Improved Quantum Hypercone Locality Sensitive Filtering in Lattice Sieving

Max Heiser[*]

Abstract

The asymptotically fastest known method for solving SVP is via lattice sieving, an algorithm whose computational bottleneck is solving the Nearest Neighbor Search problem. The best known algorithm for solving this problem is Hypercone Locality Sensitive Filtering (LSF). The classical time complexity of a sieve using Hypercone LSF is $2^{0.2925d+o(d)}$. The quantum time complexity is $2^{0.2653d+o(d)}$, which is acquired by using Grover's algorithm to speed up part of the enumeration.

We present an improvement to the quantum algorithm, which improves the time complexity to $2^{0.2571d+o(d)}$. Essentially, we provide a way to use Grover's algorithm to speed up another part of the process, providing a better tradeoff. This improvement affects the security of lattice-based encryption schemes, including NIST PQC Round 3 finalists.

1 Introduction

Lattice-based cryptography has received wide attention in recent years, in part due to its perceived resistance against quantum attacks. As such, the vast majority of applicants in NIST's post-quantum standardization process are lattice-based. Attacks against lattice-based ciphers often require finding the shortest vector in a lattice whose shortest vector is exceptionally short. This is the unique SVP problem. The most efficient known algorithm for solving this problem is BKZ [4, 24, 28], which uses an exact SVP solver of a smaller dimension as a subroutine.

The asymptotically fastest known algorithms for exact SVP are lattice sieving algorithms. Their time complexity is $2^{cd+o(d)}$, where d is the dimension of the lattice and c is a constant determined by which sieving algorithm is used. Compared to enumeration algorithms [11, 13, 23], whose time complexity is $2^{\omega(d)}$, sieving algorithms achieve better performance in high dimensions at the expense of large memory requirements. Accordingly, the current record in the SVP challenge [27] solves SVP in dimension 180 by sieving [10], and the top 10 performances all use sieving algorithms. Reducing the constant c in the exponent of the time complexity of sieving algorithms directly affects the expected security of lattice-based cryptographic schemes [20, 28].

The main procedure in lattice sieving is finding close pairs of lattice vectors in a long list, and substracting one vector from the other to create a shorter lattice vector. When performing this procedure repeatedly, we continually shorten the vectors in the list until

[*]heisermax@protonmail.com

it contains the shortest vector in the lattice. The problem of finding close pairs of vectors in a list is the Nearest Neighbor Search problem [12].

The first sieving algorithm was introduced in [1], which was later shown to provably solve SVP in time $2^{2.465d+o(d)}$ [22, 25]. Heuristic analyses [21, 22] further suggested that sieving algorithms can solve SVP in time $2^{0.415d+o(d)}$, and presented practical improvements. Research into k-sieves and overlattice sieves [6, 29, 30] improved the heuristic time complexity to $2^{0.378d+o(d)}$. Further work [5, 15, 19] improved sieving by applying Locality Sensitive Hashing and Locality Sensitive Filtering techniques to the nearest neighbor search routine, improving the time complexity to $2^{0.292d+o(d)}$. Various practical and subexponential improvements have been suggested over time [2, 8, 9, 10, 17, 26].

Quantum versions of sieving algorithms were analysed in [16, 18]. Namely, the quantum sieve using locality sensitive filtering was shown to run in time $2^{0.265d+o(d)}$. Quantum versions of k-sieves were examined in [14], giving a time-space tradeoff. Recently, while we were conducting this research, a sieving algorithm using quantum random walks was suggested in [7] with time complexity $2^{0.2570d+o(d)}$. Several practical aspects of quantum sieving algorithms were examined in [3].

1.1 Contributions

We present an improvement to the quantum locality sensitive filtering algorithm described in [16], which improves its time complexity from $2^{0.265d+o(d)}$ to $2^{0.2571d+o(d)}$. The main idea is applying Grover's algorithm to a part of the calculation that was previously done classically. Namely, instead of classically finding all relevant filters for a vector, we use Grover's algorithm with a circuit that samples a random relevant filter. We conjecture this improvement is somewhat orthogonal to the improvement in [7], and in future work the two algorithms may possibly be combined.

Theorem 1 *There exists a quantum algorithm using locality sensitive filtering which (heuristically) solves SVP in dimension d with time complexity* $2^{0.2571d+o(d)}$.

Outline. In Section 2 we provide background on Locality Sensitive Filtering (LSF) as a general scheme and explain the core idea of our improvement in high level. In Section 3 we discuss hypercone LSF and the details of our algorithm, as well as its analysis. In Section 4 we discuss Random Product Codes and construct a circuit used by the algorithm.

2 Overview of Our Algorithm

As a part of lattice sieving, we are required to solve the following problem: Given a long list of N vectors in $\mathbb{R}^d$, find all pairs of close vectors. This is the Nearest Neighbor Search problem (NNS). In the context of lattice sieving, vectors are said to be close (or *neighbors*) if the angle between them is $\leq 60°$, and $N \approx (\sin 60°)^{-d} \approx 2^{0.2075d}$ such that every vector has ≈ 1 close neighbor in the list.

The best known algorithm for solving NNS is Locality Sensitive Filtering (LSF). The main idea is to create a set of filters, and use them to construct buckets such that vectors in the same bucket are more likely to be close. The algorithm is to then check all pairs of vectors that are in the same bucket, and find those which are close. The pseudo-code of the scheme is as follows:

```
LSF Scheme
(1) Generate filters $\{f_1, ...f_t\}$
(2) Initialize empty buckets $\{B_1, ..., B_t\}$
(3) For $v$ in $V$:
(4)     For $f_i$ in Filters($v$):
(5)         Add $v$ to $B_i$
(6) For $v$ in $V$:
(7)     For $f_i$ in Filters($v$):
(8)         For $w \neq v$ in $B_i$:
(9)             If $v$ is close to $w$:
(10)                Add $(v, w)$ to neighboring pair list
```

Notice that in order to perform loops (4) and (7) above efficiently, we need to quickly find the relevant filters for a given vector. The simple solution is to enumerate over all filters and check which of them is relevant for v. However, this enumeration would take much more time than necessary as v is not relevant for the vast majority of filters. Ideally, we would like to find the relevant filters for v in time $\tilde{O}(\mathcal{F})$, where $\mathcal{F}$ is the number of relevant filters. This operation is realized using *Random Product Codes*, which we discuss later. If each bucket contains $\tilde{O}(\mathcal{B})$ vectors on average, then the algorithm takes

$$\tilde{O}(N\mathcal{F} + N\mathcal{F} + N\mathcal{F}\mathcal{B})$$

where

- The first term accounts for constructing the buckets; (3-5)
- The second term accounts for computing the relevant filters of each vector; (7)
- The third term accounts for checking all vectors in the relevant filters' buckets. (8-10)

Note that runtime is optimized when $\mathcal{B} = 1$, that is, every bucket contains $\tilde{O}(1)$ vectors.

2.1 Quantum Improvements

When attempting to apply Grover's algorithm in order to speed up locality sensitive filtering, the first idea may be to use Grover search on each bucket. This reduces each bucket's search time by a square root, to get $\tilde{O}(N\mathcal{F} + N\mathcal{F} + N\mathcal{F}\sqrt{\mathcal{B}})$. However, this is not optimal: Note that the vast majority of buckets do not contain a neighbor of v, as v only has ≈ 1 neighbor in the entire list. We could concatenate all relevant buckets to one long list, and use Grover search to find the $\tilde{O}(1)$ close neighbors. This would take $\tilde{O}(N\mathcal{F} + N\mathcal{F} + N\sqrt{\mathcal{F}\mathcal{B}})$ time. This is, in essence, the quantum LSF algorithm in [16].

Our starting point for this paper is the following modification of the above algorithm. Currently, in order to find neighbors of v we classically find the relevant buckets for v, then apply Grover search to the union of these buckets. If we skip the classic step of finding these buckets and instead use Grover's algorithm with a circuit that samples a relevant bucket and a potential candidate from that bucket in $\tilde{O}(1)$ time, then the time complexity would be $\tilde{O}(N\mathcal{F} + N\sqrt{\mathcal{F} + \mathcal{F}\mathcal{B}}) = \tilde{O}(N\mathcal{F} + N\sqrt{\mathcal{F}} + N\sqrt{\mathcal{F}\mathcal{B}})$, essentially applying a square root factor to the second term (bucket iteration) in addition to the third (candidate iteration).

On first glance this does not seem to improve the runtime, as the first term (constructing the buckets) always dominates the second. For this reason, we need to add a small modification to the algorithm: Each filter will have two thresholds - a strong one and weak one. Only vectors passing the strong threshold will enter the corresponding bucket in lines (4-5). However, when iterating over relevant filters in line (7), we will consider a filter relevant even if v only passes the weak threshold. This will offload runtime from the construction of the buckets to the enumeration over filters, which results in an improvement because the latter is sped up by Grover's algorithm. The updated pseudo-code is as follows:

```
LSF Scheme (multiple thresholds)
(1) Generate filters $\{f_1, ...f_t\}$
(2) Initialize empty buckets $\{B_1, ..., B_t\}$
(3) For $v$ in $V$:
(4)     For $f_i$ in StrongFilters($v$):
(5)         Add $v$ to $B_i$
(6) For $v$ in $V$:
(7)     For $f_i$ in WeakFilters($v$):
(8)         For $w \neq v$ in $B_i$:
(9)             If $v$ is close to $w$:
(10)                Add $(v, w)$ to close pair list
```

The new runtime is $\tilde{O}(N\mathcal{F}_{\text{strong}} + N\sqrt{\mathcal{F}_{\text{weak}}} + N\sqrt{\mathcal{F}_{\text{weak}}\mathcal{B}})$, and the parameters can be balanced such that the three terms have the same order of magnitude. This is the essence of the quantum algorithm we present. We note that in [16] the method of multiple thresholds was used for a different purpose - to create a time-memory tradeoff, whereas here we use it to improve the runtime itself.

As mentioned above, we will later have to describe a circuit that samples a relevant filter (and a vector from the corresponding bucket) in $\tilde{O}(1)$ time. This extends the algorithm for calculating the $\mathcal{F}$ relevant filters in time $\tilde{O}(\mathcal{F})$ which is known for Random Product Codes.

In the next section we describe the algorithm and analysis in more detail, assuming the existence of a circuit for sampling filters. The construction of this circuit is described in section 4.

3 Details of Our Algorithm

3.1 Hypercone LSF

Recall the main idea of LSF: Create a set of filters, and use them to construct buckets such that vectors in the same bucket are more likely to be neighbors. In Hypercone LSF, each bucket is a hypercone around some central vector. In more detail, each filter is a unit vector f in some direction, and a vector v passes the filter if $\langle v, f\rangle \geq \beta$ for some fixed $\beta > 0$ (since the vectors v are usually normalized to have unit length, this condition is equivalent to the angle between v and f being at most $\cos^{-1}(\beta)$). When using both weak and strong thresholds (see subsection 2.1), we would use two values $0 < \alpha < \beta$, where α and β would correspond to the weak and strong threshold respectively. We say that v

and f are *α-close* or *β-close* if $\langle v, f\rangle \geq \alpha$ or $\langle v, f\rangle \geq \beta$ respectively. Vectors v_1, v_2 are said to be *neighbors* if $\langle v_1, v_2\rangle \geq \frac{1}{2}$ (equivalent to having angle $\leq 60°$).

As described in section 2, we first construct buckets containing the vectors that are β-close to each filter. Then, for each vector v we look for neighbors of v in the buckets corresponding to filters that are α-close to v.

One needs to make sure that (almost) every pair of neighbors is found via at least one of the filters. A pair of vectors is found via a certain filter if this filter is α-close to one of the vectors and β-close to the other. We construct $\tilde{O}(p^{-1})$ filters, where p is the probability of this condition holding for a single filter, namely $p = \left(1 - \frac{4}{3}(\alpha^2 - \alpha\beta + \beta^2)\right)^{d/2+o(d)}$.

In order for the algorithm to be efficient, one needs a method to quickly find the relevant (α-close or β-close) filters for a given vector. This is provided by Random Product Codes, a method for choosing the filters with a certain structure that allows one to find the relevant filters in time $\tilde{O}(\mathcal{F})$, where $\mathcal{F}$ is the number of relevant filters. In our improvement, we extend this algorithm to enable sampling a relevant filter in $\tilde{O}(1)$ time.

3.2 Analysis of Hypercone LSF

We use similar notation to [16]. We have $n = 2^{c_n d+o(d)}$ vectors in $\mathbb{R}^d$ (with $c_n \approx 0.2075$ in the context of lattice sieving), and we create $t = 2^{c_t d+o(d)}$ filters. For each filter f, we create a bucket of all vectors v that are β-close to f . We achieve this by iterating over all vectors v, finding the β-close filters for v and placing v in the corresponding buckets. When searching for neighbors of a vector v, we compute the filters that are α-close to v, and search for neighbors in the corresponding buckets. The probability of a random vector being α-close or β-close to a given filter is $2^{c_\alpha d+o(d)}$ or $2^{c_\beta d+o(d)}$ respectively, where $c_x = \frac{1}{2}\log_2(1 - x^2)$ for $x \in \{\alpha, \beta\}$. The constraint that most close pairs of vectors are found this way with high probabilty translates into $c_t = -\frac{1}{2}\log_2(1 - \frac{4}{3}(\alpha^2 - \alpha\beta + \beta^2))$.

In our analysis, we use the heuristic that the lattice points in our list behave similarly to uniformly random points on the sphere. This heuristic is frequently used in analysis of sieving algorithms, and has been studied in [22]. We further assume that the filters constructed by the random product code behave as uniformly random points on the sphere. This heuristic was justified in [5, 16].

3.2.1 Classical Complexity

The classical complexity of Hypercone LSF is $2^{c_{\text{time}} d+o(d)}$, where

$$c_{\text{time}} = \max\left\{c_t + c_\beta, c_t + c_\alpha, c_t + c_\alpha + c_n + c_\beta\right\} + c_n$$

The three terms inside the $\max\{\cdot\}$ account for three parts of the process for each vector:

- The first term accounts for placing each vector into the β-close buckets;
- The second term accounts for computing the α-close filters of a vector;
- The third term accounts for checking all vectors in the relevant buckets.

Optimizing for time complexity, we get $c_{\text{time}} \approx 0.2925$ and $\alpha = \beta = \frac{1}{2}$.

3.2.2 Quantum Complexity

The quantum complexity of Hypercone LSF as described in [16] is $2^{q_{\text{time}} d+o(d)}$, where

$$q_{\text{time}} = \max\left\{c_t + c_\beta, c_t + c_\alpha, \frac{1}{2}\left(c_t + c_\alpha + c_n + c_\beta\right)\right\} + c_n$$

The third term is halved by using Grover search on the list of candidate vectors. Optimizing for time complexity, we get $q_{\text{time}} \approx 0.2653$ and $\alpha = \beta = \frac{\sqrt{3}}{4}$.

3.2.3 Improved Quantum Complexity

We propose an improvement to quantum Hypercone LSF, in which the new time complexity is $2^{q'_{\text{time}} d+o(d)}$, where

$$q'_{\text{time}} = \max\left\{c_t + c_\beta, \frac{1}{2}\left(c_t + c_\alpha\right), \frac{1}{2}\left(c_t + c_\alpha + c_n + c_\beta\right)\right\} + c_n$$

Optimizing for time complexity, we get $q'_{\text{time}} \approx 0.2571$ and $\beta = \frac{1}{2}, \alpha \approx 0.4434$. For reference, the values of the relevant parameters are

$$c_\alpha = -0.1579,\ c_\beta = -0.2075,\ c_t = 0.2571,\ c_n = 0.2075$$

In order to achieve this time complexity, we design a circuit that samples an α-close filter to v, and a vector in the corresponding bucket. Using Grover's algorithm with this quantum circuit to search for neighbors, we acquire a $\frac{1}{2}$ factor on the last two complexity terms, rather than just the third.

4 Circuit for Filter Sampling

4.1 Random Product Codes

The filters in LSF are constructed via Random Product Codes: We choose b, m such that $d = b \cdot m$ where $m = O(\log d)$. We let the set of filters be $C = C_1 \times \cdots \times C_m$, where the subcodes $C_i = \{f_i^{(1)}, ...f_i^{(t')}\} \subset \sqrt{1/m} \cdot S^{b-1}$ for $i = 1, ..., m$ are sets of $t' = t^{1/m}$ random, independent, uniform vectors over the sphere $\sqrt{1/m} \cdot S^{b-1}$. The parameter m is taken to be $O(\log d)$ to enable efficient decoding as well as sufficient randomness (see [5, 16] for further details).

The structure of Random Product Codes can be utilized to quickly find the relevant filters for a vector v: Suppose that we wish to find all of the α-close filters, i.e. filters f for which $\langle v, f\rangle \geq \alpha$. Writing $f = (f_1, ..., f_m)$ and $v = (v_1, ..., v_m)$, we may instead write this condition as $\sum_{i=1}^{m} \langle v_i, f_i\rangle \geq \alpha$. This structure allows one to find the relevant filters in time $\tilde{O}(\mathcal{F})$, where $\mathcal{F}$ is the number of relevant filters, as is normally done in LSF. We extend this algorithm to allow sampling one of the relevant filters in time $\tilde{O}(1)$.

4.2 Filter Sampling Overview

To describe how we sample one of the relevant filters, we first examine how one normally finds the set of all relevant filters in LSF. Broadly speaking, the relevant filters are a subset of the leaves in a certain tree (said to be *good leaves*). To find all of them, one explores the tree from the root downwards in a pruned enumeration routine. In comparison, we need to sample a uniformly random good leaf.

To sample a good leaf efficiently, we need some information about the structure of the tree. Namely, it suffices to know how many good leaves are there below each node - we could then perform a process in which we start from the root, and continually walk down to one of the children of the current node randomly with probabilities weighted by the number of good leaves below each child. This would allow sampling a uniformly random good leaf, but requires evaluating the number of good leaves below each node. Naively, this requires traversing the entire tree, which is costly. Fortunately, the tree and the subset of good nodes have a structure that allows evaluating this quantity using dynamic programming - albeit using a slightly larger set of good leaves. After sampling from this larger set, we use rejection sampling in order to sample a leaf from the original set of good leaves.

4.3 The Enumeration Tree

Consider the following tree: Its nodes at level $0 \leq k \leq m$ are labeled by vectors in $C_1 \times \cdots \times C_k$, and the parent of $(f_1, ..., f_{k-1}, f_k)$ is the direct prefix $(f_1, ..., f_{k-1})$. The leaves correspond to filters, and we fix a certain subset of the leaves $\mathcal{G} \subseteq C_1 \times \cdots \times C_m$ which are called *good leaves.* To begin with, the good leaves will be ones corresponding to α-close filters, but we will slightly alter this choice later.

For a node x, we define $\mathcal{L}(x)$ to be the number of good leaves below x. Given an oracle to $\mathcal{L}$, we may sample a random good leaf by starting at the root and iteratively walking to one of the children of the current node with probabilities weighted by the number of good leaves below each child. The following pseudo-code describes this process:

```
Leaf Sampling Scheme
(1) $x \leftarrow$ root
(2) For $i = 1, 2, ..., m$:
(3)     For $y \in$ Children($x$):
(4)         $p_y \leftarrow \frac{\mathcal{L}(y)}{\mathcal{L}(x)}$
(5)     Choose $y \in$ Children($x$) randomly with probability $p_y$
(6)     $x \leftarrow y$
(7) Return $x$
```

The values $\mathcal{L}(x)$ may be calculated recursively beforehand, starting from the leaves and going up (using $\mathcal{L}(x) = \sum_{y \in \text{Children}(x)} \mathcal{L}(y)$). However, this takes as much time as the number of nodes in the tree. In our case, the specific structure of the tree and subset of good leaves provides a more efficient way to evaluate $\mathcal{L}$.

Recall that a leaf $f = (f_1, ..., f_m)$ is good if $\langle v, f \rangle = \sum_{i=1}^{m} \langle v_i, f_i \rangle \geq \alpha$. The number of good leaves below a node $(f_1, ..., f_k)$ is the number of suffixes $(f_{k+1}, ..., f_m)$ for which

$$\sum_{i=k+1}^{m} \langle v_i, f_i \rangle \geq \alpha - \sum_{i=1}^{k} \langle v_i, f_i \rangle$$

Notice that this condition is similar for all nodes at level k: We define $\mathcal{N}_k(\gamma)$ to be the number of suffixes $(f_{k+1}, ..., f_m)$ for which $\sum_{i=k+1}^{m} \langle v_i, f_i \rangle \geq \gamma$. Given an oracle for calculating $\mathcal{N}_k(\gamma)$ we could easily calculate $\mathcal{L}(x)$ for each node x by the formula

$$\mathcal{L}\left((f_1, ..., f_k)\right) = \mathcal{N}_k\left(\alpha - \sum_{i=1}^{k} \langle v_i, f_i \rangle\right)$$

The values of $\mathcal{N}_k(\gamma)$ could also be calculated recursively, but there are infinitely many values of γ to consider, as it is a continuous parameter. Our solution to this problem is to slightly enlarge the set of good leaves, in a way that essentially limits and discretizes the set of possible values of γ.

4.4 Discretization of Good Leaf Condition

Fix an integer $R = \theta(m \cdot d)$. A filter $f = (f_1, ..., f_m)$ is called *pseudo-relevant* if

$$\sum_{i=1}^{m} \frac{\lceil R\langle v_i, f_i \rangle \rceil}{R} \geq \frac{\lfloor R\alpha \rfloor}{R}$$

This is the same as the previous definition of relevancy, except we round each summand to an integer multiple of $\frac{1}{R}$. If we change the subset of good leaves to correspond to pseudo-relevant filters instead of relevant filters, then the number of good leaves below a node $x = (f_1, ..., f_k)$, now denoted $\mathcal{L}^R(x)$, is the number of suffixes $(f_{k+1}, ..., f_m)$ for which $\sum_{i=k+1}^{m} \frac{\lceil R\langle v_i, f_i \rangle \rceil}{R} \geq \gamma$, where $\gamma = \frac{\lfloor R\alpha \rfloor - \sum_{i=1}^{k} \lceil R\langle v_i, f_i \rangle \rceil}{R}$ which is an integer multiple of $\frac{1}{R}$. We define $\mathcal{N}_k^R(\gamma)$ to be the number of such suffixes for general γ. Notice that

$$-1 \leq \frac{R\langle v_i, f_i \rangle}{R} \leq \sum_{i=k+1}^{m} \frac{\lceil R\langle v_i, f_i \rangle \rceil}{R} \leq \frac{R\langle v_i, f_i \rangle + m}{R} \leq 1 + \frac{m}{R}$$

Therefore, $\mathcal{N}_k^R(\gamma) = \mathcal{N}_k^R(1 + \frac{m}{R})$ for $\gamma > 1 + \frac{m}{R}$, and likewise $\mathcal{N}_k^R(\gamma) = \mathcal{N}_k^R(-1)$ for $\gamma < -1$. This means that we only need to check values $-1 \leq \gamma \leq 1 + \frac{m}{R}$ which are multiples of $\frac{1}{R}$, and there are $2R + m + 1 = O(R)$ such values. This allows efficiently calculating all values of $\mathcal{N}_k^R(\gamma)$ iteratively, starting from $k = m$ and going from k to $k-1$ by the formula

$$\mathcal{N}_{k-1}^R(\gamma) = \sum_{f_k \in C_k} \mathcal{N}_k^R\left(\gamma - \frac{\lceil R\langle v_k, f_k \rangle \rceil}{R}\right) \quad (*)$$

We can then calculate the value of $\mathcal{L}^R(x)$ for every node x via

$$\mathcal{L}^R\left((f_1, ..., f_k)\right) = \mathcal{N}_k^R\left(\frac{\lfloor R\alpha \rfloor - \sum_{i=1}^{k} \lceil R\langle v_i, f_i \rangle \rceil}{R}\right) \quad (**)$$

and sample a random good leaf as in the pseudo-code above, which translates to a random pseudo-relevant filter.

As we show in the next subsection, the set of pseudo-relevant filters contains the set of relevant (α-close) filters, and not much more. So after we sample a pseudo-relevant filter, we may check whether it is α-close to v, and otherwise discard it and sample a new one. This rejection sampling procedure samples a uniformly random relevant filter. Once we sample such a filter, we may simply choose a uniformly random vector from the corresponding bucket, which completes the construction of a circuit for sampling candidates.

4.5 Analysis

The complete algorithm for candidate sampling is as follows: We initially perform preprocessing in which we iteratively calculate all values of $\mathcal{N}_k^R(\gamma)$ for all $0 \leq k \leq m$ and $-1 \leq \gamma \leq 1 + \frac{m}{R}$ (where k and $R\gamma$ are integers). We start with $k = m$, setting $\mathcal{N}_m^R(\gamma) = 1$ if $\gamma \geq 0$ and $\mathcal{N}_m^R(\gamma) = 0$ if $\gamma < 0$. After calculating $\mathcal{N}_k^R(\gamma)$ for fixed k and all γ, we may calculate $\mathcal{N}_{k-1}^R(\gamma)$ via the recursive formula $(*)$. After the preprocessing is complete, we may sample a good leaf in the tree by starting from the root and traveling down as described in the pseudo code above, using the formula $(**)$ to evaluate the number of good leaves below a given node. We translate the good leaf sampled into a pseudo-relevant filter. If it is relevant, we find the corresponding bucket and sample a random vector from this bucket. If it is not relevant, we continue sampling good leaves until a relevant filter is found.

4.5.1 Preprocessing

At each level $0 \leq k \leq m$ we perform $O(R)$ summations of t' summands each, which takes $\tilde{O}(R \cdot t')$ time. Therefore the entire preprocessing routine takes $\tilde{O}(m \cdot R \cdot t') = \tilde{O}(m^2 \cdot d \cdot t')$ which is $2^{o(d)}$.

4.5.2 Sampling a Pseudo-Relevant Filter

At each step we iterate over t' children of the node x, evaluate the number of good leaves below each of them (which takes $\tilde{O}(1)$ time given the preprocessed data), and choose a child with probabilities weighted by these numbers. We do this by choosing a uniformly random integer $\ell \in \left[0, \mathcal{L}^R(x)\right)$, ordering the children as $y_1, ..., y_{t'}$, and choosing the child y_j if $\ell \in \left[\sum_{j'=1}^{j-1} \mathcal{L}^R(y_{j'}), \sum_{j'=1}^{j} \mathcal{L}^R(y_{j'})\right)$. The correct value of j can be found naively in time $O(t')$ by complete enumeration, or in time $O(\log t')$ by binary search. The latter option requires some more negligible preprocessing (namely, calculating $\sum_{j'=1}^{j-1} \mathcal{N}_k^R \left(\gamma - \frac{\left\lceil R\langle v_k, f_k^{(j')}\rangle \right\rceil}{R}\right)$ for all k, γ, j), but both options take subexponential time. The sampling routine takes $\tilde{O}(m \cdot t')$ or $\tilde{O}(m \cdot \log t')$ depending on whether or not binary search is used. In both cases the time complexity is $2^{o(d)}$, and if binary search is used then it is in fact poly(d).

4.5.3 Rejection Sampling

Observe that

$$f \text{ is } \alpha\text{-close} \implies f \text{ is pseudo-relevant} \implies f \text{ is } (\alpha - \varepsilon)\text{-close}$$

where $\varepsilon = \frac{m+1}{R} = \theta\left(\frac{1}{d}\right)$. This follows from the fact that

$$R \cdot (\langle v, f\rangle - \alpha) \leq \sum_{i=1}^{m} \left\lceil R\langle v_i, f_i^{(j)}\rangle \right\rceil - \lfloor R\alpha \rfloor \leq R \cdot (\langle v, f\rangle - \alpha) + m + 1$$

The left part means that the set of pseudo-relevant filters contains the set of relevant filters. The right part means that the proportion of relevant filters to pseudo-relevant filters is at least

$$\frac{\left(1-\alpha^2\right)^{d/2+o(d)}}{\left(1-(\alpha-\varepsilon)^2\right)^{d/2+o(d)}} = \frac{\left(1-\alpha^2\right)^{d/2+o(d)}}{\theta(1)\cdot\left(1-\alpha^2\right)^{d/2+o(d)}} = \theta(1)$$

so the rejection sampling only increases the time complexity by a constant factor.

To sum up, the preprocessing takes subexponential time, and each sample takes polynomial time. Therefore both have negligible effect on the time complexity.

5 Conclusion

In this paper we have proposed an algorithm which improves upon the quantum LSF-based sieving algorithm in [16] and reduces its time complexity from $2^{0.2653d+o(d)}$ to $2^{0.2571d+o(d)}$. While this is slightly more than the $2^{0.2570d+o(d)}$ given by the quantum random walk based algorithm in [7], future work may combine the two techniques into an algorithm with lower time complexity than both.

These improvements have a direct implication on the security of lattice-based cryptography. Our results, as well as the results in [7], lower the Core-SVP strength of all lattice-based schemes by $\approx 3\%$, and combining them may even result in further security losses. Given that some lattice-based finalists in Round 3 of NIST's PQC process already have a core-SVP strength well below the original target of 128 bits, these results should be taken into consideration and further research must be done in order to fully understand their implications. Future work might further decrease the security below acceptable levels.

References

[1] Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In STOC, pages 601–610, 2001.

[2] Albrecht, M. R., Ducas, L., Herold, G., Kirshanova, E., Postlethwaite, E. W., Stevens, M.: The General Sieve Kernel and New Records in Lattice Reduction. In SAC, pages 29–47, 2019.

[3] Albrecht, M. R., Gheorghiu, V., Postlethwaite, E. W., Schnack, J. M.: Estimating quantum speedups for lattice sieves. In Advances in Cryptology – ASIACRYPT 2020, pages 583–613, Cham, 2020. Springer International Publishing.

[4] Bai, S., Miller, S., Wen, W.: A refined analysis of the cost for solving LWE via uSVP. In AFRICACRYPT 19, volume 11627 of LNCS, pages 181–205, 2019.

[5] Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. In SODA, 2016.

[6] Becker, A., Gama, N., Joux, A.: A sieve algorithm based on overlattices. In ANTS, pages 49–70, 2014.

[7] Chailloux, A., Loyer, J.: Lattice sieving via quantum random walks. In Cryptology ePrint Archive, Report 2021/570, pages 1–29, 2021.

[8] Doulgerakis, E., Laarhoven, T., de Weger, B.: Sieve, Enumerate, Slice, and Lift: Hybrid Lattice Algorithms for SVP via CVPP. In Cryptology ePrint Archive, Report 2020/487, pages 1–20, 2020.

[9] Ducas, L.: Shortest Vector from Lattice Sieving: a Few Dimensions for Free. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 125–145, 2018.

[10] Ducas, L., Stevens, M., van Woerden, W.: Advanced Lattice Sieving on GPUs, with Tensor Cores. In Cryptology ePrint Archive, Report 2021/141, pages 1–38, 2021.

[11] Fincke, U., Pohst, M.: Improved methods for calculating vectors of short length in a lattice. In Mathematics of Computation 44(170), pages 463–471, 1985.

[12] Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In STOC, pages 604–613, 1998.

[13] Kannan, R.: Improved algorithms for integer programming and related lattice problems. In STOC, pages 193–206, 1983.

[14] Kirshanova, E., Mårtensson, E., Postlethwaite, E. W., Moulik, S. R.: Quantum Algorithms for the Approximate k-List Problem and their Application to Lattice Sieving. In ASIACRYPT, 2019.

[15] Laarhoven, T.: Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In CRYPTO, pages 3–22, 2014.

[16] Laarhoven, T.: Search problems in cryptography, From fingerprinting to lattice sieving. PhD thesis, Eindhoven University of Technology, 2016.

[17] Laarhoven, T., Mariano, A.: Progressive Lattice Sieving. In International Conference on Post-Quantum Cryptography, pages 292–311, 2018.

[18] Laarhoven, T., Mosca, M., van de Pol, J.: Finding shortest lattice vectors faster using quantum search. In Designs, Codes and Cryptography, 77(2), pages 375–400, 2015.

[19] Laarhoven, T., de Weger, B.: Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In LATINCRYPT, pages 101–118, 2015.

[20] Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In CT-RSA, pages 319–339, 2011.

[21] Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In SODA, pages 1468–1480, 2010.

[22] Nguyen, P. Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. In J. Math. Crypt. 2(2), pages 181–207, 2008.

[23] Pohst, M. E.: On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. In ACM SIGSAM Bulletin 15(1), pages 37–44, 1981.

[24] Postlethwaite, E. W., Virdia, F.: On the Success Probability of Solving Unique SVP via BKZ. In Cryptology ePrint Archive, Report 2020/1308, pages 1–35, 2020.

[25] Pujol, X., Stehlé, D.: Solving the shortest lattice vector problem in time $2^{2.465n}$. In Cryptology ePrint Archive, Report 2009/605, pages 1-7, 2009.

[26] Schneider, M.: Sieving for short vectors in ideal lattices. In AFRICACRYPT, pages. 375–391, 2013.

[27] Schneider, M., Gama, N., Baumann, P., Nobach, L.: SVP Challenge. Online at https://latticechallenge.org/svp-challenge/ .

[28] van de Pol, J., Smart, N.: Estimating key sizes for high dimensional lattice-based systems. In IMACC, pages 290–303, 2013.

[29] Wang, X., Liu, M., Tian, C., Bi, J.: Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In ASIACCS, pages 1–9, 2011.

[30] Zhang, F., Pan, Y., Hu, G.: A three-level sieve algorithm for the shortest vector problem. In SAC, pages 29–47, 2013.