Efficient (ideal) lattice sieving using cross-polytope LSH

Anja Becker[1] and Thijs Laarhoven[2*]

[1] EPFL, Lausanne, Switzerland — `anja.becker@epfl.ch`
[2] TU/e, Eindhoven, The Netherlands — `mail@thijs.com`

Abstract. Combining the efficient cross-polytope locality-sensitive hash family of Terasawa and Tanaka with the heuristic lattice sieve algorithm of Micciancio and Voulgaris, we show how to obtain heuristic and practical speedups for solving the shortest vector problem (SVP) on both arbitrary and ideal lattices. In both cases, the asymptotic time complexity for solving SVP in dimension n is $2^{0.298n+o(n)}$.
For any lattice, hashes can be computed in polynomial time, which makes our CPSieve algorithm much more practical than the SphereSieve of Laarhoven and De Weger, while the better asymptotic complexities imply that this algorithm will outperform the GaussSieve of Micciancio and Voulgaris and the HashSieve of Laarhoven in moderate dimensions as well. We performed tests to show this improvement in practice.
For ideal lattices, by observing that the hash of a shifted vector is a shift of the hash value of the original vector and constructing rerandomization matrices which preserve this property, we obtain not only a linear decrease in the space complexity, but also a linear speedup of the overall algorithm. We demonstrate the practicability of our cross-polytope ideal lattice sieve IdealCPSieve by applying the algorithm to cyclotomic ideal lattices from the ideal SVP challenge and to lattices which appear in the cryptanalysis of NTRU.

Keywords: (ideal) lattices, shortest vector problem, sieving algorithms, locality-sensitive hashing

1 Introduction

Lattice-based cryptography. Lattices are discrete additive subgroups of $\mathbb{R}^n$. More concretely, given a basis $B = \{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\} \subset \mathbb{R}^n$, the lattice generated by B, denoted by $\mathcal{L} = \mathcal{L}(B)$, is defined as the set of all integer linear combinations of the basis vectors: $\mathcal{L} = \{\sum_{i=1}^n \mu_i \boldsymbol{b}_i : \mu_i \in \mathbb{Z}\}$. The security of lattice-based cryptography relies on the hardness of certain hard lattice problems, such as the shortest vector problem (SVP): given a basis B of a lattice, find a shortest non-zero vector $\boldsymbol{v} \in \mathcal{L}$, where shortest is defined in terms of the Euclidean norm. The length of a shortest non-zero vector is denoted by $\lambda_1(\mathcal{L})$. A common relaxation of SVP is the approximate shortest vector problem (SVP$_\delta$): given a basis B of

[*] Part of this work was done while the second author was visiting EPFL.

$\mathcal{L}$ and an approximation factor $\delta > 1$, find a non-zero vector $\boldsymbol{v} \in \mathcal{L}$ whose norm does not exceed $\delta \cdot \lambda_1(\mathcal{L})$.

Although SVP and SVP_δ with constant approximation factor δ are well-known to be NP-hard under randomized reductions [4, 29], choosing parameters in lattice cryptography remains a challenge [18, 36, 50] as e.g. (i) the actual computational complexity of SVP and SVP_δ is still not very well understood; and (ii) for efficiency, lattice-based cryptographic primitives such as NTRU [24] commonly use special, structured lattices, for which solving SVP and SVP_δ may potentially be much easier than for arbitrary lattices.

SVP algorithms. To improve our understanding of these hard lattice problems, which may ultimately help us strengthen (or lose) our faith in lattice cryptography, the only solution seems to be to analyze algorithms that solve these problems. Studies of algorithms for solving SVP already started in the 1980s [16, 28, 49] when it was shown that a technique called enumeration can solve SVP in superexponential time ($2^{\Omega(n \log n)}$) and polynomial space. In 2001 Ajtai et al. showed that SVP can actually be solved in single exponential time ($2^{\Theta(n)}$) with a technique called sieving [5], which requires a single exponential space complexity as well. Even more recently, two new methods were invented for solving SVP based on using Voronoi cells [42] and on using discrete Gaussian sampling [2]. These methods also require a single exponential time and space complexity.

Sieving algorithms. Out of the latter three methods with a single exponential time complexity, sieving still seems to be the most practical to date; the provable time exponent for sieving may be as high as $2^{2.465n+o(n)}$ [23, 46, 51] (compared to $2^{2n+o(n)}$ for the Voronoi cell algorithm, and $2^{n+o(n)}$ for the discrete Gaussian combiner), but various heuristic improvements to sieving since 2001 [10, 43, 46, 61, 62] have shown that in practice sieving may be able to solve SVP in time and space as little as $2^{0.378n+o(n)}$. Other works on sieving have further shown how to parallelize and speed up sieving in practice with various polynomial speedups [12, 17, 27, 39–41, 45, 53, 54], and how sieving can be made even faster on certain structured, ideal lattices used in lattice cryptography [12, 27, 54]. Ultimately both Ishiguro et al. [27] and Bos et al. [12] managed to solve an 128-dimensional ideal SVP challenge [48] using a modified version of the GaussSieve [43], which is currently still the highest dimension in which a challenge from the ideal lattice challenge was successfully solved.

Sieving and locality-sensitive hashing. Even more recently, a new line of research was initiated which combines the ideas of sieving with a technique from the literature of nearest neighbor searching, called locality-sensitive hashing (LSH) [26]. This led to a practical algorithm with heuristic time and space complexities of only $2^{0.337n+o(n)}$ (the HashSieve [32, 41]), and an algorithm with even better asymptotic complexities of only $2^{0.298n+o(n)}$ (the SphereSieve [33]). However, for both methods the polynomial speedups that apply to the GaussSieve for ideal lattices [12, 27, 54] do not seem to apply, and the latter algorithm may be of limited practical interest due to large hidden order terms in the LSH technique and the fact that this technique seems incompatible with the GaussSieve [43] and only works with the less practical NV-sieve [46]. Understanding the possibilities and limitations of sieving with LSH, as well as finding new ways to efficiently apply similar techniques to ideal lattices remains an open problem.

Our contributions. In this work we show how to obtain practical, exponential speedups for sieving (in particular for the GaussSieve algorithm [12,27,43]) using the cross-polytope LSH technique first introduced by Terasawa and Tanaka in 2007 [60] and very recently further analyzed by Andoni et al. [9]. Our results are two-fold:

Arbitrary lattices. For arbitrary lattices, using polytope LSH leads to a practical sieve with heuristic time and space complexities of $2^{0.298n+o(n)}$. The exact trade-off between the time and memory is shown in Figure 1. The low polynomial cost of computing hashes and the fact that this algorithm is based on the GaussSieve (rather than the NV-sieve [46]) indicate that this algorithm is more practical than the SphereSieve [33], while in moderate dimensions this method will be faster than both the GaussSieve and the HashSieve due to its better asymptotic time complexity.

Ideal lattices. For ideal lattices commonly used in cryptography, we show how to obtain similar polynomial speedups and decreases in the space complexity as in the GaussSieve [12, 27, 54]. In particular, both the time and space for solving SVP decrease by a factor $\Theta(n)$, and the cost of computing hashes decreases by a quasi-linear factor $\Theta(n/\log n)$ using Fast Fourier Transforms.

These results emphasize the potential of sieving for solving high-dimensional instances of SVP, which in turn can be used inside lattice basis reduction algorithms like BKZ [56,57] to find short (rather than shortest) vectors in even higher dimensions. As a consequence, these results will be an important guide for estimating the long-term security of lattice-based cryptography, and in particular for selecting parameters in lattice-based cryptographic primitives.

Outline. The paper is organized as follows. In Section 2 we recall some background on lattices, sieving, locality-sensitive hashing, and the polytope LSH family of Terasawa and Tanaka [60]. Section 3 describes how to combine these techniques to solve SVP on arbitrary lattices, and how this leads to an asymptotic time (and space) complexity of $2^{0.298n+o(n)}$. Section 4 describes how to make the resulting algorithm even faster for lattices with a specific ideal structure, such as some of the lattices of the ideal lattice challenge [48] and lattices appearing in the cryptanalysis of NTRU [24]. Finally, Section 5 concludes with a brief discussion of the results.

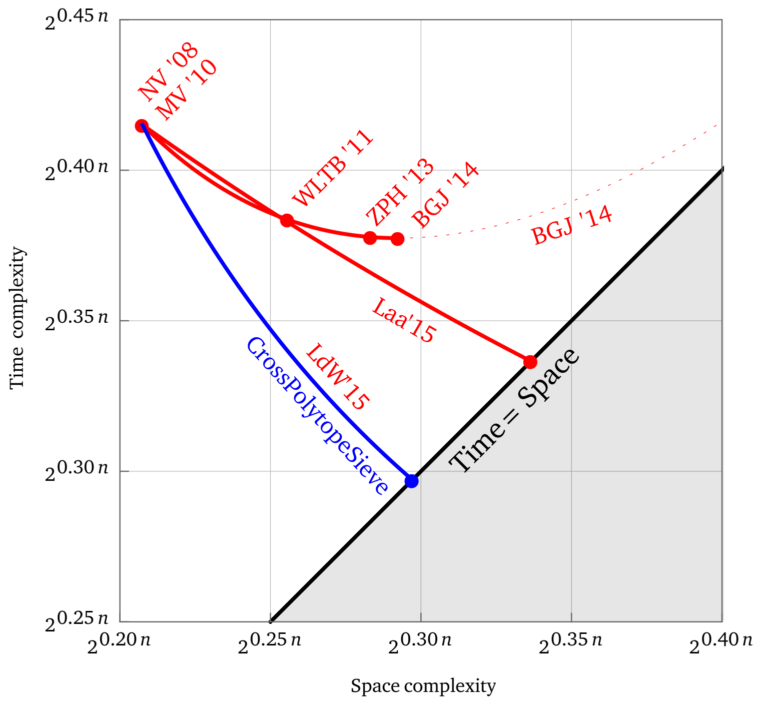

Fig. 1. The heuristic space-time trade-off of various previous heuristic sieving algorithms from the literature (the red points and curves), and the heuristic trade-off between the space and time complexities obtained with our algorithm (the blue curve). The referenced papers are: NV'08 [46] (the NV-sieve), MV'10 [43] (the GaussSieve), WLTB'11 [61] (two-level sieving), ZPH'13 [62] (three-level sieving), BGJ'14 [10] (the decomposition approach), Laa'15 [32] (the HashSieve), LdW'15 [33] (the SphereSieve). Note that the trade-off curve for the CPSieve (the blue curve) overlaps with the asymptotic trade-off of the SphereSieve of [33].

2 Preliminaries

2.1 Lattices

Let us first recall some basics on lattices. As mentioned in the introduction, we let $\mathcal{L} = \mathcal{L}(B)$ denote the lattice generated by the basis $B = \{\boldsymbol{b}_1, \dots, \boldsymbol{b}_n\} \subset \mathbb{R}^n$, and the shortest vector problem asks to find a vector of length $\lambda_1(\mathcal{L})$, i.e. a shortest non-zero vector in the lattice. Lattices are additive groups, and so if $\boldsymbol{v}, \boldsymbol{w} \in \mathcal{L}$, then also $\lambda_v \boldsymbol{v} + \lambda_w \boldsymbol{w} \in \mathcal{L}$ for $\lambda_v, \lambda_w \in \mathbb{Z}$.

Within the set of all lattices there is a subset of ideal lattices, which are defined in terms of ideals of polynomial rings. Given a ring $R = \mathbb{Z}[X]/(g)$ where $g \in \mathbb{Z}[X]$ is a degree-n monic polynomial, we can represent a polynomial $v(X) = \sum_{i=1}^n v_i X^{i-1}$ in this ring by a vector $\boldsymbol{v} = (v_1, \dots, v_n)$. Then, given a set of generators $b_1, \dots, b_k \in R$, we define the ideal $I = \langle b_1, \dots, b_k \rangle$ by the properties

(i) if $a, b \in I$ then also $\lambda a + \mu b \in I$ for scalars $\lambda, \mu \in \mathbb{Z}$; and (ii) if $a \in R$ and $b \in I$ then $a \cdot b \in I$. Note that when these polynomials are translated to vectors, the first property corresponds exactly to the property of a lattice, while the second property makes this an ideal lattice. In terms of lattices, the second property can equivalently be written as:

$$(v_1, \dots, v_n) \in \mathcal{L} \;\Leftrightarrow\; (w_1, \dots, w_n) \in \mathcal{L}, \text{ where } w \equiv X \cdot v \bmod g \text{ in } R. \tag{1}$$

In this paper we will restrict our attention to a few specific choices of g as follows:

Cyclic lattices: If $g(X) = X^n - 1$ and $\boldsymbol{v} = (v_1, \dots, v_n)$, then $w \equiv X \cdot v$ implies that $\boldsymbol{w} = (v_n, v_1, \dots, v_{n-1})$, i.e. multiplying a polynomial in the ring by X corresponds to a right-shift (with carry) of the corresponding vector, and so any cyclic shift of a lattice vector is also in the lattice.

Negacyclic lattices: For the case $g(X) = X^n + 1$ we similarly have that multiplying a polynomial by X in the ring corresponds to a right-shift with carry, but in this case an extra minus sign appears with the carry: $w \equiv X \cdot v$ implies that $\boldsymbol{w} = (-v_n, v_1, \dots, v_{n-1})$.

Whereas the above descriptions of cyclic and negacyclic lattices are quite general, below we list two instances of these lattices that appear in practice which have certain additional properties.

NTRU lattices: Cyclic lattices most notably appear in the cryptanalysis of NTRU [24], where the polynomial ring is $R = \mathbb{Z}_q[x]/(X^p - 1)$ where p, q are prime. Due to the modular ring, the corresponding lattice is not quite cyclic but rather "block-cyclic". The NTRU lattice is formed by the $n = 2p$ basis vectors $\boldsymbol{b}_i = (q \cdot \boldsymbol{e}_i \| \boldsymbol{0})$ for $i = 1, \dots, p$ and $\boldsymbol{b}_{p+i} = (\boldsymbol{h}_i \| \boldsymbol{e}_i)$ for $i = 1, \dots, p$, where $\boldsymbol{e}_i$ corresponds to the ith unit vector, and $\boldsymbol{h}_i$ corresponds to the ith cyclic shift of the public key $\boldsymbol{h}$ generated from the private key $\boldsymbol{f}, \boldsymbol{g}$ (see [24] for details). In this case, if $\boldsymbol{v} = (\boldsymbol{v}_1 \| \boldsymbol{v}_2) \in \mathcal{L}$ is a lattice vector, then also shifting both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ to the right or left leads to a lattice vector. Finding a shortest non-zero vector in this lattice corresponds to finding the secret key $(\boldsymbol{f} \| \boldsymbol{g})$ and breaking the underlying cryptosystem.

Power-of-two cyclotomic lattices: Negacyclic lattices commonly appear in lattice cryptography, where $n = 2^k$ is a power of 2 so that, among others, g is irreducible. The 128-dimensional ideal lattice attacked by Ishiguro et al. [27] and Bos et al. [12] from the ideal lattice challenge [48] also belongs to this class of lattices. Lattices of this form previously appeared in the context of lattice cryptography in e.g. [20, 38, 59].

2.2 Sieving algorithms

For solving the shortest vector problem in single exponential time (rather than superexponential time, as with enumeration), in 2001 Ajtai et al. [5] proposed a new method called sieving. This method was later refined and modified leading to various different algorithms, the most practical of which seems to be the GaussSieve of Micciancio and Voulgaris [43]. Over the years this algorithm has received considerable attention in the literature [17, 27, 31, 39, 40, 45, 53, 54] and the highest SVP records achieved using sieving all used (a modification of) the GaussSieve, both for arbitrary lattices [31] and for ideal lattices [12, 27].

The GaussSieve. The GaussSieve algorithm, described in Algorithm 1, iteratively builds a longer and longer list of lattice vectors, and makes sure that the list remains pairwise Gauss-reduced throughout the execution of the algorithm. Here, two vectors $\boldsymbol{v}, \boldsymbol{w}$ are said to be Gauss-reduced with respect to each other iff $\|\boldsymbol{v} \pm \boldsymbol{w}\| \geq \|\boldsymbol{v}\|, \|\boldsymbol{w}\|$ where all norms are Euclidean norms. In other words, two vectors are (Gauss-)reduced if adding/subtracting one vector to/from the other does not lead to a shorter vector than the two vectors we started with. Note that a reduced pair of vectors always has an angle of at least $60°$ between them, as otherwise one vector could reduce the other, and the implication holds both ways if $\|\boldsymbol{v}\| = \|\boldsymbol{w}\|$; if one is longer than the other, then they might still be reducible (e.g. not yet reduced) even if their angle is more than $60°$.

If two vectors $\boldsymbol{v}, \boldsymbol{w}$ are not reduced, then we can either reduce $\boldsymbol{v}$ with $\boldsymbol{w}$ by replacing $\boldsymbol{v}$ with the shorter vector $\boldsymbol{v} \pm \boldsymbol{w}$, or reduce $\boldsymbol{w}$ with $\boldsymbol{v}$ by replacing it with $\boldsymbol{w} \pm \boldsymbol{v}$. For each pair of vectors such replacements are done if possible, and using sufficiently many vectors, we hope that collisions (vectors being reduced to the $\mathbf{0}$-vector after pairwise reductions) do not occur that often, so that the remaining number of vectors after building this list is so big that we eventually saturate all "corners" of the n-dimensional space with vectors in our list. In short, the algorithm keeps generating new vectors that it adds to the list, and it updates the list to keep it reduced, and adds list vectors that are no longer reduced with the list to a stack of vectors to be processed later. The vectors in this list will become shorter and shorter, and in the end we hope to find a shortest lattice vector in our list. This leads to the algorithm described in Algorithm 1.

Time and space complexities. For the complexity of this algorithm, the space complexity is heuristically bounded from below by $(4/3)^{n/2+o(n)} \approx 2^{0.2075n+o(n)}$ due to bounds on the kissing constant in high dimensions [14]; if we were to systematically encounter lattices for which the list size of the GaussSieve is larger than $2^{0.2075n+o(n)}$, then we would be able to systematically generate sets of points in high dimensions exceeding the long-standing lower bound on the kissing constant of $2^{0.2075n+o(n)}$, which is deemed unlikely. For the time complexity, the "collisions" that may occur by reducing vectors and ending up with the $\mathbf{0}$-vector have so far prevented anyone from proving heuristic bounds on the time complexity; theoretically, the algorithm may run forever without finding a shortest vector by repeatedly generating collisions. In practice this does not seem to be an issue at all, and commonly collisions only occur after a shortest vector is already in the list L. In practice the time complexity may well be estimated to be quadratic in the list size, i.e. $2^{0.4150n+o(n)}$, as each pair of points needs to be compared at least once. This matches high-dimensional experimental results of the GaussSieve [31] and the GaussSieve-based HashSieve [41].

Algorithm 1 The GaussSieve algorithm

```
 1: Initialize an empty list $L$ and an empty stack $S$
 2: repeat
 3:     Get a vector $\boldsymbol{v}$ from the stack $S$ (or sample a new one if $S = \emptyset$)
 4:     for each $\boldsymbol{w} \in L$ do
 5:         Reduce $\boldsymbol{v}$ with $\boldsymbol{w}$
 6:         Reduce $\boldsymbol{w}$ with $\boldsymbol{v}$
 7:         if $w$ has changed then
 8:             Remove $\boldsymbol{w}$ from the list $L$
 9:             Add $\boldsymbol{w}$ to the stack $S$ (unless $\boldsymbol{w} = \mathbf{0}$)
10:         end if
11:     end for
12:     if $\boldsymbol{v}$ has changed then
13:         Add $\boldsymbol{v}$ to the stack $S$ (unless $\boldsymbol{v} = \mathbf{0}$)
14:     else
15:         Add $\boldsymbol{v}$ to the list $L$
16:     end if
17: until $\boldsymbol{v}$ is a shortest vector of the lattice
```

Note that to actually prove (heuristically) that our proposed algorithm achieves a certain heuristic time and space complexity, one should apply the same techniques to the sieve algorithm of Nguyen and Vidick [46] as previously outlined in [32]. Nguyen and Vidick's algorithm comes with heuristic bounds on the time complexity (not based on conjectures on the kissing constant or on the conjectured absence of collisions), and the speedup we obtain applies to that algorithm in the same way. However, similar to [32] we are interested in designing the fastest and most practical sieving algorithm possible for solving SVP rather than the best provable heuristic algorithm, and so in the remainder of this paper we will focus on the GaussSieve. But one should keep in mind that for theoretical arguments these ideas may also be applied to the NV-sieve [46] which actually leads to provable bounds under suitable heuristic assumptions.

2.3 Locality-sensitive hashing

To distinguish between pairs of vectors which are nearby in space and pairs of vectors which are far apart, it is possible to use *locality-sensitive hash functions* first introduced in [26]. These are functions h which map an n-dimensional vector $\boldsymbol{v}$ to a low-dimensional *sketch* of $\boldsymbol{v}$, such that two vectors which are nearby in $\mathbb{R}^n$ have a higher probability of having the same sketch than two vectors which are far apart. A simple example of such a function is the function h mapping $\boldsymbol{v} = (v_1, \ldots, v_n)$ to $h(\boldsymbol{v}) = v_1$; two vectors which are nearby in n-dimensional space have a slightly higher probability of having similar first coordinates than vectors which are far apart. Formalizing this property leads to the following definition of a *locality-sensitive hash family* $\mathcal{H}$. Here, we assume D is a certain similarity measure[3], and the set U below may be thought of as (a subset of) the natural numbers $\mathbb{N}$.

Definition 1. *[26] A family $\mathcal{H} = \{h : \mathbb{R}^n \to U\}$ is called (r_1, r_2, p_1, p_2)-sensitive for a similarity measure D if for any $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ we have*

- *If $D(\boldsymbol{v}, \boldsymbol{w}) \leq r_1$ then $\mathbb{P}_{h \in \mathcal{H}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] \geq p_1$.*
- *If $D(\boldsymbol{v}, \boldsymbol{w}) \geq r_2$ then $\mathbb{P}_{h \in \mathcal{H}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] \leq p_2$.*

Note that if we are given a hash family $\mathcal{H}$ which is (r_1, r_2, p_1, p_2)-sensitive with $p_1 \gg p_2$, then we can use $\mathcal{H}$ to distinguish between vectors which are at most r_1 away from $\boldsymbol{v}$, and vectors which are at least r_2 away from $\boldsymbol{v}$ with non-negligible probability, by only looking at their hash values (and that of $\boldsymbol{v}$).

2.4 Amplification

In general it is unknown whether efficiently computable (r_1, r_2, p_1, p_2)-sensitive hash families even exist for the ideal setting of $r_1 \approx r_2$ (small gap) and $p_1 \approx 1$ and $p_2 \approx 0$ (strong distinguishing power). Instead, one commonly first constructs an (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$ with $p_1 \approx p_2$, and then uses several AND- and OR-compositions to turn it into an (r_1, r_2, p_1', p_2')-sensitive hash family $\mathcal{H}'$ with $p_1' > p_1$ and $p_2' < p_2$, thereby amplifying the gap between p_1 and p_2.

AND-composition Given an (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$, we can construct an (r_1, r_2, p_1^k, p_2^k)-sensitive hash family $\mathcal{H}'$ by taking k different, pairwise independent functions $h_1, \dots, h_k \in \mathcal{H}$ and a one-to-one mapping $f : U^k \to U$, and defining $h \in \mathcal{H}'$ as $h(\boldsymbol{v}) = f(h_1(\boldsymbol{v}), \dots, h_k(\boldsymbol{v}))$. Clearly $h(\boldsymbol{v}) = h(\boldsymbol{w})$ iff $h_i(\boldsymbol{v}) = h_i(\boldsymbol{w})$ for all $i \in [k]$, so if $\mathbb{P}[h_i(\boldsymbol{v}) = h_i(\boldsymbol{w})] = p$ for all i, then $\mathbb{P}[h(\boldsymbol{v}) = h(\boldsymbol{w})] = p^k$.

OR-composition Given an (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$, we can construct an $(r_1, r_2, 1-(1-p_1)^t, 1-(1-p_2)^t)$-sensitive hash family $\mathcal{H}'$ by taking t different, pairwise independent functions $h_1, \dots, h_t \in \mathcal{H}$, and defining $h \in \mathcal{H}'$ by the relation $h(\boldsymbol{v}) = h(\boldsymbol{w})$ iff $h_i(\boldsymbol{v}) = h_i(\boldsymbol{w})$ for *at least one* $i \in [t]$. Clearly $h(\boldsymbol{v}) \neq h(\boldsymbol{w})$ iff $h_i(\boldsymbol{v}) \neq h_i(\boldsymbol{w})$ for all $i \in [t]$, so if $\mathbb{P}[h_i(\boldsymbol{v}) \neq h_i(\boldsymbol{w})] = 1 - p$ for all i, then $\mathbb{P}[h(\boldsymbol{v}) \neq h(\boldsymbol{w})] = (1-p)^t$ for $j = 1, 2$.

Combining a k-wise AND-composition with a t-wise OR-composition, we can turn an (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$ into an $(r_1, r_2, 1-(1-p_1^k)^t, 1-(1-p_2^k)^t)$-sensitive hash family $\mathcal{H}'$. As long as $p_1 > p_2$, we can always find values k and t such that $p_1^* \overset{\text{def}}{=} 1 - (1-p_1^k)^t \approx 1$ and $p_2^* \overset{\text{def}}{=} 1 - (1-p_2^k)^t \approx 0$.

[3] A similarity measure D may informally be thought of as a "slightly relaxed" distance metric, which may not satisfy all properties associated to metrics.

2.5 Finding nearest neighbors with LSH

The near(est) neighbor problem is the following [26]: Given a long list L of n-dimensional vectors, i.e., $L = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_N\} \subset \mathbb{R}^n$, preprocess L in such a way that, when later given a target vector $\boldsymbol{v} \notin L$, one can efficiently find an element $\boldsymbol{w} \in L$ which is close(st) to $\boldsymbol{v}$. While in low (fixed) dimensions n there are ways to trivially answer these queries in time sub-linear or even logarithmic in the list size N, in high dimensions it seems hard to do better than with a naive brute-force list search of time $O(N)$. This inability to efficiently store and query lists of high-dimensional objects is sometimes referred to as the "curse of dimensionality" [26]. Fortunately, if we know that e.g. there is a significant gap between what is meant by "nearby" and "far away," then there are ways to preprocess L such that queries can be answered in time sub-linear in N, using locality-sensitive hash families.

To use these LSH families to find nearest neighbors, we can use the following method first described in [26]. First, we choose $t \cdot k$ random hash functions $h_{i,j} \in \mathcal{H}$, and we use the AND-composition to combine k of them at a time to build t different hash functions $h_1, \ldots, h_t$. Then, given the list L, we build t different hash tables $T_1, \ldots, T_t$, where for each hash table T_i we insert $\boldsymbol{w}$ into the bucket labeled $h_i(\boldsymbol{w})$. Finally, given the vector $\boldsymbol{v}$, we compute its t images $h_i(\boldsymbol{v})$, gather all the candidate vectors that collide with $\boldsymbol{v}$ in at least one of these hash tables (an OR-composition) in a list of candidates, and search this set of candidates for a nearest neighbor.

Clearly, the quality of this algorithm for finding nearest neighbors depends on the quality of the underlying hash family and on the parameters k and t. Larger values of k and t amplify the gap between the probabilities of finding 'good' (nearby) and 'bad' (faraway) vectors, which makes the list of candidates shorter, but larger parameters come at the cost of having to compute many hashes (during the preprocessing and querying phases) and having to store many hash tables in memory. The following lemma shows how to balance k and t such that the overall time complexity is minimized.

Lemma 1. *[26] Let $\mathcal{H}$ be an (r_1, r_2, p_1, p_2)-sensitive hash family. Then, for a list L of size N, taking*

$$\rho = \frac{\log(1/p_1)}{\log(1/p_2)}, \qquad k = \frac{\log(N)}{\log(1/p_2)}, \qquad t = O(N^\rho), \tag{2}$$

with high probability we can either (a) find an element $\boldsymbol{w}^* \in L$ with $D(\boldsymbol{v}, \boldsymbol{w}^*) \leq r_2$, or (b) conclude that with high probability, no elements $\boldsymbol{w} \in L$ with $D(\boldsymbol{v}, \boldsymbol{w}) > r_1$ exist, with the following costs:

1. *Time for preprocessing the list: $O(N^{1+\rho} \log_{1/p_2} N)$.*
2. *Space complexity of the preprocessed data: $O(N^{1+\rho})$.*
3. *Time for answering a query $\boldsymbol{v}$: $O(N^\rho)$.*
   - *Hash evaluations of the query vector $\boldsymbol{v}$: $O(N^\rho)$.*
   - *List vectors to compare to the query vector $\boldsymbol{v}$: $O(N^\rho)$.*

Although Lemma 1 only shows how to choose k and t to minimize the time complexity, we can also tune k and t so that we use more time and less space. In a way this algorithm can be seen as a generalization of the naive brute-force search method, as $k = 0$ and $t = 1$ corresponds to checking the whole list for nearby vectors in linear time and linear space.

2.6 Cross-polytope locality-sensitive hashing

Whereas the previous subsections covered techniques previously used in [32] and [33], we deviate from these papers by the choice of hash function. The hash function we will use is the one originally described by Terasawa and Tanaka [60] using simplices and orthoplices (cross polytopes), later analyzed by Andoni et al. [9]. The n-dimensional cross-polytope is defined by the vertices $\{\pm \boldsymbol{e}_i\}$, and the corresponding hash function based on using the n-dimensional cross-polytope is defined by finding the vector $\boldsymbol{h} \in \{\pm \boldsymbol{e}_i\}$ which is closest to the target vector $\boldsymbol{v}$. Alternatively, the hash function is defined as:

$$h(\boldsymbol{x}) = \pm \arg\max_i |x_i| \in \{\pm 1, \pm 2, \ldots, \pm n\}, \tag{3}$$

where the sign is equal to the sign of the absolute largest coordinate; if $\boldsymbol{v} = (3, -5)$ then $h(\boldsymbol{v}) = -2$ and $h(-\boldsymbol{v}) = 2$. Two vectors then have the same hash value if (i) the position of the absolute largest coordinate is the same, and (ii) the sign of this coordinate is the same for both vectors.

As this only defines one hash function rather than an entire hash family, we need to somehow rerandomize the hash function, which is done as follows. We denote by $\mathcal{A}$ the distribution on the space of $n \times n$ real matrices where each entry is drawn from a standard normal distribution $\mathcal{N}(0, 1)$. In other words, the distribution $\mathcal{A}$ outputs matrices $A = (a_{i,j}) \in \mathbb{R}^{n \times n}$ where $a_{i,j} \sim \mathcal{N}(0, 1)$ for all i, j. Then, by first multiplying a vector $\boldsymbol{v}$ with a random matrix $A \sim \mathcal{A}$ and then applying the base hash function h, we obtain a hash family $\mathcal{H}$ as

$$\mathcal{H} = \left\{ h_A : h_A(\boldsymbol{x}) \triangleq h(A\boldsymbol{x}), A \sim \mathcal{A} \right\}. \tag{4}$$

Using this hash family, we define probabilities by varying the matrix A, e.g.,

$$\mathbb{P}\left[h(\boldsymbol{v}) = h(\boldsymbol{w})\right] \triangleq \mathbb{P}_{h_A \sim \mathcal{H}}\left[h_A(\boldsymbol{v}) = h_A(\boldsymbol{w})\right] = \mathbb{P}_{A \sim \mathcal{A}}\left[h_A(\boldsymbol{v}) = h_A(\boldsymbol{w})\right]. \tag{5}$$

As suggested by experiments in [60], the above hash function family performs very well in practice for distinguishing between vectors with small and large angles (note that $\mathcal{H}$ is scale-invariant; $h(\lambda \boldsymbol{v}) = h(\boldsymbol{v})$ for arbitrary $\lambda > 0$). Terasawa and Tanaka already indicated that it seems to perform better than Charikar's angular or hyperplane hash family [13]. A recent study of Andoni et al. [9] shows that indeed it provably performs very well, leading to the following result on collision probabilities.

Lemma 2 (Cross-polytope locality-sensitive hashing). *[9, Theorem 1] Let $\theta = \theta(\boldsymbol{v}, \boldsymbol{w})$ denote the angle between two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$. Then, for large n,*

$$\mathbb{P}_{h\sim\mathcal{H}}\left[h(\boldsymbol{v}) = h(\boldsymbol{w})\right] = \exp\left[(-\ln n)\tan^2\left(\frac{\theta}{2}\right) + O(\log\log n)\right]. \tag{6}$$

For comparison later, we finally recall that for the spherical LSH family $\mathcal{S}$ described in [7] and used in the SphereSieve [33], we have the following result regarding collision probabilities.

Lemma 3 (Spherical locality-sensitive hashing). *[7, Lemma 3.3] Let $\theta = \theta(\boldsymbol{v}, \boldsymbol{w})$ denote the angle between two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$. Then, for large n,*

$$\mathbb{P}_{h\sim\mathcal{S}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] = \exp\left[-\frac{\sqrt{n}}{2}\tan^2\left(\frac{\theta}{2}\right)(1 + o(1))\right]. \tag{7}$$

Note that the leading-term dependence on θ in both spherical LSH and cross-polytope LSH is the same while the term in n is decreased from a former $\sqrt{n}/2$ to $\ln n$.

3 CPSieve: Sieving in arbitrary lattices

To combine sieving (the GaussSieve of Micciancio and Voulgaris) with locality-sensitive hashing (the cross-polytope LSH family of Terasawa and Tanaka) we will make the following changes to the GaussSieve, similar to [32, 33]:

- Instead of building a list of pairwise-reduced lattice vectors, we store each vector in t hash tables $T_1, \dots, T_t$.
- For each hash table T_i, we combine k hash functions $h_{i,1}, \dots, h_{i,k}$ into one function h_i with an AND-composition.
- To reduce a new vector with the vectors which are already in the hash tables, we only compare it to those vectors that have the same hash value in one or more of these t hash tables (OR-composition).
- When a vector is removed from the list and added to the stack, it is removed from all t hash tables before it is modified and added to S.
- When a vector is added to the list, it is inserted in the t hash tables in the buckets corresponding to its t hash values.

The main difference with previous work [32, 33] lies in the choice of the hash function family, which in this paper is the efficient and asymptotically superior cross-polytope LSH, rather than the asymptotically worse angular or hyperplane LSH [13, 32] or the less practical spherical LSH [8, 33]. This leads to the CPSieve algorithm described in Algorithm 2.

Algorithm 2 The CPSieve algorithm

```
1: Initialize an empty list $L$ and an empty stack $S$
2: Sample $t \cdot k$ random Gaussian matrices $A_{i,j}$
3: Define $h_{i,j}(\boldsymbol{x}) = h(A_{i,j}\boldsymbol{x})$ and $h_i(\boldsymbol{x}) = (h_{i,1}(\boldsymbol{x}), \dots, h_{i,k}(\boldsymbol{x}))$
4: Initialize $t$ empty hash tables $T_i$
5: repeat
6:     Get a vector $\boldsymbol{v}$ from the stack (or sample a new one)
7:     Obtain the set of candidates $C = \left(\bigcup_{i=1}^{t} T_i[h_i(\boldsymbol{v})] \cup \bigcup_{i=1}^{t} T_i[h_i(-\boldsymbol{v})]\right)$
8:     for each $\boldsymbol{w} \in C$ do
9:         Reduce $\boldsymbol{v}$ with $\boldsymbol{w}$
10:        Reduce $\boldsymbol{w}$ with $\boldsymbol{v}$
11:        if $\boldsymbol{w}$ has changed then
12:            Remove $\boldsymbol{w}$ from the list $L$
13:            Remove $\boldsymbol{w}$ from all $t$ hash tables $T_i$
14:            Add $\boldsymbol{w}$ to the stack $S$ (unless $\boldsymbol{w} = \mathbf{0}$)
15:        end if
16:    end for
17:    if $\boldsymbol{v}$ has changed then
18:        Add $\boldsymbol{v}$ to the stack $S$ (unless $\boldsymbol{v} = \mathbf{0}$)
19:    else
20:        Add $\boldsymbol{v}$ to the list $L$
21:        Add $\boldsymbol{v}$ to all $t$ hash tables $T_i$
22:    end if
23: until $\boldsymbol{v}$ is a shortest vector
```

3.1 Solving SVP in time and space $2^{0.298n+o(n)}$

To analyze the resulting algorithm and to choose suitable parameters k and t, what matters most is the performance of the underlying locality-sensitive hash functions; the better these functions are at separating reducible from unreducible pairs of vectors, the fewer hash functions and hash tables we will need and the faster the algorithm will be. In particular, as described in various literature on locality-sensitive hashing, to estimate the performance of the LSH family one should consider the parameter $\rho = \frac{\log 1/p_1}{\log 1/p_2}$.

Note that the LSH family $\mathcal{H}$ described in Section 2.6 has 'performance parameter' ρ as follows, where the collision probabilities $p_{1,2}$ correspond to certain angles $\theta_{1,2}$ between pairs of vectors:

$$\rho_{\mathcal{H}} = \frac{\log 1/p_1}{\log 1/p_2} = \frac{\tan^2\left(\frac{\theta_1}{2}\right)}{\tan^2\left(\frac{\theta_2}{2}\right)}\,(1 + o(1)). \tag{8}$$

Comparing this result to Andoni et al.'s spherical hash functions $h \in \mathcal{S}$ [7, 8] used in the SphereSieve [33], which have a collision probability of

$$\mathbb{P}_{h \sim \mathcal{S}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] = \exp\left[-\frac{\sqrt{n}}{2}\tan^2\left(\frac{\theta}{2}\right)(1 + o(1))\right], \tag{9}$$

it is clear that also this spherical LSH family $\mathcal{S}$ achieves a ρ of

$$\rho_{\mathcal{S}} = \frac{\log 1/p_1}{\log 1/p_2} = \frac{\tan^2\left(\frac{\theta_1}{2}\right)}{\tan^2\left(\frac{\theta_2}{2}\right)}(1 + o(1)). \tag{10}$$

In terms of analyzing the effects of the use of either of these hash families on sieving, this implies that both families achieve asymptotically equivalent exponents; the analysis from [33] to derive the optimal time and space complexities of $2^{0.298n+o(n)}$ also applies here, thus leading to the following result.

Theorem 1. *The here presented CPSieve heuristically solves SVP in time and space* $2^{0.2972n+o(n)}$ *using the following parameters:*

$$k = \Theta(n/\log n), \qquad t = 2^{0.0896n+o(n)}. \tag{11}$$

By varying k and t, we further obtain the trade-off between the time and space complexities indicated by the solid blue curve in Figure 1.

Proof. As the dependence on θ in the collision probabilities for $\mathcal{H}$ and $\mathcal{S}$ is the same, the analysis from [33, Appendix A] also applies to $\mathcal{H}$. The only impact of the different factor in the exponent of the collision probability (in terms of n) is the value of k, which after a similar analysis (where it should hold that the number of buckets roughly equals the eventual list size, i.e., $\Theta(n^k) \sim 2^{\Theta(n)}$) turns out to lead to the given expression for k.

Note that a major difference between the two hash families $\mathcal{H}$ and $\mathcal{S}$ is that computing a single hash value (for one hash function, before amplification) costs $2^{\Theta(\sqrt{n})}$ time for $\mathcal{S}$ and only at most $O(n^2)$ time for $\mathcal{H}$ (due to the matrix-vector multiplication by a random Gaussian matrix A). So by replacing $\mathcal{S}$ by $\mathcal{H}$, the cost of computing hashes goes down from subexponential (but superpolynomial) to only at most quadratic in n. Especially for large n, this means cross-polytope hashing will be orders of magnitude faster than spherical hashing, and may be competitive with the angular hashing of Charikar [13] used in the HashSieve [32, 41].

3.2 Practical aspects of the CPSieve

Although this theoretical result already offers a substantial (albeit subexponential) improvement over the SphereSieve, and an exponential improvement over other sieve algorithms, to make the resulting algorithm truly practical we would like to further reduce the worst-case quadratic cost of computing hashes.

Theoretically, to compute hashes we first multiply a target vector $\boldsymbol{v}$ by a fully random Gaussian matrix A where each entry $a_{i,j}$ is drawn from the same Gaussian distribution, and then look for the largest coordinate of $\boldsymbol{v}' = A\boldsymbol{v}$; the index of the largest coordinate of $\boldsymbol{v}'$ will be the hash value. Note that finding this largest coordinate, given $\boldsymbol{v}'$, can be done in worst-case linear time, and so the main bottleneck in computing hashes lies in computing the product $A\boldsymbol{v}$. As also described in [1, 32, 35], in practice it may be possible to reduce the amount of entropy in the hash functions (the "randomness") without significantly affecting the performance of the scheme. As long as the amount of entropy is high enough that we can build sufficiently many random, independent hash functions, the algorithm will generally still work fine. Some possibilities to reduce the complexity of computing hashes in practice are:

- Use low-precision floating-point matrices A.
- Use sparse random projection matrices.
- Use structured matrices that allow for fast matrix-vector multiplication.

Using structured matrices that allow for e.g. the use of Fast Fourier Transforms for computing matrix-vector multiplications may significantly reduce the cost of computing a hash value from $O(n^2)$ to $O(n \log n)$.

Probing The idea of probing, where various hash buckets in each hash table are traversed and checked for reductions with $\boldsymbol{v}$ (rather than only the bucket labeled $h(\boldsymbol{v})$), can also be applied to the CPSieve. For a given vector $\boldsymbol{v}$, the highest-quality bucket (the bucket most likely to contain vectors for reductions) is the one labeled $h(\boldsymbol{v})$, containing other vectors which also have the same index of the largest coordinate. It is not hard to see that the second-best bucket for reductions with $\boldsymbol{v}$ is exactly the bucket corresponding to the second-largest absolute coordinate of $\boldsymbol{v}$. For instance, if $\boldsymbol{v} = (3, -1, 8, -5, 11)$ then the vectors whose largest coordinate is the fifth coordinate are most likely to be useful for reductions, and the next best option to check is those vectors whose largest coordinate is the third coordinate. By checking multiple buckets in each hash table (rather than just one bucket), we may be able to reduce the number of hash tables and the overall space complexity by a polynomial factor at almost no cost.

For further details on clever (multi-)probing techniques for the cross-polytope LSH family $\mathcal{H}$, as well as ways to use structured matrices to reduce the quadratic cost of hashing, see [9].

3.3 Relation with angular hashing and a practical trade-off

To put the hash family $\mathcal{H}$ into context, recall that the angular hash family of Charikar [13] used in the HashSieve [32] is defined as follows: one samples a random vector $\boldsymbol{r} \in \mathbb{R}^n$ (its length is irrelevant), and assigns a hash value to a vector $\boldsymbol{v}$ based on whether the inner product $\boldsymbol{v} \cdot \boldsymbol{r}$ is positive ($h(\boldsymbol{v}) = 1$) or not ($h(\boldsymbol{v}) = 0$). Equivalently, we apply a suitable random projection to $\boldsymbol{v}$, and check whether v_1 is positive ($h(\boldsymbol{v}) = 1$) or not ($h(\boldsymbol{v}) = 0$).

In this way it is easy to see some similarities with cross-polytope hashing, where all (instead of only one) entries of $\boldsymbol{v}$ are compared and the index of the maximum of these entries (and the sign of the maximum entry) is used as the hash value. This suggests a natural generalization of both angular and cross-polytope hashing as follows:

$$\tilde{h}_m(\boldsymbol{x}) = \pm \underset{i \in \{1,\dots,m\}}{\arg\max}\, |x_i|. \qquad (1 \leq m \leq n) \tag{12}$$

Using random Gaussian projection matrices A and setting $m = 1$ then exactly corresponds to the angular hashing technique of Charikar, while with rerandomizations and $m = n$ we obtain the cross-polytope LSH family. This generalization with arbitrary m is also equivalent to first applying a random projection onto a low-dimensional subspace and then using the standard full-dimensional cross-polytope hash function in this low-dimensional space.

Note that although the CPSieve is asymptotically faster than the HashSieve, for the HashSieve the practical cost of computing hash values is much lower. To formalize this potential trade-off, note that for arbitrary m the hash function $\tilde{h}_m$ has $2m$ possible outcomes, and we eventually choose the parameter k to (asymptotically) satisfy that the total number of hash buckets in each hash table is roughly the same as the number of vectors in the system, i.e., $(2m)^k \approx 2^{0.21n}$. For given m, this translates to a condition on k as $k \approx \frac{0.21n}{\log_2 m+1}$. For actually computing hash values (for the moment ignoring the cost of the rerandomizations) we need to go through m of the vector coordinates to find the largest one in absolute value, incurring a cost of about m comparisons. In total, this means that for one hash table (which uses k hash functions) the cost of computing a vector's hash bucket is

$$(\text{Cost of computing the right bucket}) \approx k \cdot m \approx 0.21n \cdot \left[\frac{m}{\log_2 m + 1}\right]. \tag{13}$$

This suggests that to bring down the polynomial factors of computing hashes, we should choose m as small as possible, i.e. $m = 1$; this also explains why in low dimensions the HashSieve may outperform the CPSieve due to smaller polynomial terms. On the other hand, as m increases the asymptotic exponent of the algorithm's time complexity decreases from $0.337n + o(n)$ (the HashSieve) to $0.298n + o(n)$ (the CPSieve), so for high dimensions it is clear that setting $m = n$ is best. For moderate dimensions one might find the best option to be somewhere in between these two extremes. Experimentally we verified this to be the case for $n = 50$, where we heuristically found the best choice of m to lie significantly closer to $m = n$ than to $m = 1$; for fixed t, it seems we can slightly reduce the time complexity by less than 20% by choosing m slightly less than n, e.g. $m \approx 2n/3$.

3.4 Experimental results

We first show that already in mid-size dimensions ($n > 50$), we observe that the costs are similar to the asymptotic estimate for small choices of k. For a given dimension, we can vary the parameters t and k and observe varying numbers of vector comparisons, changes of the list size and number of hash computations.

For example, let us fix the number t of hash tables, $t \in [80; 120]$. We can now choose different values for k in practice that influence the probability that a candidate is a valid vector for reduction. A smaller k leads to a less restrictive hash value such that more vectors need to be checked for reduction. Increasing k produces a more restrictive hash value and we might need to increase the number t of hash tables to find good collisions; otherwise the list size may increase drastically, leading to a higher time complexity as well. Varying the parameters means trading time against memory as illustrated in Figures 2 and 3.[4]

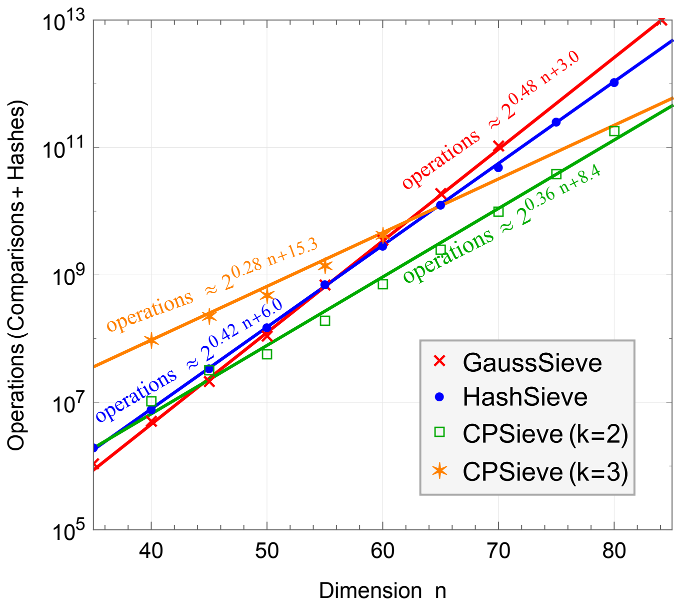

Fig. 2. A comparison of the number of operations (vector comparisons + hash computations) performed by the original GaussSieve algorithm, the angular HashSieve algorithm, and our proposed CPSieve algorithm using either $k = 2$ or $k = 3$ hash functions in each hash table (top).

Setting first $k = 2$, we performed experiments on random lattices in dimensions $n = 40$ to 80 with varying $t \in [80; 120]$ and observed an interpolated time complexity of around $0.36n + o(n)$ in logarithmic scale as illustrated by the lower (green) line in Figure 2. The advantage of this choice is a reduced list size which lies close to $0.21n + o(n)$ as depicted in Figure 3. If we wish to reduce the number of computations and to approach the minimal asymptotic time, we need to increase k (and t) with n which leads to larger list sizes of around $0.24n + o(n)$

[4] The figures represent the collected data at the time of submission. More fine grained tests w.r.t. the dimension and the various parameter choices are in progress an will be included in the final version.

in our experiments (cf. Figure 3). For $k = 3$ we observe a better approximation of the heuristic running time of $0.298n + o(n)$ as shown in Figure 2 by the upper (orange) line. The observed cost lies slightly below the asymptotic estimate.

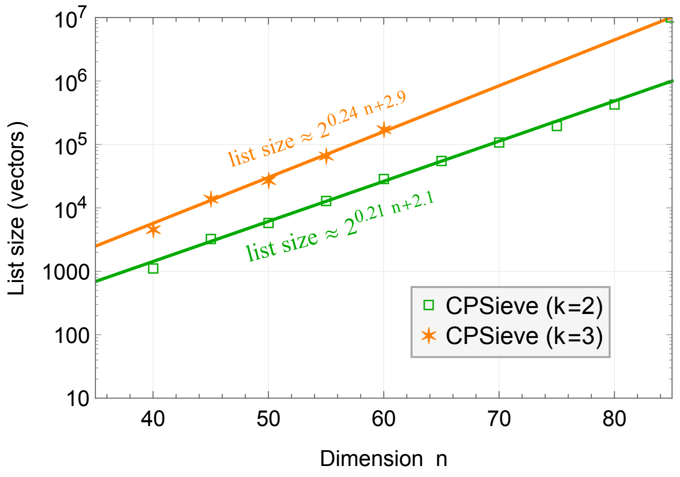

Fig. 3. Number of vectors in the list of the CPSieve in dimension 40 to 80 for optimal $t \in [80; 120]$ and $k = 2, 3$.

Figure 2 also shows how various algorithms from the literature compare, including (i) the GaussSieve, which performs an exhaustive search over the list L; (ii) the HashSieve, which uses hash tables based on angular LSH; and (iii) our new CPSieve algorithm, with parameters $k = 2, 3$. As indicated by the theoretical cost, the new CPSieve performs clearly better in terms of the asymptotic exponent, and this also appears from the experiments: the linear interpolation for the data based on the CPSieve in Figure 2 has a significantly smaller slope than both the GaussSieve and the HashSieve. In dimensions below 60 the polynomial factors for sieving still play an important role in practice, and therefore the absolute number of operations for CPSieve lies partially above the GaussSieve and/or the angular HashSieve.

Overall we see that the new algorithm has a distinguished lower increase in the complexity in practice compared to the traditional GaussSieve and the angular HashSieve, and the crossover points are already in low dimensions. As the gap between the CPSieve and other algorithms will only increase as n increases, this clearly highlights the potential of the CPSieve on arbitrary lattices.

4 IdealCPSieve: Sieving in ideal lattices

While the CPSieve is very capable of solving the shortest vector problem on arbitrary lattices, it was already shown in various papers [12, 27, 54] that for certain ideal lattices it is possible to obtain substantial polynomial speed-ups to sieving in practice, which may make sieving even more competitive with e.g. enumeration-based SVP solvers. As ideal lattices are commonly used in lattice cryptography, and our main goal is to estimate the complexity of SVP on lattices that are actually used in lattice cryptography, it is important to know if our proposed CPSieve can be sped up on ideal lattices as well. We will show that this is indeed the case, using similar techniques as in [12, 27, 54] but where we need to do some extra work to make sure these speed-ups apply here as well.

4.1 Ideal GaussSieve

For the ideal lattices mentioned in the preliminaries, cyclic shifts of a vector are also in the lattice (modulo minus signs) and have the same Euclidean norm. As first described by Schneider [54], this property can be used in the GaussSieve as follows. First, note that any vector $\boldsymbol{v}$ can be viewed as representing n vectors, namely its n shifted versions $\boldsymbol{v}, \boldsymbol{v}_{(1)}, \boldsymbol{v}_{(2)}, \dots, \boldsymbol{v}_{(n-1)}$, where we write $\boldsymbol{x}_{(s)} = (x_{n-s+1}, \dots, x_n, x_1, \dots, x_{n-s})$ for the sth cyclic right-shift of $\boldsymbol{x} = (x_1, \dots, x_n)$. Similarly, another vector $\boldsymbol{w}$ represents n different lattice vectors $\boldsymbol{w}, \boldsymbol{w}_{(1)}, \boldsymbol{w}_{(2)}, \dots, \boldsymbol{w}_{(n-1)}$.

Non-ideal GaussSieve: In the standard GaussSieve, we would treat these $2n$ shifts of $\boldsymbol{v}$ and $\boldsymbol{w}$ as different vectors, and we would store all of them in the system, leading to a storage cost of $2n$ vectors. Furthermore, to make sure that the list remains pairwise reduced, all $\binom{2n}{2} \approx 2n^2$ pairs of vectors are compared for reductions, leading to a time cost of approximately $2n^2$ vector comparisons.

Ideal GaussSieve: To make use of the cyclic structure of certain ideal lattices, the main idea of the ideal GaussSieve is that comparing $\boldsymbol{v}_{(s)}$ to $\boldsymbol{w}_{(s')}$ is the same as comparing $\boldsymbol{v}_{(s-s')}$ to $\boldsymbol{w}$ for any s, s': there exist shifts of $\boldsymbol{v}$ and $\boldsymbol{w}$ that can (cannot) reduce each other if and only if there exists a shift of $\boldsymbol{v}$ that can reduce (be reduced by) $\boldsymbol{w}$. So instead of storing all $2n$ shifts, we only store the two representative vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ in the system (storage cost of 2 vectors), and more importantly, to see if any of the shifts of $\boldsymbol{v}$ and $\boldsymbol{w}$ can reduce each other we only compare all n shifts of $\boldsymbol{v}$ to the single vector $\boldsymbol{w}$ stored in memory (n comparisons). To make sure that also $\boldsymbol{v}$ ($\boldsymbol{w}$) and its own cyclic shifts are pairwise reduced, we further need $n/2$ ($n/2$) comparisons to compare $\boldsymbol{v}$ to $\boldsymbol{v}_{(s)}$ ($\boldsymbol{w}$ to $\boldsymbol{w}_{(s)}$) for $s = 1, \dots, n/2$. In total, we therefore need $n + n/2 + n/2 = 2n$ comparisons to reduce $\boldsymbol{v}, \boldsymbol{w}$ and all their cyclic shifts.

Overall, this shows that in cyclic and negacyclic lattices, the memory cost of the GaussSieve goes down by a factor n, and the number of inner products that we compute to make sure the list is pairwise reduced also goes down by a factor approximately n. Although only polynomial, a factor 100 speedup and using 100 times less memory in dimension 100 can be very useful.

4.2 Hashing shifted vectors is shifting hashes of vectors

To see how we can obtain similar improvements for the CPSieve, let us first look at the basic hash function $h(\boldsymbol{x}) = \pm \arg\max_i x_i$. Suppose we have a cyclic lattice, and for some lattice vector $\boldsymbol{v}$ we have $h(\boldsymbol{v}) = i$ for some $i \in \{1, \dots, n\}$. Due to the choice of the hash function, we know that if we shift the entries of $\boldsymbol{v}$ to the right by s positions to get $\boldsymbol{v}_{(s)}$, then the hash of this vector will increase by s as well, modulo n:

$$h(\boldsymbol{v}_{(s)}) = [h(\boldsymbol{v}) + s] \bmod n, \tag{14}$$

where the result of the modular addition is assumed to lie in $\{1, \dots, n\}$. As a result, we know that $h(\boldsymbol{v}) = h(\boldsymbol{w})$ if and only if $h(\boldsymbol{v}_{(s)}) = h(\boldsymbol{w}_{(s)})$ for any s. For the basic hash function h, this property allows us to use a similar trick as in the ideal GaussSieve: we only store one representative of $\boldsymbol{w}$ in the hash tables, and for reducing $\boldsymbol{v}$ we compare all n shifts $\boldsymbol{v}_{(s)}$ to the lattice vectors in their corresponding buckets $h(\boldsymbol{v}_{(s)})$. We are then guaranteed that if any pair of vectors $\boldsymbol{v}_{(s)}$ and $\boldsymbol{w}_{(s')}$ can be reduced and have the same hash value, we will encounter this reduction when we compare $\boldsymbol{v}_{(s-s')}$ and $\boldsymbol{w}$ as they will also have the same hash values and can reduce each other.

4.3 Ideal rerandomizations through circulant matrices

While this shows that the basic hash function h has this nice property that allows us to obtain the linear decreases in the time and space complexity similar to the ideal GaussSieve, to make this algorithm work we will need many different hash functions from $\mathcal{H}$ for each of the hash tables for the AND- and OR-compositions; in particular, the number of hash tables t (and therefore also the number of hash functions) increases exponentially with n. And once we apply a random rotation to a vector, we may lose the property described in (14):

$$h_A(\boldsymbol{v}_{(s)}) = h(A\boldsymbol{v}_{(s)}) \overset{?}{=} [h(A\boldsymbol{v}) + s] \bmod n = [h_A(\boldsymbol{v}) + s] \bmod n, \tag{15}$$

The second equality is crucial here, as without preserving the property that the hash of a shift of a vector equals the shift of the hash of a vector, it might be that there exists a pair of vectors $\boldsymbol{v}_{(s)}$ and $\boldsymbol{w}_{(s')}$ that can be reduced *and has the same hash value*, while we will not reduce $\boldsymbol{v}_{(s-s')}$ and $\boldsymbol{w}$ because they have different hash values. If that happens, then not all $2n$ shifts of both vectors are pairwise reduced, which implies that the 'quality' of the list goes down, so the list size goes up, and we lose the factor n speedup again.

To guarantee that the second equality in (15) is always an equality, we would like to make sure that $A\boldsymbol{v}_{(s)} = (A\boldsymbol{v})_{(s)}$, i.e., multiplying a shifted vector by A is the same as shifting the vector which has already been multiplied by A. After all, in that case we would have

$$h_A(\boldsymbol{v}_{(s)}) = h(A\boldsymbol{v}_{(s)}) = h((A\boldsymbol{v})_{(s)}) = [h(A\boldsymbol{v}) + s] \bmod n = [h_A(\boldsymbol{v}) + s] \bmod n, \tag{16}$$

where the second equality follows from the condition $A\boldsymbol{v}_{(s)} = (A\boldsymbol{v})_{(s)}$ and the third equality follows from the property (14) of the base hash function h. So if we can guarantee that $A\boldsymbol{v}_{(s)} = (A\boldsymbol{v})_{(s)}$ for all $\boldsymbol{v}$ and s, then also these rerandomized hash functions satisfy the property we need to obtain a linear speedup. Now, it is not hard to see that $A\boldsymbol{v}_{(s)} = (A\boldsymbol{v})_{(s)}$ for all $\boldsymbol{v}$ and s is equivalent to the fact that A is circulant; substituting $\boldsymbol{v} = \boldsymbol{e}_1$ and varying $s = 1, \dots, n$ tells us that $a_{i,j} = a_{1,[j-i+1] \bmod n}$ for all i and j. In other words, we are free to choose the first row of A, and the ith row of the matrix is then defined as the $(i-1)$th cyclic shift of A.

So finally, the question becomes: can we simply impose the condition that A is circulant? While proving that the answer is yes or no seems hard, experimentally the answer seems to be yes: by only generating the first rows of each rerandomization matrix A at random from a standard Gaussian distribution, and then deriving the remaining entries of A from the first row, we obtain circulant matrices which appear to be as suitable for random rotations as fully random Gaussian matrices. The resulting circulant matrices on average appear to be as orthogonal as non-circulant ones, thus preserving relative norms and distances between vectors, and do not seem to perform worse in our experiments than non-circulant matrices.

Remark 1. The angular/hyperplane hash function of the HashSieve [13, 32], as well as the spherical hash functions in the SphereSieve [7, 33] do not have the properties mentioned above, and so while it may be possible to obtain the trivial decrease in the space complexity of a factor n, it seems impossible to obtain the factor n time speedup described above that applies to the GaussSieve and to the CPSieve.

Remark 2. By using circulant matrices, computing hashes of shifted vectors (to compare all shifts of a target vector $\boldsymbol{v}$ against the vectors in the hash tables) can be done by shifting the hash of the original vector. Also, one can compute the product of a circulant matrix with an arbitrary vector in $O(n \log n)$ time using Fast Fourier Transforms [22] instead of $O(n^2)$ time, which for large n may further reduce the overall time complexity of the algorithm. However, the even faster random rotations described in [9] which may be useful for the non-ideal case do not apply here, as we need A to be circulant to obtain the factor n speedup.

4.4 Power-of-2 cyclotomic ideal lattices ($X^n + 1$)

For our experiments we will consider two specific classes of ideal lattices, the first of which is the class of ideal lattices over the ring $\mathbb{Z}[X]/(X^n + 1)$ where n is a power of 2. These are negacyclic lattices, and so for any lattice vector $\boldsymbol{v}$ all its $2n$ shifts are in the lattice as well, and $\boldsymbol{v}_{(n)} = -\boldsymbol{v}$. As for comparisons in the GaussSieve/CPSieve we usually compare both $\pm\boldsymbol{v}$ to candidate vectors $\boldsymbol{w}$, in this case this corresponds to going through all $2n$ shifts of a target vector $\boldsymbol{v}$ (which all have different hash values) and searching the hash buckets for vectors that

Algorithm 3 Reducing a vector $\boldsymbol{v}$ in the IdealCPSieve

```
1: for each hash table $T_i$ do
2:     Compute the $k$ base hash values $(H_1, \dots, H_k) = (h_{i,1}(\boldsymbol{v}), \dots, h_{i,k}(\boldsymbol{v}_{(s)}))$
3:     for each cyclic shift $s = 0, \dots, 2n-1$ do
4:         Compute $\boldsymbol{v}_{(s)}$'s partial hash values $H_i^{(s)} = [H_i + s] \bmod 2n$
5:         Compute $\boldsymbol{v}_{(s)}$'s hash value $h_i(\boldsymbol{v}_{(s)}) = f(H_1^{(s)}, \dots, H_k^{(s)})$
6:         for each $\boldsymbol{w} \in T_i[h_i(\boldsymbol{v}_{(s)})]$ do
7:             Reduce $\boldsymbol{v}_{(s)}$ with $\boldsymbol{w}$
8:             Reduce $\boldsymbol{w}$ with $\boldsymbol{v}_{(s)}$
9:             ...
10:        end for
11:    end for
12: end for
13: ...
14: if $\boldsymbol{v}$ and its shifts have not changed then
15:     Add $\boldsymbol{v}$ (and only $\boldsymbol{v}$!) to all hash tables
16: end if
```

may reduce these vectors. In short, for each new target vector taken from the stack, the algorithm will proceed as described in Algorithm 3. For convenience, we will assume that negative partial hash values $h_{i,j}(\boldsymbol{v}) < 0$ are replaced by $h'_{i,j}(\boldsymbol{v}) = n - h_{i,j}(\boldsymbol{v})$, so that the partial hash values always lie in the range $1, \dots, 2n$ and are consecutive hash values of consecutive shifted vectors.

4.5 NTRU lattices ($X^n - 1$)

The lattice basis of an NTRU encryption scheme [24, 25] can be described by a prime power p, the ring $R = \mathbb{Z}_q[X]/(X^p - 1)$, a small power q of two and two polynomials $f, g \in R$ with small coefficient, for example in $\{-1, 0, 1\}$. We require that f is invertible in R and set $h = g/f \mod q$. The public basis is then given by p, q and h as the $n \times n$ matrix M (where $n = 2p$) as follows:

$$M = \left(\begin{array}{cccc|cccc} q & & & & & & & \\ & q & & & & 0 & & \\ & & \ddots & & & & & \\ & & & q & & & & \\ \hline h_0 & h_1 & \cdots & h_{n-1} & 1 & & & \\ h_{n-1} & h_0 & \cdots & h_{n-2} & & 1 & & \\ \vdots & \vdots & \ddots & \vdots & & & \ddots & \\ h_1 & h_2 & \cdots & h_0 & & & & 1 \end{array}\right).$$

Note that not only (f, g) but also all block-wise rotations (fX^k, gX^k) are short vectors in the lattice. More generally, we observe that each block of $p = n/2$ entries of a lattice vector can be shifted (without minus sign) to obtain another valid lattice vector.

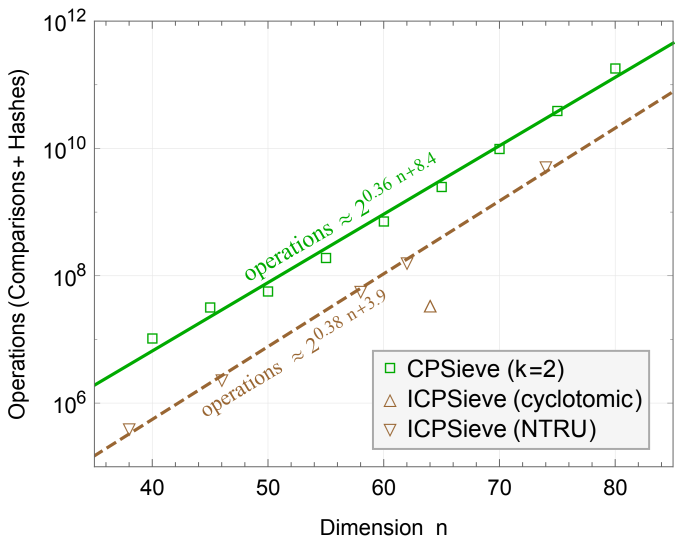

Fig. 4. A comparison between the performance of our algorithm on arbitrary and ideal lattices for $k = 2$ (bottom).

For these lattices we can apply similar techniques as in the previous subsection, but in this case we only have $n/2$ shifts of a vector in n dimensions; the speedups and memory gains are not equal to the dimension, but only to half the dimension of the lattice we are trying to tackle. The improvement we expect with respect to the non-ideal case will therefore be less than for the power-of-2 lattices described above.

4.6 Experiments for ideal lattices

For testing the performance of SVP algorithms on ideal lattices, we focused on NTRU lattices where $n = 2p$ and p is prime, and on negacyclic lattices where $n = 2^s$ is a power of 2, which can be generated with the ideal lattice challenge generator [48]. For the NTRU lattices we considered values $n = 38, 46, 58, 62, 74$, while for the cyclotomic lattices we restricted our experiments to only $n = 64$; for $n = 32$ the data will be unreliable as the algorithm terminates very quickly and the basis reduction sometimes already finds a shortest vector, while $n = 128$ is out of reach for our single-core proof-of-concept implementation; investigating the costs of solving the 128-dimensional ideal lattice challenge with the IdealCPSieve, as done in [12, 27], is left for future work.

The limited set of experiments performed as expected, and the results are shown in Figure 4 in comparison to the random, non-ideal complexities of the CPSieve. The costs in the ideal case are decreased by a factor linear in n as we make use of the (block) cyclic structure of the respective ideal lattices as outlined in the previous subsections. We expect an analogue observation for different choices of the parameters. Note that for cyclotomic lattices we get a better exponent as the speedup and memory improvement are equal to n, rather than $n/2$ for NTRU lattices.

5 Conclusion

We presented new algorithms for the shortest vector problem, making use of a special locality-sensitive hash family that performs well both in theory and in practice. Using the previous heuristic analysis of Laarhoven and De Weger we derived that this algorithm has an asymptotic time and space complexity of $2^{0.298n+o(n)}$, thus leading to an exponential improvement over e.g. the GaussSieve and the HashSieve, and a substantial subexponential speedup over the SphereSieve. Experiments validate our heuristic analyses, and show that already in moderate dimensions the CPSieve may outperform other algorithms. As the advantage over other methods will only grow in higher dimensions, we expect CPSieve to form an important guide for assessing the hardness of SVP in high dimensions on arbitrary lattices.

As the base hash function lends itself well for speedups on (nega)cyclic lattices, we then investigated whether these speedups can also be applied to the entire hash family. By choosing the rerandomization matrices A appropriately we argued that indeed this can be achieved, and we experimentally verified that our IdealCPSieve can solve SVP on ideal lattices significantly faster than on non-ideal lattices; something that does not often occur for lattice algorithms. We further expect that a fully optimized, parallel implementation of IdealCPSieve is able to solve the 128-dimensional lattice challenge faster than the GaussSieve.

Acknowledgments

The authors gratefully acknowledge Léo Ducas for discussions and comments that eventually led to the study of these hash functions. The authors further thank Nicolas Gama and Benne de Weger for various useful discussions on this and related topics during a visit at EPFL. The authors also thank the authors of [9] for providing an early draft of this paper for reference and for use here. The second author acknowledges Memphis Depay, Meilof Veeningen and Niels de Vreede for their inspiration.

References

1. Achlioptas, D.: Database-friendly random projections. In: PODS, pp. 274–281 (2001)
2. Aggarwal, D., Dadush, D., Regev, O., Stephens-Davidowitz, N.: Solving the shortest vector problem in 2^n time via discrete Gaussian sampling. In: STOC (2015)
3. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: STOC, pp. 99–108, (1996)

4. Ajtai, M.: The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In: STOC, pp. 10–19 (1998)
5. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: STOC, pp. 601–610 (2001)
6. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: FOCS, pp. 459–468 (2006)
7. Andoni, A., Indyk, P., Nguyen, H. L., Razenshteyn, I.: Beyond locality-sensitive hashing. In: SODA, pp. 1018–1028 (2014)
8. Andoni, A., Razenshteyn, I.: Optimal data-dependent hashing for approximate near neighbors. In: STOC (2015)
9. Andoni, A., Indyk, P., Kapralov, M., Laarhoven, T., Razenshteyn, I., Schmidt, L.: Fast and optimal LSH for cosine similarity. Work in progress (2015)
10. Becker, A., Gama, N., Joux, A.: A sieve algorithm based on overlattices. In: ANTS, pp. 49–70 (2014)
11. Bernstein, D. J., Buchmann, J., Dahmen, E.: Post-quantum cryptography (2009)
12. Bos, J. W., Naehrig, M., van de Pol, J.: Sieving for Shortest Vectors in Ideal Lattices: a Practical Perspective. Cryptology ePrint Archive, Report 2014/880 (2014)
13. Charikar, M. S.: Similarity estimation techniques from rounding algorithms. In: STOC, pp. 380–388 (2002)
14. Conway, J. H., Sloane, N. J. A.: Sphere packings, lattices and groups (1999)
15. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: SOCG, pp. 253–262 (2004)
16. Fincke, U., Pohst, M.: Improved methods for calculating vectors of short length in a lattice. Mathematics of Computation 44(170), pp. 463–471 (1985)
17. Fitzpatrick, R., Bischof, C., Buchmann, J., Dagdelen, Ö., Göpfert, F., Mariano, A., Yang, B.-Y.: Tuning GaussSieve for speed. In: LATINCRYPT, pp. 284–301 (2014)
18. Gama, N., Nguyen, P. Q.: Predicting lattice reduction. In: EUROCRYPT, pp. 31–51 (2008)
19. Gama, N., Nguyen, P. Q., Regev, O.: Lattice enumeration using extreme pruning. In: EUROCRYPT, pp. 257–278 (2010)
20. Garg, S., Gentry, C., Halevi, S.: Candidate multilinear maps from ideal lattices. In: EUROCRYPT, pp. 1–17 (2013)
21. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC, pp. 169–178 (2009)
22. Golub, G. H., Van Loan, C. F.: Matrix computations. John Hopkins University Press (2012)
23. Hanrot, G., Pujol, X., Stehlé, D.: Algorithms for the shortest and closest lattice vector problems. In: IWCC, pp. 159–190 (2011)
24. Hoffstein, J., Pipher, J., Silverman, J. H.: NTRU: A ring-based public key cryptosystem. In: ANTS, pp. 267–288 (1998). Previously presented at the CRYPTO'96 rump session.
25. Hoffstein, J., Pipher, J., Silverman, J. H.: NS: An NTRU lattice-based signature scheme. In: EUROCRYPT, pp. 211–228 (2001)
26. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
27. Ishiguro, T., Kiyomoto, S., Miyake, Y., Takagi, T.: Parallel Gauss Sieve algorithm: solving the SVP challenge over a 128-dimensional ideal lattice. In: PKC, pp. 411–428 (2014)

28. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: STOC, pp. 193–206 (1983)
29. Khot, S.: Hardness of approximating the shortest vector problem in lattices. In: FOCS, pp. 126–135 (2004)
30. Klein, P.: Finding the closest lattice vector when it's unusually close. In: SODA, pp. 937–941 (2000)
31. Kleinjung, T.: Private communication (2014)
32. Laarhoven, T.: Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In: CRYPTO (2015)
33. Laarhoven, T., de Weger, B.: Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In: LATINCRYPT (2015)
34. Lenstra, A. K., Lenstra, H. W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), pp. 515–534 (1982)
35. Li, P., Hastie, T. J., Church, K. W.: Very sparse random projections. In: KDD, pp. 287–296 (2006)
36. Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: CT-RSA, pp. 319–339 (2011)
37. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: VLDB, pp. 950–961 (2007)
38. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: EUROCRYPT, pp. 1–23 (2010)
39. Mariano, A., Timnat, S., Bischof, C.: Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. In: SBAC-PAD (2014)
40. Mariano, A., Dagdelen, Ö., Bischof, C.: A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In: APCI&E (2014)
41. Mariano, A., Laarhoven, T., Bischof, C.: Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In: ICPP (2015)
42. Micciancio, D., Voulgaris, P.: A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations. In: STOC, pp. 351–358 (2010)
43. Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: SODA, pp. 1468–1480 (2010)
44. Micciancio, D., Walter, M.: Fast lattice point enumeration with minimal overhead. In: SODA, pp. 276–294 (2015)
45. Milde, B., Schneider, M.: A parallel implementation of GaussSieve for the shortest vector problem in lattices. In: PaCT, pp. 452–458 (2011)
46. Nguyen, P. Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. J. Math. Crypt. 2(2), pp. 181–207 (2008)
47. Panigraphy, R.: Entropy based nearest neighbor search in high dimensions. In: SODA, pp. 1186–1195 (2006)
48. Plantard, T., Schneider, M.: Ideal lattice challenge. Online at `http://latticechallenge.org/ideallattice-challenge/` (2014)
49. Pohst, M. E.: On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. ACM SIGSAM Bulletin 15(1), pp. 37–44 (1981)
50. van de Pol, J., Smart, N. P.: Estimating key sizes for high dimensional lattice-based systems. In: IMACC, pp. 290–303 (2013)
51. Pujol, X., Stehlé, D.: Solving the shortest lattice vector problem in time $2^{2.465n}$. Cryptology ePrint Archive, Report 2009/605 (2009)
52. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: STOC, pp.84–93 (2005)

53. Schneider, M.: Analysis of Gauss-Sieve for solving the shortest vector problem in lattices. In: WALCOM, pp. 89–97 (2011)
54. Schneider, M.: Sieving for short vectors in ideal lattices. In: AFRICACRYPT, pp. 375–391 (2013)
55. Schneider, M., Gama, N., Baumann, P., Nobach, L.: SVP challenge. Online at `http://latticechallenge.org/svp-challenge` (2014)
56. Schnorr, C.-P.: A hierarchy of polynomial time lattice basis reduction algorithms. Theoretical Computer Science 53(2), pp. 201–224 (1987)
57. Schnorr, C.-P., Euchner, M.: Lattice basis reduction: improved practical algorithms and solving subset sum problems. Mathematical Programming 66(2), pp. 181–199 (1994)
58. Shoup, V.: Number Theory Library (NTL), v6.2. Online at `http://www.shoup.net/ntl/` (2014)
59. Stehlé, D., Steinfeld, R.: Making NTRU as secure as worst-case problems over ideal lattices. In: EUROCRYPT, pp. 27–47 (2011)
60. Terasawa, K., Tanaka, Y.: Spherical LSH for approximate nearest neighbor search on unit hypersphere. In: WADS, pp. 27–38 (2007)
61. Wang, X., Liu, M., Tian, C., Bi, J.: Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In: ASIACCS, pp. 1–9 (2011)
62. Zhang, F., Pan, Y., Hu, G.: A three-level sieve algorithm for the shortest vector problem. In: SAC, pp. 29–47 (2013)