 2\theta}\right)^{d/2},\ n^{\rho_{\mathrm{u}}} = \left(\frac{\sin^2\theta}{1-\cot^2\theta\,(u^2-2u\cos\theta+1)}\right)^{d/2}. \tag{10}$$

In the above lemma, the parameter $u \in [\cos\theta, 1/\cos\theta]$ controls the trade-off between the preprocessing time and space complexity on the one hand, and the query time complexity on the other. The two extreme cases correspond to near-linear space and preprocessing with a slightly sublinear query time complexity (for $u = \cos\theta$), and very high space and preprocessing complexities with almost instant query responses (for $u = 1/\cos\theta$). The case $u = 1$ corresponds to $\rho_{\mathrm{q}} = \rho_{\mathrm{u}}$.

A.6 Lattice sieving algorithms

Heuristic lattice sieving algorithms for solving SVP are based on the following two principles: (1) if $\boldsymbol{v}, \boldsymbol{w} \in \mathcal{L}$, then their sum/difference $\boldsymbol{v} \pm \boldsymbol{w}$ is also a lattice vector; and (2) if we have a sufficiently long list L of lattice vectors, then we expect there to be pairs $\boldsymbol{v}, \boldsymbol{w} \in L$ with $\|\boldsymbol{v} \pm \boldsymbol{w}\| < \|\boldsymbol{v}\|, \|\boldsymbol{w}\|$. This intuitively describes the approach: we first generate a sufficiently long list of lattice vectors, and then keep combining pairs of vectors in our list to form shorter and shorter lattice vectors until we (hopefully) find a shortest lattice vector in our list.

To make sure the algorithm makes progress in finding shorter lattice vectors, L needs to contain exponentially many lattice vectors; for vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathcal{L}$ of similar norm, the vector $\boldsymbol{v} - \boldsymbol{w}$ is shorter than $\boldsymbol{v}, \boldsymbol{w}$ if the angle between $\boldsymbol{v}, \boldsymbol{w}$ is smaller than $\pi/3$, which for random vectors $\boldsymbol{v}, \boldsymbol{w}$ of similar norm would occur with probability $(3/4)^{d/2+o(d)}$. Under the aforementioned heuristic assumption, that when normalized, vectors in L follow the same distribution as vectors sampled uniformly at random from the unit sphere, this then also models the probability that two vectors in our list can reduce one another.

The expected space complexity of heuristic sieving algorithms follows from the previous observation: if we sample $(4/3)^{d/2+o(d)}$ vectors uniformly at random from the unit sphere, then we expect a significant number of pairs of vectors to have angle less than $\pi/3$, leading to many short difference vectors. Therefore, if we start by sampling a list L of $(4/3)^{d/2+o(d)}$ rather long lattice vectors, and iteratively consider combinations of vectors in L to find shorter vectors (and replace the longer vector with the shorter combination), we expect to keep making progress. Combining all pairs of vectors in a list of size $(4/3)^{d/2+o(d)} \approx 2^{0.208d+o(d)}$ naively takes time $(4/3)^{d+o(d)} \approx 2^{0.415d+o(d)}$.

The Nguyen–Vidick sieve. The heuristic sieve of Nguyen and Vidick [68] starts by sampling a list L of $(4/3)^{d/2+o(d)}$ reasonably long lattice vectors, sampled from a discrete Gaussian $D_{\mathcal{L},s}$ with the standard deviation s chosen such that (1) we can efficiently sample from this distribution, and (2) the returned vectors are at most of norm $2^{O(d)}\lambda_1(\mathcal{L})$. Then we use a *sieve* to map L, with some maximum norm $R := \max_{\boldsymbol{v} \in L} \|\boldsymbol{v}\|$, to a new list L', with maximum norm at most $R' := \gamma R$ for a geometric factor $0 \ll \gamma < 1$ close to 1. By repeatedly applying this sieve operation, after $\mathrm{poly}(d)$ iterations we expect to find a long list of lattice vectors of norm at most $\gamma^{\mathrm{poly}(d)}R = O(\lambda_1(\mathcal{L}))$, which then (with high probability) contains a shortest vector in the lattice.

Algorithm 2 describes a variant of Nguyen–Vidick's original sieve, to map L to L' in $|L|^2$ time (ignoring costs polynomial in d). The presented algorithm is a more intuitive version of the original sieve; see [50, Appendix B] for details on this equivalence. Without any further modifications to this algorithm, the heuristic complexities for solving SVP with this method are as follows [68, Section 4].

Lemma 3 (Complexities of the Nguyen–Vidick sieve). *Heuristically, the Nguyen–Vidick sieve solves SVP in space* S *and time* T*, with*

$$\mathrm{S} = (4/3)^{d/2+o(d)} \approx 2^{0.208d+o(d)}, \qquad \mathrm{T} = (4/3)^{d+o(d)} \approx 2^{0.415d+o(d)}. \tag{11}$$

Algorithm 2 The Nguyen–Vidick sieve for finding shortest vectors [68]

Require: An LLL-reduced basis B of a lattice $\mathcal{L}(B)$
Ensure: The algorithm finds a shortest lattice vector
1: Initialize empty lists L, L' and set $\gamma \leftarrow 1 - 1/d$
2: Sample $(4/3)^{d/2+o(d)}$ lattice vectors and add them to L
3: Set $R \leftarrow \max_{\boldsymbol{w} \in L} \|\boldsymbol{w}\|$
4: **repeat**
5: **for each** $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$ **do** ▷ *NNS techniques can be used to speed this up*
6: **if** $\|\boldsymbol{w}_1 - \boldsymbol{w}_2\| < \gamma R$ **then**
7: Add $\boldsymbol{w}_1 - \boldsymbol{w}_2$ to the list L'
8: Replace $L \leftarrow L'$, set $L' \leftarrow \emptyset$, and recompute $R \leftarrow \max_{\boldsymbol{w} \in L} \|\boldsymbol{w}\|$
9: **until** L contains a shortest lattice vector
10: **return** $\operatorname{argmin}_{\boldsymbol{0} \neq \boldsymbol{v} \in L} \|\boldsymbol{v}\|$

By applying more sophisticated techniques for indexing the list L and searching for pairs of vectors that can be combined to form shorter vectors, the time complexity can be further reduced to $(3/2)^{d/2+o(d)} \approx 2^{0.292d+o(d)}$ [18]. Using a trick first described in [20], this can be done without increasing the space complexity, i.e., maintaining an asymptotic space complexity of only $2^{0.208d+o(d)}$, as the following result (a restatement of results from [18, Section 7]) shows.

Lemma 4 (Complexities of the optimized Nguyen–Vidick sieve). *The Nguyen–Vidick sieve with the spherical locality-sensitive filters of Becker–Ducas–Gama–Laarhoven heuristically solves SVP in space* S *and time* T*, with*

$$\mathrm{S} = (4/3)^{d/2+o(d)} \approx 2^{0.208d+o(d)}, \qquad \mathrm{T} = (3/2)^{d/2+o(d)} \approx 2^{0.292d+o(d)}. \tag{12}$$

Micciancio and Voulgaris' GaussSieve. Micciancio and Voulgaris used a slightly different approach in their GaussSieve algorithm [65]. This algorithm reduces the memory footprint by immediately *reducing* all pairs of lattice vectors that can be combined to form shorter lattice vectors. The algorithm uses a single list L, which is continuously kept in a state where for all $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$, $\|\boldsymbol{w}_1 \pm \boldsymbol{w}_2\| \geq \|\boldsymbol{w}_1\|, \|\boldsymbol{w}_2\|$. Each time a new vector $\boldsymbol{v} \in \mathcal{L}$ is sampled, its norm is reduced with vectors in L by adding/subtracting vectors $\boldsymbol{w} \in L$ which would lead to a shorter vector, and vectors in the list are also reduced with the vector $\boldsymbol{v}$. After $\boldsymbol{v}$ can no longer be reduced with L, $\boldsymbol{v}$ is finally added to the list, guaranteeing that the pairwise reduction property is maintained. Modified list vectors are added to a stack to be reconsidered later. Algorithm 3 describes this procedure in pseudocode.

By immediately reducing all pairs of vectors, the GaussSieve achieves significantly better practical time and space complexities than the Nguyen–Vidick sieve. At the same time however, Nguyen and Vidick's (heuristic) proof technique does not apply to the GaussSieve, and there is no proven theoretical bound on the time complexity of the GaussSieve, even using heuristic assumptions. However, it is commonly believed that the Nguyen–Vidick sieve and the GaussSieve have

Algorithm 3 The GaussSieve algorithm for finding shortest vectors [65]

Require: A basis B of a lattice $\mathcal{L}(B)$
Ensure: The algorithm outputs a shortest non-zero lattice vector
Initialize an empty list L and an empty stack S
repeat
  Get a vector $\boldsymbol{v}$ from the stack (or sample a new one if $S = \emptyset$)
  for each $\boldsymbol{w} \in L$ **do** ▷ *NNS techniques can be used to speed this up*
    if $\|\boldsymbol{v} - \boldsymbol{w}\| < \|\boldsymbol{v}\|$ **then**
      Replace $\boldsymbol{v} \leftarrow \boldsymbol{v} - \boldsymbol{w}$
    if $\|\boldsymbol{w} - \boldsymbol{v}\| < \|\boldsymbol{w}\|$ **then**
      Replace $\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{v}$
      Move $\boldsymbol{w}$ from the list L to the stack S (unless $\boldsymbol{w} = \boldsymbol{0}$)
  if $\boldsymbol{v}$ has changed **then**
    Add $\boldsymbol{v}$ to the stack S (unless $\boldsymbol{v} = \boldsymbol{0}$)
  else
    Add $\boldsymbol{v}$ to the list L (unless $\boldsymbol{v} = \boldsymbol{0}$)
until L contains a shortest lattice vector
return $\operatorname{argmin}_{\boldsymbol{0} \neq \boldsymbol{v} \in L} \|\boldsymbol{v}\|$

the same heuristic asymptotic space and time complexities, i.e., using $2^{0.208d+o(d)}$ space and $2^{0.415d+o(d)}$ time without any further modifications, and using only $2^{0.292d+o(d)}$ time using nearest neighbor searching [18]. However, to apply nearest neighbor techniques to the GaussSieve, the space complexity would increase to $2^{0.292d+o(d)}$ as well, as the same trick from [20, 50] cannot be applied here.

A.7 Voronoi cells

We recall some definitions and results regarding the Voronoi cell of a lattice from [4, 64, 73, 76]. First, we give a formal definition of Voronoi cells below, which are essentially the enclosing regions of points closer to the origin than to any other lattice point.

Definition 8 (Voronoi cell of a lattice). *The Voronoi cell of a lattice $\mathcal{L}$ is defined as the region $\mathcal{V} \subset \mathbb{R}^d$ such that $\boldsymbol{v} \in \mathcal{V}$ iff $\|\boldsymbol{v}\| \leq \|\boldsymbol{v} - \boldsymbol{x}\|$ for all $\boldsymbol{x} \in \mathcal{L}$. In other words:*

$$\mathcal{V} := \bigcap_{\boldsymbol{r} \in \mathcal{L}} \mathcal{H}(\boldsymbol{r}). \tag{13}$$

An important property of Voronoi cells, which immediately follows from the definition, is that the closest vector to a target vector $\boldsymbol{t} \in \mathbb{R}^d$ in a lattice $\mathcal{L}$ is the vector $\boldsymbol{s} \in \mathcal{L}$ if and only if $\boldsymbol{t} \in \boldsymbol{s} + \mathcal{V}$. In particular, the latter condition is equivalent to $\boldsymbol{t} - \boldsymbol{s} \in \mathcal{V}$, which indicates that if we can find a point $\boldsymbol{t}' \in (\boldsymbol{t} + \mathcal{L}) \cap \mathcal{V}$ (i.e., $\boldsymbol{t}' = \boldsymbol{t} - \boldsymbol{s}$), then we have found a solution to CVP for $\boldsymbol{t}$ as $\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}'$.

In (13) above, we see that the Voronoi cell can be described in terms of an infinite number of half-spaces generated by the vectors in the lattice $\mathcal{L}$. In reality,

the Voronoi cell of a lattice is a convex polytope with only a bounded number of facets, and its facets are closely related to what are commonly known as the *relevant vectors*, defined below.

Definition 9 (Relevant vectors). *Given a lattice $\mathcal{L}$, a vector $\boldsymbol{r} \in \mathcal{L}$ is a* relevant vector *of the lattice $\mathcal{L}$ if and only if $\mathcal{V}$ and $\boldsymbol{r} + \mathcal{V}$ share a non-empty boundary. We denote the set of all relevant vectors by $\mathcal{R} \subseteq \mathcal{L}$.*

The relevant vectors of the lattice shape the boundary of $\mathcal{V}$, and the set $\mathcal{R}$ can be seen as a more compact way of representing/storing the Voronoi cell of a lattice, as $\mathcal{V}$ can be equivalently described by the following equation:

$$\mathcal{V} = \bigcap_{\boldsymbol{r} \in \mathcal{R}} \mathcal{H}(\boldsymbol{r}). \tag{14}$$

In other words, the Voronoi cell is also equal to the intersection of half-spaces generated only by the relevant vectors $\boldsymbol{r} \in \mathcal{R}$. The set $\mathcal{R}$ is by definition the minimal set $S \subseteq \mathcal{L}$ with the property that $\mathcal{V} = \bigcap_{\boldsymbol{r} \in S} \mathcal{H}(\boldsymbol{r})$ – other vectors do not contribute to the shape of the Voronoi cell, and removing any vector from $\mathcal{R}$ would result in a different, larger enclosed region.

To efficiently describe and store the Voronoi cell of a lattice, it is important to know that $\mathcal{R}$ is finite and cannot be too large. Fortunately the size of $\mathcal{R}$ can be bounded (in the worst-case) as follows; see, e.g., [64, Corollary 2.5] for a proof.

Lemma 5 (Number of relevant vectors). *For arbitrary lattices $\mathcal{L}$, the set $\mathcal{R}$ of relevant vectors satisfies $|\mathcal{R}| \leq 2^{d+1}$.*

As a result, a description of the Voronoi cell of a lattice can be stored in $2^{d+o(d)}$ memory, by storing all the relevant vectors. An example of a Voronoi cell for a two-dimensional lattice, as well as the related relevant vectors, is given in the introduction in Figure 1a. On the negative side, note that a storage requirement of the order 2^d means it is infeasible to store the exact Voronoi cell of lattices in dimensions higher than 80. This in contrast to heuristic sieving algorithms, whose space requirements of the order $2^{0.21d}$ means these algorithms can still be used in higher dimensions as well.

A.8 Voronoi cell algorithms

As stated above, a crucial property of Voronoi cells, highlighting their relevance for closest point searching, is that $\boldsymbol{s} \in \mathcal{L}$ is the closest point to a target vector $\boldsymbol{t} \in \mathbb{R}^d$ if and only if $\boldsymbol{t} - \boldsymbol{s} \in \mathcal{V}$. Since $\boldsymbol{t} - \boldsymbol{s} \in \boldsymbol{t} + \mathcal{L}$, a common approach of Voronoi cell algorithms for finding closest points to target vectors $\boldsymbol{t}$ is to start with $\boldsymbol{t}$ and gradually move along the coset $\boldsymbol{t} + \mathcal{L}$ towards the origin (by adding/subtracting lattice vectors to our current vector in the coset $\boldsymbol{t} + \mathcal{L}$). When no shorter vector in $\boldsymbol{t} + \mathcal{L}$ exists than our current guess $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$, we know that $\boldsymbol{t}' \in \mathcal{V}$ and therefore $\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}'$ is the closest lattice point to $\boldsymbol{t}$.

Building upon work of Sommer, Feder and Shalvi [73], Micciancio and Voulgaris [64] described algorithms for constructing the Voronoi cell of a lattice (or

equivalently, the set of $2^{d+o(d)}$ relevant vectors of the lattice), and with proven time complexities at most $2^{2d+o(d)}$ this allowed them to solve SVP, CVP and CVPP. Bonifas and Dadush [25] later showed how to improve the time complexity for CVPP to only $2^{d+o(d)}$, by bounding the number of iterations in the post-processing stage to poly(d) rather than $2^{d+o(d)}$.

An important technique for finding closest vectors, using the list of relevant vectors to peform the aforementioned procedure of finding a point $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$, is the iterative slicer of Sommer–Feder–Shalvi [73] given in Algorithm 4. Given a target vector $\boldsymbol{t}$ and the Voronoi cell of the lattice as input, within a finite number of steps [73, Theorem 1] this algorithm terminates and finds the closest vector to any target $\boldsymbol{t} \in \mathbb{R}$. Micciancio–Voulgaris later showed that by selecting relevant vectors for reduction in a specific order, the number of iterations can be bounded by $2^{d+o(d)}$ [64, Lemma 3.2].

Algorithm 4 The iterative slicer for finding closest vectors [73]

Require: The relevant vectors $\mathcal{R} \subset \mathcal{L}$ and a target $\boldsymbol{t} \in \mathbb{R}^d$
Ensure: The algorithm outputs a closest lattice vector $\boldsymbol{s} \in \mathcal{L}$ to $\boldsymbol{t}$
1: Initialize $\boldsymbol{t}' \leftarrow \boldsymbol{t}$
2: **for each** $\boldsymbol{r} \in \mathcal{R}$ **do**
3: **if** $\|\boldsymbol{t}' - \boldsymbol{r}\| < \|\boldsymbol{t}'\|$ **then**
4: Replace $\boldsymbol{t}' \leftarrow \boldsymbol{t}' - \boldsymbol{r}$ and restart the **for**-loop
5: **return** $\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}'$

By similar techniques as in heuristic lattice sieving (or as in [25]), one can bound the number of iterations of this slicer until termination. Given a target $\boldsymbol{t}$, one can use Babai rounding on an LLL-reduced basis of the lattice to get an initial guess $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$ satisfying $\|\boldsymbol{t}'\| \leq 2^{O(d)} \min_{\boldsymbol{s} \in \boldsymbol{t}+\mathcal{L}} \|\boldsymbol{s}\|$. Then, by only performing reductions whenever $\|\boldsymbol{t}' \pm \boldsymbol{r}\| < \gamma \|\boldsymbol{t}'\|$ for some geometric factor $\gamma = 1 - 1/d^k$ for certain $k > 1$, one can ensure that the number of iterations is polynomially bounded by $\log_{1/\gamma} \|\boldsymbol{t}'\| = O(d^{1+k})$. At the same time, due to this geometric factor γ, after the algorithm terminates we might only have $\boldsymbol{t}' \in (1/\gamma)\mathcal{V}$ instead of $\boldsymbol{t}' \in \mathcal{V}$. Since $\mathrm{vol}(\mathcal{V}/\gamma) = \mathrm{vol}(\mathcal{V})/\gamma^d$, we therefore expect this algorithm to succeed with probability proportional to $\gamma^d = 1 - O(d^{1-k}) = 1 - o(1)$ over the randomness of $\boldsymbol{t}$. As k increases, the (polynomial) number of iterations increases, while heuristically the success probability becomes overwhelming.

Bonifas and Dadush [25] described a different method to bound the number of iterations to poly(d), by carefully choosing which relevant vectors to use for reduction in each step. Similar to there being no formal proof that other approaches allow for solving exact CVP, in the remainder of this paper we will assume (heuristically) that the number of iterations of this slicer (until termination) is only poly(d), for random lattices and average-case target vectors.

B Approximate Voronoi cells

Our approach for solving CVPP and its variants can best be described in terms of *approximate Voronoi cells*. These are in a sense similar to the intermediate "raw diamonds" described by Viterbo–Biglieri [76], before the diamond-cutting algorithm completes and returns the exact Voronoi cell. Below we start with a formal definition of approximate Voronoi cells, where as before we write $\mathcal{H}(\boldsymbol{x})$ for half-spaces whose boundaries are orthogonal to $\boldsymbol{x}$ and pass through $\frac{1}{2}\boldsymbol{x}$.

Definition 10 (Approximate Voronoi cells). *For a lattice $\mathcal{L}$ and a list $L \subseteq \mathcal{L}$, the approximate Voronoi cell generated by L is defined as:*

$$\mathcal{V}_L := \bigcap_{\boldsymbol{r} \in L} \mathcal{H}(\boldsymbol{r}). \tag{15}$$

Note that $\mathcal{V} \subseteq \mathcal{V}_L$ for any $L \subseteq \mathcal{L}$, and $\mathcal{V} = \mathcal{V}_L$ if and only if $\mathcal{R} \subseteq L$ [73, Lemma 5]. Similarly, $\mathcal{R}$ is the smallest set $L \subseteq \mathcal{L}$ with the property that $\mathcal{V}_L = \mathcal{V}$. To quantize the 'quality' of an approximate Voronoi cell $\mathcal{V}_L$ (or a list L), recall that the volume of the exact Voronoi cell $\mathcal{V}$ is equal to the volume of the lattice: $\mathrm{vol}(\mathcal{V}) = \det(\mathcal{L})$. If $\mathcal{R}$ is not contained in L, then $\mathcal{V}_L$ will have a strictly larger volume, and the following quantity can therefore serve as a guideline as to how well an approximate Voronoi cell $\mathcal{V}_L$ approximates $\mathcal{V}$.

Definition 11 (Approximation factor). *Given a lattice $\mathcal{L}$ and a list $L \subseteq \mathcal{L}$, we define the approximation factor A_L for the cell $\mathcal{V}_L$ generated by L as:*

$$A_L := \frac{\mathrm{vol}(\mathcal{V}_L)}{\mathrm{vol}(\mathcal{V})}\,. \tag{16}$$

Clearly $A_L \geq 1$ with equality iff $\mathcal{R} \subseteq L$. If L is very small, A_L may be infinite, but as long as L contains a basis of the lattice one has $A_L < \infty$ [76]. For arbitrary lists L, L' with $L \subseteq L' \subseteq \mathcal{L}$ we have $A_{L'} \leq A_L$, i.e., if we add vectors to L to form L', the approximation factor either stays the same or decreases.

B.1 Good approximations

The main result of [52] for solving CVPP can be summarized in terms of approximate Voronoi cells by the following lemma, stating how big L must heuristically be to obtain approximation factors close to 1.

Lemma 6 (Good approximations). *Let L consist of the $\alpha^{d+o(d)}$ shortest vectors of a lattice $\mathcal{L}$, with $\alpha \geq \sqrt{2} + o(1)$. Then heuristically,*

$$A_L = 1 + o(1). \tag{17}$$

In other words, if L contains the $2^{d/2+o(d)}$ shortest lattice vectors of a random lattice $\mathcal{L}$, we expect $\mathcal{V}_L$ to approximate the exact Voronoi cell $\mathcal{V}$ very well. This in contrast with the best proven worst-case bounds, which suggest that up to $2^{d+o(d)}$ vectors are needed to accurately represent the Voronoi cell of a lattice.

Heuristically, Lemma 6 implies that if we use Voronoi cell algorithms for CVP(P), using only the $2^{d/2+o(d)}$ shortest lattice vectors as our approximate list of relevant vectors (instead of all $2^{d+o(d)}$ relevant vectors), the algorithm will still succeed with high probability in returning the actual closest vector to random target vectors. The resulting heuristic slicer, which serves as the algorithm for the query phase of CVPP in [52], is given in Algorithm 5.

Algorithm 5 The heuristic slicer for finding closest vectors [52]

Require: A list $L \subset \mathcal{L}$ of the $2^{d/2+o(d)}$ shortest vectors of $\mathcal{L}$, and a target $\boldsymbol{t} \in \mathbb{R}^d$
Ensure: The algorithm outputs a closest lattice vector $\boldsymbol{s} \in \mathcal{L}$ to $\boldsymbol{t}$

```
Initialize t' ← t
for each r ∈ L do                          ▷ NNS speedups can be used here
    if ‖t' − r‖ < ‖t'‖ then
        Replace t' ← t' − r and restart the for-loop
return s ← t − t'
```

Assuming the list L contains the $2^{d/2+o(d)}$ shortest vectors of $\mathcal{L}$, it returns the closest vector with high probability. Naively, this algorithm has query time and space complexities of $2^{d/2+o(d)}$, but with nearest neighbor search techniques the search for relevant vectors that can reduce $\boldsymbol{t}'$ can be sped up significantly.

B.2 Arbitrary approximations

To obtain improved results compared to [52], we will use the following generalization of Lemma 6, providing a heuristic upper bound on the approximation factor A_L for lists L containing fewer than $2^{d/2+o(d)}$ lattice vectors. The heuristic proof/derivation of this result may not be tight, and improving upon the given upper bound is left as an open problem. Note that a tighter upper bound on the approximation factor would immediately result in better complexities for CVPP in Figure 2.

Lemma 7 (Arbitrary approximations). *Let L consist of the $\alpha^{d+o(d)}$ shortest vectors of a lattice $\mathcal{L}$, with $\alpha \in (1.03396, \sqrt{2})$. Then heuristically,*

$$A_L \leq \left(\frac{16\alpha^4 \left(\alpha^2 - 1\right)}{-9\alpha^8 + 64\alpha^6 - 104\alpha^4 + 64\alpha^2 - 16} \right)^{d/2+o(d)} . \tag{18}$$

Above, $1.03396\ldots$ is a root of the polynomial in the denominator, or equivalently the smallest root $x > 1$ of $3x^4 + 8x^3 - 8x - 4$. Note that as $\alpha \to \sqrt{2}$, the ratio approaches 1, and Lemma 7 therefore is a proper generalization of Lemma 6. For $\alpha \downarrow 1.03396\ldots$, the ratio tends to ∞, suggesting that for very small lists our analysis does not provide a proper, meaningful upper bound on the approximation factor – there is no reason to believe that $A_L = \infty$ for exponentially large preprocessed lists.

For random target vectors, we heuristically expect the probability of success of finding a closest vector to this target with a preprocessed list L to be approximately $p_L = 1/A_L$ – assuming a reduced vector returned by the slicer lies uniformly in $\mathcal{V}_L$, the probability that it also lies in $\mathcal{V}$ is proportional to $\mathrm{vol}(\mathcal{V})/\mathrm{vol}(\mathcal{V}_L)$. With the above result in mind, we can thus generalize the previous heuristic slicer to construct a CVPP algorithm which works with even smaller lists L, but has an exponentially small success probability in the dimension d. This is somewhat unsatisfactory, as an algorithm succeeding with exponentially small success probability is unlikely to be useful in any application. However, similar to extreme pruning in enumeration [40], as long as the gain in the time (and space) complexity is more than the loss in the success probability, such an algorithm may well turn out to be useful if we can somehow randomize our reduction algorithm.

B.3 Randomized slicing

To instantiate Lemma 7 with an actual CVPP algorithm succeeding with high probability, ideally we need to be able to rerandomize problem instances such that, if an algorithm succeeds with small probability p in time T, we can repeat the algorithm approximately $1/p$ times to obtain an algorithm succeeding with constant probability in T/p time. The (heuristic) iterative slicer mostly works deterministically[4], so if an initialized data structure and problem instance fail, running the same slicing algorithm on the same target would likely result in failure again.

To rerandomize problem instances, we will use the following procedure: instead of starting with a single vector $\boldsymbol{t}' \leftarrow \boldsymbol{t}$ and attempting to reduce it to a vector $\boldsymbol{t}'' \in \mathcal{V}$, we use several vectors of the form $\boldsymbol{t}' \sim D_{\boldsymbol{t}+\mathcal{L},s}$ sampled from a discrete Gaussian over the coset $\boldsymbol{t} + \mathcal{L}$ with a well-chosen parameter s. This parameter s needs to be chosen large enough so that sampling can be done in polynomial time in the lattice dimension d, and small enough so that the sampled vectors are not too long, and so that the reductions of $\boldsymbol{t}'$ to short vectors $\boldsymbol{t}''$ do not take too long. To analyze our CVPP method using this sampling procedure, we require the heuristic assumption 1 given in the introduction, essentially stating that this rerandomization procedure works perfectly.

For intuition behind this heuristic assumption, recall that with a preprocessed list L, a vector $\boldsymbol{t}'$ will ultimately be reduced by the slicer to a vector $\boldsymbol{t}'' = \mathrm{Slice}(\boldsymbol{t}') \in \mathcal{V}_L$. If $\boldsymbol{t}''$ now also lies in $\mathcal{V}$, which we may heuristically model as a sphere of radius $\lambda_1(\mathcal{L})$, then $\boldsymbol{0}$ is the closest lattice vector to $\boldsymbol{t}'' \in \boldsymbol{t} + \mathcal{L}$, and so $\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}''$ is indeed the closest lattice vector to $\boldsymbol{t}$. However, since $\mathcal{V}_L$ is potentially much larger than $\mathcal{V}$, this may only occur with small probability, and heuristically we essentially assume that $\Pr_{\boldsymbol{t}'' \leftarrow \mathrm{Slice}(\boldsymbol{t}')}[\boldsymbol{t}'' \in \mathcal{V} \mid \boldsymbol{t}'' \in \mathcal{V}_L] = 1/A_L$.

[4] Observe that there is some room for randomization within the slicing algorithm itself: if an intermediate vector $\boldsymbol{t}'$ can be reduced with two vectors $\boldsymbol{v}_1, \boldsymbol{v}_2 \in L$ to form a shorter vector, one could randomly choose either option for potentially different outcomes of the slicer.

Note that the probability on the left hand side of (1) is for arbitrary, fixed target vectors $\boldsymbol{t}$, and the randomness is purely over the Gaussian sampling of $\boldsymbol{t}' \sim D_{\boldsymbol{t}+\mathcal{L},s}$ – we heuristically expect that we can effectively apply this rerandomization procedure to *any* target vector, rather than always failing for certain targets and succeeding for other targets. In Appendix C we will show that indeed, experimentally it seems that it is a reasonable assumption to assume that this rerandomization procedure works for any target vector and not just for a small fraction of target vectors.

Assuming the above heuristic assumption holds, an algorithm for CVPP follows, by repeating the slicing algorithm on randomly sampled vectors $\boldsymbol{t}' \sim D_{\boldsymbol{t}+\mathcal{L},s}$. The randomized heuristic slicer that we obtain is given in Algorithm 1.

For randomized slicing, the costs of the algorithm are mostly the same in terms of α as for a single run of the slicer, except that the (expected) time complexity T_2 for the query phase is multiplied by a factor $1/p$, to account for the expected number of trials necessary to find a closest vector. On the positive side, this means that we do not need to fix $\alpha = \sqrt{2}$ in advance, and can obtain significantly better space complexities, as well as better time–space trade-offs.

B.4 Preprocessing costs

For the preprocessing phase, we need to generate a list of the $\alpha^{d+o(d)}$ lattice vectors of norm at most $\alpha \cdot \lambda_1(\mathcal{L})$, and store it in a nearest neighbor data structure to allow for fast searching. This preprocessing step can be done using different methods. In moderate dimensions, the fastest way may be to use lattice enumeration [40, 48], but in high dimensions (as well asymptotically) heuristic lattice sieving methods will lead to the best complexities. As we are mostly interested in obtaining the best asymptotic complexities here, let us consider the preprocessing costs for a sieving-based preprocessing stage.

Recall that with standard heuristic sieving methods, we reduce pairs of lattice vectors if their angle is at most $\theta = \frac{\pi}{3}$, resulting in a list of size $(\sin\theta)^{-d+o(d)}$. To generate a list of the $\alpha^{d+o(d)}$ shortest lattice vectors of a lattice $\mathcal{L}$ with the GaussSieve, rather than the $(4/3)^{d/2+o(d)}$ lattice vectors one would normally get, we relax the reduction step in sieving: we reduce a list vector $\boldsymbol{v}$ with another list vector $\boldsymbol{w}$ if and only if their pairwise angle is less than $\theta = \arcsin(1/\alpha)$, which for vectors $\boldsymbol{v}, \boldsymbol{w}$ of similar norm corresponds to the following condition:

$$\|\boldsymbol{v} - \boldsymbol{w}\|^2 < 2(1 - \cos\theta) \cdot \|\boldsymbol{v}\|^2 = \left(2 - \frac{2}{\alpha}\sqrt{\alpha^2 - 1}\right) \cdot \|\boldsymbol{v}\|^2. \tag{19}$$

This leads to the modified GaussSieve described in Algorithm 6. Note that for $\alpha = \sqrt{2}$, the reduction criterion becomes $\|\boldsymbol{v} - \boldsymbol{w}\| < \sqrt{2 - \sqrt{2}} \cdot \|\boldsymbol{v}\|$.

Intuitively, Algorithm 6 could be interpreted as a relaxed version of standard heuristic sieving approaches. In standard sieving, pairwise reductions are *always* performed, even if they lead to minor progress in reducing the norms of the vectors. This turns out to lead to the smallest list sizes. By only reducing vectors when significant progress is made in reducing their norms, reductions occur less

Algorithm 6 The GaussSieve-based preprocessing phase for solving CVPP

Require: A basis B of a lattice $\mathcal{L}(B)$, a parameter $\alpha \geq \sqrt{4/3}$
Ensure: The output list $L \subset \mathcal{L}$ contains $\alpha^{d+o(d)}$ vectors of norm at most $\alpha \cdot \lambda_1(\mathcal{L})$

Initialize an empty list L and an empty stack S
repeat
  Get a vector $\boldsymbol{v}$ from the stack (or sample a new one if $S = \emptyset$)
  for each $\boldsymbol{w} \in L$ **do** ▷ *NNS can be used to speed this up*
    if $\|\boldsymbol{v} - \boldsymbol{w}\|^2 < (2 - \frac{2}{\alpha}\sqrt{\alpha^2 - 1}) \cdot \|\boldsymbol{v}\|^2$ **then**
      Replace $\boldsymbol{v} \leftarrow \boldsymbol{v} - \boldsymbol{w}$
    if $\|\boldsymbol{w} - \boldsymbol{v}\|^2 < (2 - \frac{2}{\alpha}\sqrt{\alpha^2 - 1}) \cdot \|\boldsymbol{w}\|^2$ **then**
      Replace $\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{v}$
      Move $\boldsymbol{w}$ from the list L to the stack S (unless $\boldsymbol{w} = \boldsymbol{0}$)
  if $\boldsymbol{v}$ has changed **then**
    Add $\boldsymbol{v}$ to the stack S (unless $\boldsymbol{v} = \boldsymbol{0}$)
  else
    Add $\boldsymbol{v}$ to the list L (unless $\boldsymbol{v} = \boldsymbol{0}$)
until $\boldsymbol{v}$ is a shortest vector
return L

frequently, leading to larger list sizes before real progress is made. By not always doing reductions in Algorithm 6, it takes longer to complete this preprocessing step, but a longer list of lattice vectors is returned. Moreover, by only searching for vectors with very small angles, the speed-ups obtained from applying nearest neighbor techniques become bigger as well.

Note that in Algorithm 6, as well as other lattice sieving algorithms in Appendix A, the stopping criterion is stated as continuing until the list contains a shortest vector. In the literature on lattice sieving, many different stopping criterions have been considered, often involving a bound on the number of "collisions" to the all-zero vector. An alternative stopping criterion here might also be to continue until, say, at least 90% of the expected number of lattice vectors below a certain norm $\alpha \cdot \lambda_1(\mathcal{L})$ have been encountered by the sieve.[5] In reality, the precise termination condition is somewhat irrelevant – at some point during the run, the list will be a very good quality and contain most short vectors, and continuing a bit longer only means that slightly more time is spent on the preprocessing phase, and slightly more of the shortest lattice vectors will be in the preprocessed list for the query phase.

The following lemma summarizes the preprocessing costs obtained by using the optimized nearest neighbor techniques from Lemma 2.

Lemma 8 (Preprocessing complexities). *Let $\alpha \in (1, \infty)$ and suppose that $u \in (\sqrt{\frac{\alpha^2-1}{\alpha^2}}, \sqrt{\frac{\alpha^2}{\alpha^2-1}})$. With Algorithm 6, we can heuristically generate a list of the $\alpha^{d+o(d)}$ shortest vectors in a lattice $\mathcal{L}$ with the following space and time*

[5] As described in Figure 6a later on, indeed the preprocessing step finds a large fraction of all lattice vectors below the given norm.

complexities S_1 *and* T_1*.*

$$\mathrm{S}_1 = \max\left\{\mathrm{S}_2,\ \left(\frac{4}{3}\right)^{d/2+o(d)}\right\}, \tag{20}$$

$$\mathrm{T}_1 = \max\left\{\mathrm{S}_2,\ \left(\frac{3}{2}\right)^{d/2+o(d)}\right\}, \tag{21}$$

$$\mathrm{S}_2 = \left(\frac{\alpha}{\alpha - (\alpha^2-1)(\alpha u^2 - 2u\sqrt{\alpha^2-1}+\alpha)}\right)^{d/2+o(d)}. \tag{22}$$

Moreover, at the end of the preprocessing step, we have a data structure of size S_2 *which can answer CVP queries in time* T_2 *as follows:*

$$\mathrm{T}_2 = \left(\frac{\alpha + u\sqrt{\alpha^2-1}}{-\alpha^3 + \alpha^2 u\sqrt{\alpha^2-1}+2\alpha}\right)^{d/2+o(d)}. \tag{23}$$

Observe that reductions between vectors only make sense if vectors get shorter; if $\boldsymbol{v}$ and $\boldsymbol{w}$ have similar norms, then one cannot reduce $\boldsymbol{v}$ with $\boldsymbol{w}$ if their pairwise angle is larger than $\frac{\pi}{3}$. To generate a list with $\alpha^{d+o(d)}$ short vectors with $\alpha < \sqrt{4/3}$, one can just run the algorithm for $\alpha = \sqrt{4/3}$ (corresponding to the regular GaussSieve), and afterwards discard lattice vectors which are too long. Alternatively, if one is interested in minimizing the memory complexity, for small values α one could consider using tuple lattice sieving approaches discussed in [17,43,44]. We restrict our attention to sieving using pairwise reductions, and we leave a complexity analysis based on tuple reductions to future work.

Note also that S_1 and T_1 in Lemma 8 are lower bounded by the costs for solving SVP, which based on the current best space and time complexities for (pairwise) sieving are $(4/3)^{d/2+o(d)}$ and $(3/2)^{d/2+o(d)}$ respectively [18]. Using tuple sieving [17, 43, 44], it is possible to eliminate this lower bound on S_1, at the cost of worse preprocessing time complexities – we also leave this further generalization of the complexities to the interested reader.

B.5 Main results

With the previous results and techniques in place, we are now ready to state the main result. First, we restate the main result of [52] for solving exact CVPP, which says that if $\alpha \geq \sqrt{2}$ then the slicer succeeds with high probability, and we thus get the following heuristic complexities for CVPP.

Theorem 2 (Complexities for CVPP with good approximations). *[52, Theorem 2] Let* $u \in (\frac{1}{2}\sqrt{2}, \sqrt{2})$*. With a preprocessed list of size* $|L| = 2^{d/2+o(d)}$*, we can solve CVPP with preprocessing time* T_1 *and space* S_1*, and query time* T_2 *and space complexity* S_2 *as follows:*

$$\mathrm{S}_{1,2} = \mathrm{T}_1 = \left(\frac{1}{u(\sqrt{2}-u)}\right)^{d/2+o(d)}, \qquad \mathrm{T}_2 = \left(\frac{\sqrt{2}+u}{2u}\right)^{d/2+o(d)}. \tag{24}$$

The rightmost red curve in Figure 2 illustrates the time and space complexities one can achieve by building upon Lemmas 6 and 8 and Algorithms 5 and 6 with different time–space trade-offs for the NNS data structure (in particular, by varying the parameter u).

In its most general form, including both the parameter α for the preprocessed list size, as well as the nearest neighbor parameter u from Lemma 2, we finally obtain the following result.

Theorem 3 (Complexities for CVPP with arbitrary approximations). *Let $\alpha \in (1.03996, \sqrt{2})$ and $u \in (\sqrt{\frac{\alpha^2-1}{\alpha^2}}, \sqrt{\frac{\alpha^2}{\alpha^2-1}})$. With randomized slicing, we can heuristically solve CVPP with preprocessing time and space T_1 and S_1, and query time and space T_2 and S_2, where*

$$\mathrm{S}_1 = \max\left\{\mathrm{S}_2, \left(\frac{4}{3}\right)^{d/2+o(d)}\right\}, \qquad \mathrm{T}_1 = \max\left\{\mathrm{S}_2, \left(\frac{3}{2}\right)^{d/2+o(d)}\right\}, \tag{25}$$

$$\mathrm{S}_2 = \left(\frac{\alpha}{\alpha - (\alpha^2-1)(\alpha u^2 - 2u\sqrt{\alpha^2-1} + \alpha)}\right)^{d/2+o(d)}, \tag{26}$$

$$\mathrm{T}_2 = A_L \cdot \left(\frac{\alpha + u\sqrt{\alpha^2-1}}{-\alpha^3 + \alpha^2 u\sqrt{\alpha^2-1} + 2\alpha}\right)^{d/2+o(d)} \tag{27}$$

$$\leq \left(\frac{16\alpha^4\left(\alpha^2-1\right)\left(\alpha + u\sqrt{\alpha^2-1}\right)}{\left(-9\alpha^8+64\alpha^6-104\alpha^4+64\alpha^2-16\right)\left(-\alpha^3+\alpha^2 u\sqrt{\alpha^2-1}+2\alpha\right)}\right)^{d/2+o(d)}. \tag{28}$$

For $\alpha = \sqrt{2}$, Theorem 3 leads to the exact same complexities as without rerandomizations as in [52], while for instance for $\alpha = \sqrt{4/3}$ we can vary the parameter u to obtain the following trade-offs:

- Setting $u = \frac{1}{2}$ leads to $\mathrm{S}_2 \approx 2^{0.2075d+o(d)}$ and $\mathrm{T}_2 = (20/13)^{d/2+o(d)} \approx 2^{0.3107d+o(d)}$. This still offers a speed-up over (a linear number of) naive linear searches over the list, without increasing the memory.
- Setting $u = 1$ leads to $\mathrm{S}_2 \approx 2^{0.2925d+o(d)}$ and $\mathrm{T}_2 = (18/13)^{d/2+o(d)} \approx 2^{0.2347d+o(d)}$. These are the same complexities as those given in Proposition 1 in the introduction.
- Letting $u \to 2$, the preprocessing time and space complexities become $2^{\omega(d)}$, while the query time complexity becomes $2^{o(d)}$.

The time–space curve $\mathcal{C}_\alpha$ corresponding to $\alpha = \sqrt{4/3}$, as well as a few other values α, is shown in Figure 5. By taking the minimum over all these curves $\{\mathcal{C}_\alpha\}_{\alpha \in (1.03396, \sqrt{2})}$, where curves are defined by varying the parameter $u \in (\sqrt{1-1/\alpha^2}, \sqrt{1+1/(\alpha^2-1)})$, we obtain the thick blue curve in Figure 5, which is also depicted in Figure 2. There seems to be no simple expression for this curve; for a particular choice of the space complexity, the best query time complexity T_2 can be found by considering all different α, and for each α computing the value u such that the space complexity is as desired, and taking the

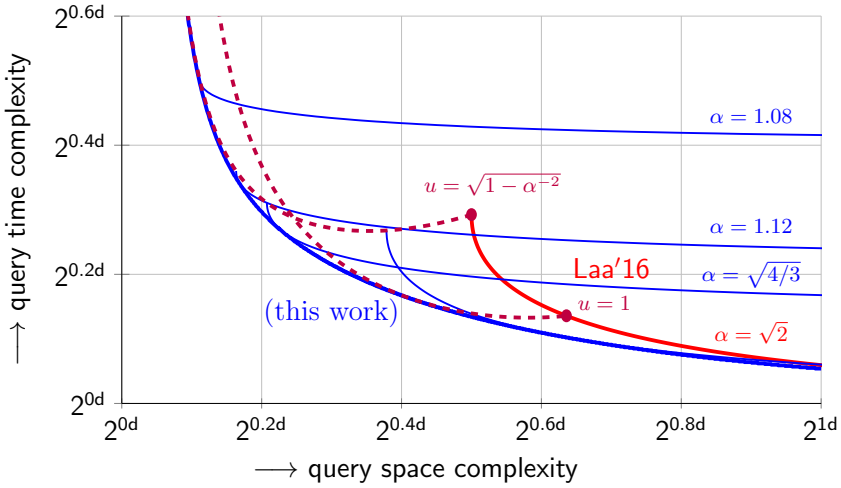


Fig. 5. Complexities for randomized slicing. Different curves correspond to different values α and different success probabilities p_α. The right blue curve corresponds to $\alpha = \sqrt{2}$ and $p_\alpha \approx 1$, i.e., not using randomized slicing as in Section B, and purple curves inbetween correspond to smaller values α with smaller values p_α. Dashed purple curves correspond to fixing the nearest neighbor parameter u and varying α. No single curve lies below all others, and the minimum over all curves is depicted by the bottom blue curve.

minimum over all these values. Note that due to the condition $\alpha > 1.03396$ (which follows from $p_\alpha > 0$), the curve terminates on the left side at a minimum space complexity of $1.03396^{d+o(d)} \approx 2^{0.0482d+o(d)}$; with this method we cannot obtain a space complexity $\mathrm{S}_2 = 2^{o(d)}$ for exact CVPP.

B.6 Proof of Lemma 7

We conclude this section with proofs of Lemma 7. Here we follow the high-level outline of the proof described in the introduction, and prove the individual claims below.

Relation between $|L|$ and $\|t'\|$. Suppose we have a target vector $\boldsymbol{t}$, which after slicing with the list L turns into $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$, and suppose that $\boldsymbol{t}'$ is no longer reducible with L. In other words, $\boldsymbol{t}'$ is contained in $\mathcal{V}_L$. First, by the Gaussian heuristic, we expect the distances from $\boldsymbol{t}$ and $\boldsymbol{t}'$ to the lattice to be approximately $\lambda_1(\mathcal{L})$. To guarantee that $\mathbf{0}$ is the closest lattice vector to the reduced vector $\boldsymbol{t}'$, so that also $\boldsymbol{t}' \in \mathcal{V}$, we therefore heuristically need $\boldsymbol{t}'$ to have norm at most approximately $\lambda_1(\mathcal{L})$ – we roughly expect $\mathcal{V}$ to have the shape of a sphere of radius $\lambda_1(\mathcal{L})$. We start with the following lemma regarding the probability of reduction between two uniformly random vectors with given norms.

Lemma 9 (Probability of the existence of a reducing vector). *Let $v, w > 0$ and let $\boldsymbol{v} = v \cdot \boldsymbol{e}_v$ and $\boldsymbol{w} = w \cdot \boldsymbol{e}_w$. Then:*

$$\Pr_{\boldsymbol{e}_v, \boldsymbol{e}_w \sim \mathcal{S}^{d-1}} \left(\|\boldsymbol{v} - \boldsymbol{w}\|^2 \leq \|\boldsymbol{v}\|^2 \right) \sim \left[1 - \left(\frac{w}{2v} \right)^2 \right]^{d/2+o(d)}. \tag{29}$$

Proof. Expanding $\|\boldsymbol{v} - \boldsymbol{w}\|^2 = v^2 + w^2 - 2vw \langle \boldsymbol{e}_v, \boldsymbol{e}_w \rangle$ and $\|\boldsymbol{v}\|^2 = v^2$, the condition $\|\boldsymbol{v} - \boldsymbol{w}\|^2 \leq \|\boldsymbol{v}\|^2$ equals $\frac{w}{2v} \leq \langle \boldsymbol{e}_v, \boldsymbol{e}_w \rangle$. The result follows from [18, Lemma 2.1]. □

For now, suppose that $|L| = \alpha^{d+o(d)}$. We will next obtain a relation between the choice of α for the input list size and the expected norm $\|\boldsymbol{t}'\| = \beta \cdot \lambda_1(\mathcal{L})$ of the reduced vector $\boldsymbol{t}'$ after the reductions with the slicer. More formally, we will show that if $\beta = \|\boldsymbol{t}'\|/\lambda_1(\mathcal{L})$ is large, we can still make progress, while if β is smaller than some threshold, we heuristically expect that we will make no progress anymore with the iterative slicer with overwhelming probability – the probability that we can still find a vector in the list to reduce the norm of $\boldsymbol{t}'$ is then exponentially small in d.

Lemma 10 (Relation between $|L|$ and $\|\boldsymbol{t}'\|$). *Let $L \subset \alpha \cdot \mathcal{S}^{d-1}$ be a list of $\alpha^{d+o(d)}$ uniformly random vectors of norm $\alpha > 1$, and let $\boldsymbol{t} \in \beta \cdot \mathcal{S}^{d-1}$ be sampled uniformly at random. Then, for high dimensions d, with non-negligible probability there exists a $\boldsymbol{v} \in L$ such that $\|\boldsymbol{t} - \boldsymbol{v}\| \leq \|\boldsymbol{t}\|$ if and only if*

$$\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 \leq 0. \tag{30}$$

Furthermore, if $\theta_{\boldsymbol{t},\boldsymbol{v}}$ denotes the angle between $\boldsymbol{t}$ and $\boldsymbol{v}$, then $\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 = 0$ and $\|\boldsymbol{t} - \boldsymbol{v}\| \leq \|\boldsymbol{t}\|$ together imply that $\theta_{\boldsymbol{t},\boldsymbol{v}} \leq \arcsin(1/\alpha)$.

Proof. By Lemma 9 we can reduce a vector $\boldsymbol{t}$ with a vector $\boldsymbol{v} \in L$ with probability $p = [1 - \frac{\alpha^2}{4\beta^2}]^{d/2+o(d)}$. Since we have $n = \alpha^{d+o(d)}$ such vectors $\boldsymbol{v} \in L$, the probability that none of them can reduce $\boldsymbol{t}$ is $(1-p)^n$, which is $o(1)$ if $n \gg 1/p$ and $1 - o(1)$ if $n \ll 1/p$. Expanding $n \cdot p$, we obtain the given equation (30), where $\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 > 0$ implies $n \ll 1/p$.

For the second part, consider the triangle formed by $\mathbf{0}, \boldsymbol{t}, \boldsymbol{v}$. If $\|\boldsymbol{t} - \boldsymbol{v}\| = \|\boldsymbol{t}\|$, then this triangle has two sides β and one side α, and two angles $\theta_{\boldsymbol{t},\boldsymbol{v}}$ and one angle $\pi - 2\theta_{\boldsymbol{t},\boldsymbol{v}}$. By the sine law, $\alpha \sin\theta_{\boldsymbol{t},\boldsymbol{v}} = \beta \sin(\pi - 2\theta_{\boldsymbol{t},\boldsymbol{v}}) = 2\beta \sin\theta_{\boldsymbol{t},\boldsymbol{v}} \cos\theta_{\boldsymbol{t},\boldsymbol{v}}$. Simplifying, we get $\cos\theta_{\boldsymbol{t},\boldsymbol{v}} = \alpha/(2\beta)$, or $\sin^2\theta_{\boldsymbol{t},\boldsymbol{v}} = 1 - \alpha^2/(4\beta^2)$. Multiplying by α^2 yields

$$\alpha^2 \sin^2\theta_{\boldsymbol{t},\boldsymbol{v}} = \frac{4\beta^2\alpha^2 - \alpha^4}{4\beta^2} = \frac{4\beta^2}{4\beta^2} = 1, \tag{31}$$

where the next-to-last equality follows from the assumption that $\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 = 0$. Therefore $\|\boldsymbol{t} - \boldsymbol{v}\| = \|\boldsymbol{t}\|$ corresponds to $\sin\theta_{\boldsymbol{t},\boldsymbol{v}} = 1/\alpha$, or equivalently $\theta_{\boldsymbol{t},\boldsymbol{v}} = \arcsin(1/\alpha)$. If $\|\boldsymbol{t} - \boldsymbol{v}\| < \|\boldsymbol{t}\|$, then the angle $\theta_{\boldsymbol{t},\boldsymbol{v}}$ further decreases, leading to $\theta_{\boldsymbol{t},\boldsymbol{v}} \leq \arcsin(1/\alpha)$. □

As a result of the second part of the previous lemma, reducing $\boldsymbol{t}$ with L can be done by searching for vectors $\boldsymbol{v} \in L$ at angle at most $\theta_{\boldsymbol{t},\boldsymbol{v}} = \arcsin(1/\alpha)$ from $\boldsymbol{t}$: if $\boldsymbol{t}$ can be reduced with some vector $\boldsymbol{v} \in L$, then with high probability a vector $\boldsymbol{v}$ at this angle from $\boldsymbol{t}$ exists which can reduce $\boldsymbol{t}$. This will be necessary for applying Lemma 2 later, as that lemma assumes that the list size $n = |L|$ and the target angle θ satisfy the relation $n = (1/\sin\theta)^{d+o(d)}$. In our setting $n = \alpha^{d+o(d)}$ and $\theta = \arcsin(1/\alpha)$, which means this relation is satisfied.

Note that we do not just assume that L contains $\alpha^{d+o(d)}$ lattice vectors of norm approximately $\alpha \cdot \lambda_1(\mathcal{L})$: we heuristically expect the preprocessed list to contain (almost) all shortest vectors in $\mathcal{L}$, including all those vectors even shorter than $\alpha \cdot \lambda_1(\mathcal{L})$. In other words, for any $\alpha_0 \in [1, \alpha]$ we expect L to contain $\alpha_0^{d+o(d)}$ lattice vectors of norm at most $\alpha_0 \cdot \lambda_1(\mathcal{L})$. To be able to make progress and obtain a reduced vector $\boldsymbol{t}'$ of norm $\beta \cdot \lambda_1(\mathcal{L})$, it therefore suffices that for *some* value $\alpha_0 \in [1, \alpha]$, we have $\alpha_0^4 - 4\beta^2\alpha_0^2 + 4\beta_0^2 \leq 0$. Factoring the left hand side of (30) in terms of its roots for α yields

$$p(\alpha) = \alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 \tag{32}$$

$$= \left(\alpha^2 - 2\beta\left(\beta - \sqrt{\beta^2 - 1}\right)\right)\left(\alpha^2 - 2\beta\left(\beta + \sqrt{\beta^2 - 1}\right)\right). \tag{33}$$

The polynomial $p(\alpha)$ has two positive roots $0 < r_1 < \sqrt{2} < r_2$, which both lie close to $\sqrt{2}$ if $\beta \approx 1$. The condition that $p(\alpha_0) \leq 0$ for some $\alpha_0 \leq \alpha$ is equivalent to the condition that α is at least equal to the smallest root:

$$\alpha \geq r_1 = \sqrt{2\beta\left(\beta - \sqrt{\beta^2 - 1}\right)}. \tag{34}$$

To obtain $\beta = 1 + o(1)$, i.e., to guarantee that our iterative slicer returns a vector $\boldsymbol{t}'' \in \mathcal{V}$, we need that $\alpha \geq \sqrt{2} + o(1)$, i.e., we must use at least $|L| = 2^{d/2+o(d)}$ preprocessed lattice vectors to guarantee that w.h.p. the algorithm succeeds, as previously shown in [52].

Probability that closest vector is in $|L|$. To analyze the success probability of the slicer more generally, note again that the previous analysis gives us a relation between the parameter α for the preprocessed list, and the norm $\|\boldsymbol{t}'\| = \beta \cdot \lambda_1(\mathcal{L})$ of the reduced vector returned by the slicer, before we will likely not make a lot of progress anymore. To guarantee that we succeed with high probability we need $\beta = 1 + o(1)$, but even if $\beta > 1$ this algorithm succeeds with a certain, small probability.

Heuristically, as mentioned before, we expect the closest vector to a target vector $\boldsymbol{t}$ to lie at distance approximately $\lambda_1(\mathcal{L})$ from $\boldsymbol{t}$. More specifically, one might assume that the closest vector follows a uniformly random distribution on a sphere of radius $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}$. After reduction, we therefore also expect the closest vector $\boldsymbol{s}$ to $\boldsymbol{t}'$ to lie uniformly at random in a ball of radius $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}'$. If the closest vector has norm at most $\alpha \cdot \lambda_1(\mathcal{L})$, it will actually be in our list L, and one might estimate the probability that this happens as the probability

that, if a vector is drawn at random from a ball of radius $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}'$, it has norm at most $\alpha \cdot \lambda_1(\mathcal{L})$.

To estimate this probability, we will use the following lemma regarding the volume of intersections of balls in high-dimensional spaces.

Lemma 11 (Intersections of balls). *[17, Lemma 2.4] Two balls $\mathcal{B}(\boldsymbol{v}_1, r_1)$ and $\mathcal{B}(\boldsymbol{v}_2, r_2)$ at distance $\|\boldsymbol{v}_1 - \boldsymbol{v}_2\| = D$ with centers $\boldsymbol{v}_1, \boldsymbol{v}_2$ and radii r_1, r_2, such that $\sqrt{|r_1^2 - r_2^2|} < D < r_1 + r_2$, satisfy*

$$\frac{|\mathcal{B}(\boldsymbol{v}_1, r_1) \cap \mathcal{B}(\boldsymbol{v}_2, r_2)|}{|\mathcal{B}(\mathbf{0}, 1)|} = \left(\frac{-D^4 + 2D^2\left(r_1^2 + r_2^2\right) - \left(r_1^2 - r_2^2\right)^2}{4D^2} \right)^{d/2+o(d)}. \quad (35)$$

To apply this lemma, and obtain a lower bound on the probability that the closest vector $\boldsymbol{s}$ to $\boldsymbol{t}'$ (which is assumed to lie uniformly at random on a sphere of radius $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}'$) has norm at most $\alpha \cdot \lambda_1(\mathcal{L})$ (so that with high probability $\boldsymbol{s} \in L$), we apply this lemma with the parameters instantiated as $(\boldsymbol{v}_1, \boldsymbol{v}_2, r_1, r_2, D) = (\boldsymbol{t}', \mathbf{0}, \lambda_1(\mathcal{L}), \alpha\lambda_1(\mathcal{L}), \beta\lambda_1(\mathcal{L}))$, where we make use of the previous lemma to conclude that with high probability, after reductions the vector $\boldsymbol{t}'$ lies at distance at most $\beta\lambda_1(\mathcal{L})$ from the origin. This heuristically tells us that the probability of success for the slicer, on input a list of the $\alpha^{d+o(d)}$ shortest vectors in the lattice, is at least proportional to:

$$p_\alpha \geq \frac{|\mathcal{B}(\mathbf{0}, \alpha \cdot \lambda_1(\mathcal{L})) \cap \mathcal{B}(\boldsymbol{t}', \lambda_1(\mathcal{L}))|}{|\mathcal{B}(\boldsymbol{t}', \lambda_1(\mathcal{L}))|} \quad (36)$$

$$= \left(\frac{-\beta^4 + 2\beta^2\left(\alpha^2 + 1\right) - \left(\alpha^2 - 1\right)^2}{4\beta^2} \right)^{d/2+o(d)}. \quad (37)$$

Substituting the aforementioned relation between α and β, which translates to $\beta^2 = \alpha^4/(4\alpha^2 - 4)$ when optimized for β, and expanding the polynomials in terms of α, we get the lower bound on p_α (or equivalently, the upper bound on $A_L = 1/p_\alpha$) given in Lemma 7.

Observe that the denominator of p_α is non-zero for arbitrary $\alpha \in (1, \sqrt{2})$, while the numerator has one root in this interval, at $\alpha \approx 1.03396$. For this value of α, we have $\beta = \alpha + 1$ and so the two balls around $\boldsymbol{t}'$ and $\mathbf{0}$ of radii $\{1, \alpha\} \cdot \lambda_1(\mathcal{L})$ are disjoint, resulting in $p_\alpha = 0$. For $\alpha = \sqrt{2}$ the expression between brackets evaluates to 1 as expected, while for $\alpha = \beta = \sqrt{4/3}$ (using the same list size as in sieving for SVP) we obtain $p_{\sqrt{4/3}} = (13/16)^{d/2+o(d)}$. So if we used a standard GaussSieve as preprocessing for CVPP, we would expect the success probability of a single reduction to be $(13/16)^{d/2+o(d)} \approx 2^{-0.150d+o(d)}$.

C Experimental results

To verify the heuristic assumptions, as well as to provide a preliminary assessment of the practicality of the proposed CVPP method, we tested the approximate Voronoi cell approach with randomized slicing for finding closest vectors

on the 50-dimensional lattice of the SVP challenge [1] with seed 0. All experiments were performed on a Medion Erazer P6661 laptop with an Intel Core i7-6500U processor (2.50GHz), with more than enough RAM for these experiments. Experiments typically consumed about 25% of the total CPU power, i.e., 50% of one of the two cores. The experiments were conducted by (1) generating a large set of short lattice vectors (by using the preprocessing algorithm described in Algorithm 6); (2) indexing the shortest of these in a nearest neighbor data structure for fast lookups; and (3) running the randomized slicer to find closest vectors for random target vectors.

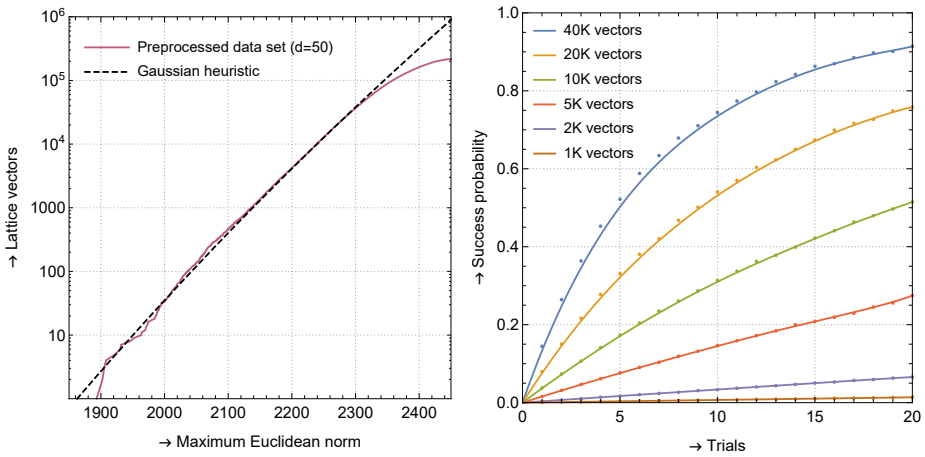
C.1 Nearest neighbor data structure

For nearest neighbor indexing, in our experiments we chose to use the hyperplane locality-sensitive hashing data structure [29] used in the HashSieve [50], rather than the spherical locality-sensitive filters [12, 18] used in the LDSieve [18] and which forms the basis of Lemma 2. There are two main reasons behind this choice.

First, for the HashSieve [50], the only parameters that need to be chosen are k (the number of hyperplanes) and t (the number of hash tables). As described in [50, 58], the asymptotically optimal parameter choices $k = 0.2206d$ and $t = 2^{0.1290d}$ for solving SVP with this method seem quite accurate in practice as near-optimal parameters for small/moderate dimensions as well. Being able to choose and fix the parameters easily means that fewer parameters need to be chosen for our experiments, and there will be a smaller variance in the results due to the changes of parameters between different experiments. This in contrast to the asymptotically superior spherical locality-sensitive filters [18, 51], for which several parameters must be chosen with less clear optimal choices in moderate dimensions.

Second, for solving SVP in dimension 50, a proof-of-concept HashSieve has previously been shown to outperform the LDSieve [18, Figure 3] by a factor more than 2. Using the LDSieve may ultimately lead to better time complexities in higher dimensions, with optimized code and an accurate analysis of how to actually choose the parameters in practice, but that lies beyond the scope of this section. The main targets here are to verify the heuristic assumptions, and to get an idea of the speedup compared to solving SVP or CVP directly.

For the experiments in dimension 50, we used the HashSieve with the same parameter choice of $k = 11$ hyperplanes and $t = 87$ hash tables as in [50]. We varied both the number of lattice vectors indexed in the data structure (out of all the vectors obtained from the preprocessing stage), and the number of rerandomizations before calling the search for a closest vector to this target a failure. We naturally sorted the preprocessed vectors by norm, so that using fewer vectors means that only the shortest of the lattice vectors found during the preprocessing phase are used. For randomizations, we sampled a random lattice vector from a discrete Gaussian over the lattice similar to how lattice vectors are sampled for the sieve, added it to the target vector, and reduced this new target instead.



(a) Norms of vectors in the data set (b) Rerandomization heuristics

Fig. 6. Verifying the heuristic assumptions. Figure 6a compares the norms of the vectors in the data set to the expected norms of the shortest vectors in the lattice, based on the Gaussian heuristic. Figure 6b depicts how rerandomizations affect the success probability. Different curves in Figure 6b correspond to a different number of preprocessed vectors being used for reductions.

C.2 The preprocessing step

First, let us describe the preprocessing step and the preprocessed data set in more detail. As described, we used Algorithm 6 to generate a list of short vectors in the lattice. Depending on how many list vectors we use after the preprocessing, the time needed to generate this list varies between a few seconds, for running the HashSieve and finding $2^{0.208d+o(d)}$ short lattice vectors, and several minutes or even hours to find tens of thousands of short lattice vectors in this lattice.

For the CVPP experiments, we ultimately generated an extremely large data set of short vectors in this lattice, consisting of about 250 000 lattice vectors of norm less than 3000, with the majority of them having norm less than 2500. Figure 6a shows the number of vectors in the data set below a certain norm, and compares it to the prediction for our preprocessed list size based on the Gaussian heuristic. Note that the Gaussian heuristic for this lattice predicts $\lambda_1(\mathcal{L}) \approx 1836.52\ldots$, and so we expect the list to contains approximately $\frac{1}{2}(N/1836)^{50}$ vectors in the lattice of norm at most N for $N \geq 1836$, where the factor $\frac{1}{2}$ comes from the observation that we always only store either $\boldsymbol{x}$ or $-\boldsymbol{x}$. If the Gaussian heuristic is accurate, then as we see in Figure 6a, the data set indeed contains almost all of the lattice vectors of norm less than 2300.

C.3 The rerandomization assumption

For analyzing the asymptotic performance of our randomized slicer, we assumed that rerandomizations (reducing the same target vector with the slicer multiple times, by considering random shifts $\boldsymbol{t}' \sim D_{\boldsymbol{t}+\mathcal{L},s}$) lead to independent successes and roughly a linear increase in the success probability as more trials are performed. Figure 6b plots the total success probability against the number of rerandomizations (trials), for various sizes of the list indexed in the data structure. Similar figures in the introduction show that indeed, it seems that if the success probability for a single trial is p, then the success probability for m trials is roughly proportional to $1 - (1 - p)^m$.

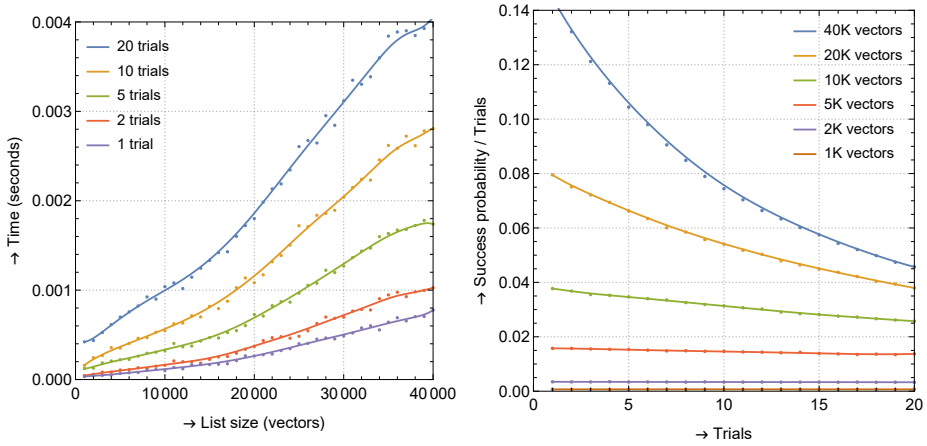
Note that for the experiments we set s such that the output vectors are not too long. As the Gaussian becomes more narrow, it becomes more and more likely that similar vectors are sampled, and results may not be "as fresh". In practice one can tweak the parameter s to obtain a proper trade-off between having as independent trials as possible, and not making the sampled vectors too long.

C.4 Experimental results

As stated before, for the CVPP experiments we fixed the nearest neighbor data structure parameters as $k = 11$ and $t = 87$. Focusing on the query costs, what remains is optimizing the size of the list L to use for the reductions, and assessing the precise practical impact of rerandomizations on the success probability and the time complexity. Note that by our heuristic assumption, the number of rerandomizations should not severely impact the normalized, expected time complexity, i.e., the time complexity divided by the success probability.

Figures 7 and 3 in the introduction show the results of these experiments, where we measured the average time complexity per target vector, the average success probability for each instance, and the normalized time complexity per instance. Different curves in these figures correspond to different numbers of rerandomizations/trials, and although this affects the success probability and the time complexity, in Figure 3 we see a confirmation that the normalized time complexity is essentially independent of the number of trials. Figure 3 further shows that the normalized time complexity seems to be smallest when the list contains between 10 000 and 15 000 lattice vectors, in which case the query time complexity T_2 for solving one CVPP instance is approximately 0.002 seconds. The memory complexity S_2 when using this number of list vectors is approximately 10MB.

Note also that Figure 3d shows that if we normalize the time complexity not only by the number of trials, but also by the number of vectors in the list, then the maximum success probability per trial, per vector, is also achieved at roughly the same number of vectors. Using fewer vectors would make the list too short and the success probability too small, while adding more vectors to the preprocessed list means adding longer vectors as well – the shortest vectors in the lattice are the most useful for reducing a target vector to the (approximate) Voronoi cell.



(a) Time complexity per CVP instance (b) Success probability increase per trial

Fig. 7. Experimental results for solving CVPP with approximate Voronoi cells in dimension 50. Figure 7a displays the average time complexity for running the slicer, not taking into account the resulting success probability. Figure 7b displays the normalized success probability per trial as a function of the number of trials, showing that indeed the success probability almost scales linearly with the number of trials. Each data point corresponds to 10 000 random target vectors for this choice of parameters. (See also Figure 3.)

C.5 Comparison with sieving for SVP and CVP

To put these results into perspective, let us compare the (normalized) time complexities for CVPP with the complexities of sieving for solving SVP or CVP[6] directly, i.e., without preprocessing.

First, we observe that with the same amount of optimization, the HashSieve algorithm solves SVP in approximately 4 seconds on the same machine. This means that in dimension 50, the expected time complexity for CVPP with our algorithm (about 2 milliseconds) is approximately 2000 times smaller than the time for solving SVP. To explain this gap, observe that the list size for solving SVP is approximately 4000, and so roughly speaking the HashSieve algorithm would need to perform 4000 reductions of newly sampled vectors with a list of maximum size 4000. For solving CVPP, we only need to reduce a single target vector with the list, but the list is now between 10 000 and 15 000 vectors long. This means that we save a factor 4000 on the number of searches through the list, but the searches are slightly more expensive as the list size is longer, leading to a speed-up of a factor slightly less than 4000.

[6] By results of [52], heuristically the complexities for SVP and CVP for sieving are equivalent up to at most a factor two.

To make these estimates more precise, note that the HashSieve for solving SVP [50] reported time complexities in dimension d of approximately $2^{0.45d-19}$ seconds, which corresponds to approximately 11.3 seconds in dimension 50, i.e., a factor 3 slower than our implementation. As explained above, this is based on doing $n = 2^{0.21d+o(d)}$ reductions. If for simplicity we assume that doing only one of these searches in a slightly larger list takes a factor $2^{0.21d}$ less time, and we take into account that for SVP the time complexity is now a factor 3 less than in [50], then we obtain an estimated time complexity in dimension d of $2^{0.24d-19}/3$, which for $d = 50$ corresponds to approximately 0.0026 seconds with our implementation, closely matching the observed time complexity. A rough extrapolation would then lead to a time complexity in dimension 100 of less than a minute.

C.6 Comparison with enumeration for SVP and CVP

In low dimensions, the fastest algorithms for solving SVP and CVP are based on enumeration. To compare our preprocessing approach with enumeration-based methods, we list several of the reported complexities for SVP and CVP with enumeration from the literature below, in chronological order. The listed time complexities are all for lattices in dimension 50.

- Agrell–Eriksson–Vardy–Zeger [4, Figure 2] give costs for CVP which, when extrapolated to dimension 50, would correspond to between 10 and 20 seconds.
- Nguyen–Vidick [68, Figure 4] report costs of Schnorr-Euchner enumeration with BKZ-20 preprocessing between 2 and 3 minutes.
- Hermans–Schneider–Buchmann–Vercauteren–Preneel [42, Table 2] give an estimate of between 5 and 7 seconds for enumeration with BKZ-20 preprocessing.
- Gama–Nguyen–Regev [40, Table 1] give four data points for the number of nodes processed during enumeration for three different versions of enumeration, which when fitted to the model 2^{ad^2+bd}, give $2^{0.00416d^2+0.255d}$ (full enumeration), $2^{0.00379d^2+0.115d}$ (Schnorr-Hörner pruning), and $2^{0.00387d^2+0.059d}$ (linear pruning). Taking into account their estimated rate of 10^7 nodes processed per second, in dimension $d = 50$ this leads to a sequential time complexity of approximately 0.94 seconds (full enumeration), 0.0038 seconds (Schnorr-Hörner pruning) and 0.00062 seconds (linear pruning). For extreme pruning, only two data points are provided, which is not enough to extrapolate to dimension 50.
- Dagdelen–Schneider [69, Table 1] report timings between 6 and 8 minutes for running their sequential implementation and for running fplll's enumeration with LLL preprocessing.
- Micciancio–Walter [66, Figure 7] give an experimental time complexity of Fincke-Pohst enumeration of approximately 30 seconds.
- Correia–Mariano–Proenca–Bischof–Agrell [32, Figure 6b] state a time complexity of enumeration for solving CVP in dimension 50 of approximately 10 seconds with BKZ-20 preprocessing.

Calling `shortest_vector()` within fplll 4.0 on the machine used for the experiments in this section (on a BKZ-20 reduced basis), the algorithm returns a shortest vector in approximately 30 seconds. In the most recent release of fplll (version 5) [34], this now takes approximately 0.7 seconds.

All reported experimental time complexities for enumeration in dimension 50 are significantly worse than our normalized 0.002 seconds per target vector. On the other hand, enumeration with linear pruning (and likely also extreme pruning) is still expected to solve more SVP (CVP) instances per second than our proof-of-concept CVPP implementation with hyperplane locality-sensitive hashing, and with extreme pruning the gap may be even bigger. On the other hand, in our implementation we did not make use of very recent improvements to sieving, which may also speed up the randomized slicing method, such as the POPCNT-trick described in [35], and faster heuristic nearest neighbor data structures used in [7]. Further work on high-speed implementations is needed to see how fast batches of CVP instances can really be solved in practice with approximate Voronoi cells.

D Asymptotics for variants of CVPP

Let us finally take a look at variants of CVPP, which sometimes make their appearance in the literature. The two main variants we will consider here are the approximation version of CVPP, and the preprocessing version of bounded distance decoding.

D.1 Solving BDDP$_\delta$

In bounded distance decoding, the target $\boldsymbol{t}$ lies unusually close to the lattice, i.e., closer than one might expect by the Gaussian heuristic. This problem naturally appears in lattice-based cryptography, when a private key consists of a *good basis* of a lattice with short basis vectors, and the public key is a *bad basis* of the same lattice. An encryption of a message could then consist of the message being mapped to a lattice point $\boldsymbol{s} \in \mathcal{L}$, and a small error vector $\boldsymbol{e}$ being added to $\boldsymbol{s}$ ($\boldsymbol{t} = \boldsymbol{s} + \boldsymbol{e}$) to hide $\boldsymbol{s}$. If the noise $\boldsymbol{e}$ is small enough, then with a good basis one can decode $\boldsymbol{t}$ to the closest lattice vector $\boldsymbol{s}$, while someone with the bad basis cannot decode correctly. As decoding for arbitrary $\boldsymbol{t}$ (solving CVP) is known to be hard even with knowledge of a good basis [8, 37, 62, 70], $\boldsymbol{e}$ needs to be very short for decryptions to work, and $\boldsymbol{t}$ must lie very close to the lattice. So instead of assuming that target vectors $\boldsymbol{t}$ are sampled at random, here we will assume that $\boldsymbol{t}$ lies at distance at most $\delta \cdot \lambda_1(\mathcal{L})$ from $\mathcal{L}$, for $\delta \in (0, 1)$.

For the approximate Voronoi cell approach for CVPP, let us again start by assuming that the preprocessed list L contains almost all $\alpha^{d+o(d)}$ lattice vectors of norm at most $\alpha \cdot \lambda_1(\mathcal{L})$. The choice of α implies a maximum norm $\beta \cdot \lambda_1(\mathcal{L})$ of the reduced vector $\boldsymbol{t}'$, as described in the analysis for exact CVPP. Here we now assume the nearest lattice vector $\boldsymbol{s}$ to $\boldsymbol{t}'$ lies within radius $\delta \cdot \lambda_1(\mathcal{L})$ of $\boldsymbol{t}'$, rather than within a radius $\lambda_1(\mathcal{L})$, and so we find the (heuristic) probability of

finding the closest lattice vector among the list, after reducing the target vector $\boldsymbol{t}$ to a vector $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$ of norm $\|\boldsymbol{t}'\| \leq \beta \cdot \lambda_1(\mathcal{L})$ to be:

$$p_\alpha^{(\delta)} = \frac{|\mathcal{B}(\boldsymbol{t}', \delta) \cap \mathcal{B}(\boldsymbol{0}, \alpha)|}{|\mathcal{B}(\boldsymbol{t}', \delta)|} = \left(\frac{-\beta^4 + 2\beta^2 \left(\alpha^2 + \delta^2\right) - \left(\alpha^2 - \delta^2\right)^2}{4\beta^2\delta^2} \right)^{d/2+o(d)}. \tag{38}$$

Without rerandomizations, to achieve $p \approx 1$, we need $\sqrt{\beta_\alpha^2 + \delta^2} \leq \alpha$ to expect the nearest lattice vector to $\boldsymbol{t}'$ to be contained in L, so that ultimately $\boldsymbol{0}$ is nearest to $\boldsymbol{t}'$ after reductions. Substituting $\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 = 0$ and $\beta^2 + \delta^2 \leq \alpha^2$, and solving for $\alpha > 1$, without rerandomizing this leads to the condition $\alpha^2 \geq \frac{2}{3}(1+\delta^2) + \frac{2}{3}\sqrt{(1+\delta^2)^2 - 3\delta^2}$. Taking $\delta = 1$, corresponding to exact CVPP, leads to the condition $\alpha \geq \sqrt{2}$ as expected, while in the limiting case of $\delta \to 0$ we obtain the condition $\alpha \geq \sqrt{4/3}$. This matches experimental observations using the GaussSieve, where after finding the shortest vector, newly sampled lattice vectors often cause *collisions* (i.e., reductions to the $\boldsymbol{0}$-vector). In other words, Algorithm 1 quite often reduces target vectors $\boldsymbol{t}$ which essentially lie *on* the lattice ($\delta \to 0$) to the $\boldsymbol{0}$-vector when the list has size $(4/3)^{d/2+o(d)}$. This explains why collisions in the GaussSieve are common when the list size grows to size $(4/3)^{d/2+o(d)}$.

With rerandomizations, a choice of α implies a norm β of the reduced vector $\boldsymbol{t}'$, and a probability $p_\alpha^{(\delta)}$ that the closest lattice vector is then found with the algorithm. For each α we can further use nearest neighbor searching with varying parameters u as in Lemma 2, and we can vary $\alpha \in (\alpha_0, \alpha_1)$ where α_0, α_1 follow from the equations $p_{\alpha_0,\delta} = 0$ and $p_{\alpha_1,\delta} = 1$ respectively. In other words, α_1 satisfies $\beta_{\alpha_1}^2 + \delta^2 = \alpha_1^2$, and α_0 is a root of the denominator of (38). This ultimately leads to the following theorem.

Theorem 4 (Optimized BDDP$_\delta$ complexities). *Let $p_\alpha^{(\delta)}$ be as in* (38)*. Let $\alpha_0^{(\delta)}, \alpha_1^{(\delta)}$ be the smallest values α larger than 1 with $p_\alpha^{(\delta)} = 0$ and $p_\alpha^{(\delta)} = 1$ respectively. Let $\alpha \in (\alpha_0^{(\delta)}, \alpha_1^{(\delta)})$ and $u \in (\sqrt{\frac{\alpha^2-1}{\alpha^2}}, \sqrt{\frac{\alpha^2}{\alpha^2-1}})$. With approximate Voronoi cells we can heuristically solve BDDP$_\delta$ with preprocessing space and time* S_1 *and* T_1*, and query space and time* S_2 *and* T_2*, where:*

$$\mathrm{S}_1 = \max\left\{ \mathrm{S}_2, \left(\frac{4}{3}\right)^{d/2+o(d)} \right\}, \qquad \mathrm{T}_1 = \max\left\{ \mathrm{S}_2, \left(\frac{3}{2}\right)^{d/2+o(d)} \right\}, \tag{39}$$

$$\mathrm{S}_2 = \left(\frac{\alpha}{\alpha - (\alpha^2-1)(\alpha u^2 - 2u\sqrt{\alpha^2-1} + \alpha)} \right)^{d/2+o(d)}, \tag{40}$$

$$\mathrm{T}_2^{(\delta)} = \left(\frac{16\alpha^4\left(\alpha^2-1\right)\delta^2}{-9\alpha^8+8\alpha^6(3+5\delta^2)-8\alpha^4(2+9\delta^2+2\delta^4)+32\alpha^2(\delta^2+\delta^4)-16\delta^4} \right)^{d/2+o(d)} \tag{41}$$

$$\cdot \left(\frac{\alpha + u\sqrt{\alpha^2-1}}{-\alpha^3 + \alpha^2 u\sqrt{\alpha^2-1} + 2\alpha} \right)^{d/2+o(d)}. \tag{42}$$

For arbitrary δ, we can do a search over all values of α and u to obtain the best time–space trade-off. For various δ the resulting trade-offs are depicted in Figure 4a. Note that in the limit $\delta \to 0$, we have $\alpha_0, \alpha_1 \to \sqrt{4/3}$. In other words, we then always have $p \to 0$ or $p \to 1$ as $\delta \to 0$, which means that rerandomizations do not help; either the algorithm almost always succeeds, or it always fails.

To further illustrate the behavior of the limiting case $\delta \to 0$ (with $\alpha \to \sqrt{4/3}$), note:

- For $u = \frac{1}{2}$, we have $\mathrm{S} \approx 2^{0.2075d+o(d)}$ and $\mathrm{T}_2 = (5/4)^{d/2+o(d)} \approx 2^{0.1610d+o(d)}$.
- For $u = 1$, we have $\mathrm{S} \approx 2^{0.2925d+o(d)}$ and $\mathrm{T}_2 = (9/8)^{d/2+o(d)} \approx 2^{0.0850d+o(d)}$.

In the limit of large $u \to \sqrt{\frac{\alpha^2}{\alpha^2-1}}$ we again obtain $\mathrm{S}_{1,2}, \mathrm{T}_1 \to 2^{\omega(d)}$ and $\mathrm{T}_2 \to 2^{o(d)}$ (regardless of δ) similar to solving CVPP without a distance guarantee.

D.2 Solving CVPP$_\kappa$

Besides BDD, where $\boldsymbol{t}$ lies unusually close to the lattice, another easier variant of CVP is the Approximate Closest Vector Problem. Given a lattice $\mathcal{L}$ and a target vector $\boldsymbol{t} \in \mathbb{R}^d$, the approximate closest vector problem with approximation factor κ is to find a vector $\boldsymbol{s} \in \mathcal{L}$ such that $\|\boldsymbol{s} - \boldsymbol{t}\|$ is at most a factor κ larger than the real distance from $\boldsymbol{t}$ to $\mathcal{L}$. For random instances $\boldsymbol{t}$, by the Gaussian heuristic this means a lattice vector $\boldsymbol{s}$ counts as a solution for approximate CVP with approximation factor κ iff $\boldsymbol{s}$ lies at distance at most $\kappa \cdot \lambda_1(\mathcal{L})$ from $\boldsymbol{t}$.

For the approach from Appendices B–C, we may hope to further improve upon the query complexities (after the preprocessing phase), similar to BDD. Without rerandomizations, instead of reducing $\boldsymbol{t}$ to a vector $\boldsymbol{t}'$ of norm at most $\lambda_1(\mathcal{L})$, as is needed for solving exact CVP ($\beta = 1$), we now update the analysis to take into account that we want to make sure that the reduced vector $\boldsymbol{t}'$ has norm at most $\kappa \cdot \lambda_1(\mathcal{L})$ ($\beta = \kappa$). If this is the case, then the vector $\boldsymbol{t} - \boldsymbol{t}'$ is a lattice vector lying at distance at most $\kappa \cdot \lambda_1(\mathcal{L})$ from $\boldsymbol{t}$, which w.h.p. qualifies as a solution. This means that instead of substituting $\beta = 1$ in Lemma 10 for exact CVPP (without rerandomizations), we now substitute $\beta = \kappa$, leading to the condition that $\alpha^4 - 4\kappa^2\alpha^2 + 4\beta^2 \leq 0$. By a similar analysis α must therefore be larger than the smallest root $r_1 = \sqrt{2\kappa(\kappa - \sqrt{\kappa^2 - 1})}$ of this quartic polynomial. A sanity check shows that $\kappa = 1$, corresponding to exact CVP, indeed results in $\alpha \geq \sqrt{2}$, while in the limit of $\kappa \to \infty$ a value $\alpha \approx 1$ suffices to obtain a vector $\boldsymbol{t}'$ of norm at most $\kappa \cdot \lambda_1(\mathcal{L})$. In other words, to solve approximate CVP with very large (constant) approximation factors, a preprocessed list of size $(1+\varepsilon)^{d+o(d)}$ suffices. Further note that $\kappa = \sqrt{4/3}$ leads to the same value $\alpha = \sqrt{4/3}$ as in BDD with $\delta \to 0$.

With rerandomizations, the analysis can be updated as follows. Instead of asking that the single closest vector $\boldsymbol{s}$ at distance $\lambda_1(\mathcal{L})$ from $\boldsymbol{t}'$ is contained in the list L (and has norm at most $\alpha \cdot \lambda_1(\mathcal{L})$), we now want that at least one of the $\kappa^{d+o(d)}$ lattice vectors $\boldsymbol{s}$ at distance at most $\kappa \cdot \lambda_1(\mathcal{L})$ from $\boldsymbol{t}'$ has norm at

most $\alpha \cdot \lambda_1(\mathcal{L})$. This leads to the following alternative definition for the success probability:

$$p_\alpha^{(\kappa)} = \kappa^d \cdot \frac{|\mathcal{B}(\boldsymbol{t}', \kappa) \cap \mathcal{B}(\mathbf{0}, \alpha)|}{|\mathcal{B}(\boldsymbol{t}', \kappa)|} = \left(\frac{-\beta^4 + 2\beta^2\left(\alpha^2+\kappa^2\right) - \left(\alpha^2-\kappa^2\right)^2}{4\beta^2} \right)^{d/2+o(d)}. \quad (43)$$

The conditions on the parameter α are analogous to BDD: we require that the asymptotic formulas for p lie in the range $[0, 1]$. More precisely, if this asymptotic expression exceeds 1, then the conditions of Lemma 11 are not met, and we instead have $p = 1 - o(1)$. As increasing α beyond the smallest value for which $p \approx 1$ only leads to worse complexities, we can simply assume that α is chosen such that for these asymptotic expressions, $p \leq 1$. Substituting the above expressions for p, with rerandomizations we now obtain the following result.

Theorem 5 (Optimized CVPP$_\kappa$ complexities). *Let $p_\alpha^{(\kappa)}$ be as in (43). Let $\alpha_0^{(\kappa)}, \alpha_1^{(\kappa)}$ be the smallest values α larger than 1 with $p_\alpha^{(\kappa)} = 0$ and $p_\alpha^{(\kappa)} = 1$ respectively. Let $\alpha \in (\alpha_0^{(\kappa)}, \alpha_1^{(\kappa)})$ and $u \in (\sqrt{\frac{\alpha^2-1}{\alpha^2}}, \sqrt{\frac{\alpha^2}{\alpha^2-1}})$. With approximate Voronoi cells we can heuristically solve CVPP$_\kappa$ with preprocessing space and time S_1 and T_1, and query space and time S_2 and T_2, where:*

$$\mathrm{S}_1 = \max\left\{ \mathrm{S}_2, \left(\frac{4}{3}\right)^{d/2+o(d)} \right\}, \qquad \mathrm{T}_1 = \max\left\{ \mathrm{S}_2, \left(\frac{3}{2}\right)^{d/2+o(d)} \right\}, \quad (44)$$

$$\mathrm{S}_2 = \left(\frac{\alpha}{\alpha - (\alpha^2-1)(\alpha u^2 - 2u\sqrt{\alpha^2-1} + \alpha)} \right)^{d/2+o(d)}, \quad (45)$$

$$\mathrm{T}_2^{(\kappa)} = \left(\frac{16\alpha^4\left(\alpha^2-1\right)}{-9\alpha^8 + 8\alpha^6(3+5\kappa^2) - 8\alpha^4(2+9\kappa^2+2\kappa^4) + 32\alpha^2(\kappa^2+\kappa^4) - 16\kappa^4} \right)^{d/2+o(d)} \quad (46)$$

$$\cdot \left(\frac{\alpha + u\sqrt{\alpha^2-1}}{-\alpha^3 + \alpha^2 u\sqrt{\alpha^2-1} + 2\alpha} \right)^{d/2+o(d)}. \quad (47)$$

Various optimized trade-offs for different values κ are depicted in Figure 4a, with the curve for $\kappa = \sqrt{4/3}$ partially overlapping with the BDD-curve for $\delta \to 0$. Recall that for 0-BDD, the interval for α only contains one value, resulting in one trade-off. For $\sqrt{4/3}$-CVP, this interval is not just one value, and choosing $\alpha < \sqrt{4/3}$ leads to better space complexities than for 0-BDD.

Also observe that from the bottom (thick, red) curve in Figure 4a, we can see that (after preprocessing the lattice) we can solve 2-CVPP in time and space both less than $2^{0.05d+o(d)}$, which follows from setting $\alpha \approx 1.035$ and $u \approx 0.381$, and only storing a small list of vectors L in memory. As κ increases, both α_0, α_1 tend to $1 + 1/(8\kappa^2) + O(\kappa^{-4})$, and for arbitrary superconstant κ we therefore obtain query time and space complexities both tending to $2^{o(d)}$.

Corollary 1 (Subexponential query complexities for CVPP$_\kappa$). *For arbitrary $\varepsilon > 0$, for sufficiently large κ we can use approximate Voronoi cells to*

heuristically solve approximate CVPP with approximation factor κ with preprocessing time $\mathrm{T}_1 = (3/2)^{d/2+o(d)}$, preprocessing space $\mathrm{S}_1 = (4/3)^{d/2+o(d)}$, and query time and space complexities $\mathrm{T}_2, \mathrm{S}_2 = 2^{\varepsilon d+o(d)}$. In particular, for $\kappa = \omega(1)$ we can solve CVPP$_\kappa$ in $2^{o(d)}$ time and space.

The corresponding algorithm is rather simple as well: (1) run a standard sieve for solving SVP; (2) discard all but the $2^{\varepsilon d+o(d)}$ shortest vectors found by the algorithm; and (3) use Algorithm 1 to find a sufficiently close lattice vector to $\boldsymbol{t}$. To obtain slightly subexponential query complexities one does not even need rerandomizations or nearest neighbor searching; these subexponential costs follow directly from $\kappa = \omega(1)$.

To compare Corollary 1 with previous work, note that α_0, α_1 both tend to $1 + 1/(8\kappa^2) + O(\kappa^{-4})$ as κ grows. The query space and time complexities are both proportional to $\alpha^{\Theta(d)}$. To obtain polynomial query complexities, we can solve for κ, leading to the following result.

Corollary 2 (Polynomial query complexities for CVPP$_\kappa$). *Using approximate Voronoi cells we can heuristically solve CVPP$_\kappa$ in polynomial query time and space iff $\kappa = \Omega(\sqrt{d/\log d})$.*

Proof. The query time and space complexities are given by $\alpha^{\Theta(d)}$, where $\alpha = 1+\Theta(1/\kappa^2)$. To obtain polynomial complexities in d, we must have $\alpha^{\Theta(d)} = d^{O(1)}$, or equivalently:

$$1 + \Theta\left(\frac{1}{\kappa^2}\right) = \alpha = d^{O(1/d)} = \exp O\left(\frac{\log d}{d}\right) = 1 + O\left(\frac{\log d}{d}\right). \tag{48}$$

Solving for κ leads to the given relation between κ and d.

This is equivalent (minus the heuristic assumptions) to a result of Aharonov and Regev [5], who previously showed that the decision version of CVPP with approximation factor $\kappa = \Omega(\sqrt{d/\log d})$ can provably be solved in polynomial time. This also heuristically improves upon results of [33,56], who showed how to solve the search-version of CVPP with polynomial time and space complexities for $\kappa = O(d^{3/2})$ and $\kappa = \Omega(d/\sqrt{\log d})$ respectively. These comparisons suggest that this sieving-based approximate Voronoi cell method may well be optimal from a theoretical point of view as well.