Estimating the Hidden Overheads in the BDGL Lattice Sieving Algorithm

Léo Ducas[1,2]

[1] Cryptology Group, CWI, Amsterdam, The Netherlands
[2] Mathematical Institute, Leiden University, The Netherlands

Abstract. The lattice sieving algorithm based on list-decoding of Becker-Ducas-Gama-Laarhoven (SODA 2016) is currently at the center of cryptanalysis cost estimates of candidate lattice schemes for post-quantum standardization.
Yet, only an idealized version of this algorithm has been carefully modelled, i.e. given an efficient list-decoding oracle for a perfectly random spherical code. In this work, we propose an experimental analysis of the actual algorithm. The difficulty lies in estimating the probabilistic defect with respect to perfectly random spherical codes for the task at hand. While it should be in principle infeasible to run the algorithm in cryptographically relevant dimensions, a few tricks allow to nevertheless measure experimentally the relevant quantity.
Concretely, we conclude on an overhead factor of about 2^6 on the number of gates in the RAM model compared to the idealized model for dimensions around 380 after an appropriate re-parametrization. Part of this overhead can be traded for extra memory, at a costly rate. We also clarify that these overheads apply to an internal routine, and discuss how they can be partially mitigated in the whole attack.

Keywords: Concrete Cryptanalysis, Lattice, Sieving.

1 Introduction

1.1 Context

Sieving refers to a class of algorithm for finding the shortest vector in a Euclidean lattice; it proceeds by continuously searching within a list L of lattice vectors for pairs $\mathbf{u}, \mathbf{v} \in L$ such that $\mathbf{u} - \mathbf{v}$ is shorter than either of the original vectors. Assuming that all the vectors have roughly the same length (say, length 1), this is equivalent to searching for reducing pairs of vectors, i.e. pairs with angle less than $\pi/3$.

While proving that sieving does indeed succeed requires convoluted and costly tricks [AKS01], such a simple algorithm works well in practice [NV08] when working with a list of size roughly $N \approx (4/3)^{n/2}$ in dimension n. A naive implementation of this strategy therefore leads to finding the shortest vector in time roughly $n^{O(1)} \cdot N^2$, for a complexity of $2^{.415n+o(n)}$. A line of work initiated by

Laarhoven [Laa15a,Laa15b,LW15] has led to lower complexity, by the introduction of the Near Neighbour Search formalism (NNS), using Locality Sensitive Hashing. This approach allows one to find (most of) the reducing pairs in a time L^c for some constant $c \in [1, 2]$.

Among many variants [Laa15b,LW15,BGJ15,Laa15a], the asymptotically fastest is that of Becker-Ducas-Gama-Laarhoven [BDGL16], with a time complexity of $(3/2)^{n/2+o(n)} = 2^{.292n+o(n)}$. It is based on efficient list-decoding of well chosen spherical codes. It also underlies the current fastest implementation on CPUs [ADH+19,DSvW21], though the cross-over point with the simpler sieve of [BGJ15] has not yet been reached on GPUs [DSvW21].

This algorithm has also been the object of precise gate cost estimation in large dimensions [AGPS20], or more specifically its internal near-neighbors search (NNS) routine. These estimates are used in the documentation of several NIST post-quantum standardization candidates. In particular, the Kyber documentation [ABD+21, Sec. 5.3] gives a list of eight open question about various approximations and foreseeable improvements that may affect these estimates, both upward and downward.

1.2 This work

This work aims to resolve the open question Q2 of [ABD+21, Sec. 5.3] regarding the idealized model for the near neighbors search procedure of [BDGL16]. In the idealized model, the spherical code is assumed to be perfectly random, and to be efficiently list-decodable. The instantiated NNS procedure instead resorts to a product of random codes, which induces overheads. More specifically, there is a trade-off between three overheads when instantiating the [BDGL16] framework with product codes:

- a computation overhead $\mathsf{CO}^\times$, to list-decode the spherical code,
- a memory overhead $\mathsf{MO}^\times$, to store pointers to vectors in buckets,
- a probability overhead $\mathsf{PO}^\times$, accounting for the randomness defect of product codes

The computation and probability overhead both contribute to the gate count of the algorithm, defining a time overhead $\mathsf{TO}^\times = \mathsf{CO}^\times \cdot \mathsf{PO}^\times$. It is known from [BDGL16] that all three overheads can be made subexponential, and that either the memory overhead or the computation overhead can be made negligible.

While the computation overhead $\mathsf{CO}^\times$ and memory overhead $\mathsf{MO}^\times$ are easy to calculate concretely, the probability overhead $\mathsf{PO}^\times$ seems harder to model precisely and rigorously, and naive measurements would not be feasible for cryptographically relevant parameters. The main technical contribution of this work resides in the design of feasible experiments to measure this overhead (Section 3).

Implementing this experiment leads to concrete conclusions on these overheads and their trade-off (Section 4). In dimension $n = 384$ (roughly what is needed to break lattice candidates at level NIST 1) the idealized NNS procedure is costed by [AGPS20] at $2^{134.1}$ gates and $2^{97.6}$ bits of memory. We conclude on a 2^6 slowdown factor on time for small memory overhead. A partial trade-off is possible, but costly, and even with a factor 2^{12} increase in memory consumption, a slowdown factor of $2^{2.5}$ remains.

We also discuss how these overheads of the internal NNS routine can be somewhat mitigated inside a complete lattice attacks (Section 5), and propose further open problems (Section 6) .

Source code. The artifact associated with this work are available at

`https://github.com/lducas/BDGL_overhead` .

Acknowledgments

The author wishes to thank Martin Albrecht, Vlad Gheorghiu, and Eamonn Postlethwaite, John Schanck for open-sourcing and maintaining various research artefacts making this research possible. The author was supported by the ERC-StG-ARTICULATE project (no. 947821).

2 Preliminaries

Complexity. Time and memory complexity are given in the RAM model, and for readability are given in terms of elementary operations during most of this paper; constant factors will be gracefully ignored. Only in Section 4.3 do we quantify costs more precisely in terms of binary gates.

Independent Small Probability Events. We will silently abuse the approximation $1-(1-\mathcal{W})^M \approx \mathcal{W}\cdot M$ to our best convenience.

Euclidean Vector Space. In all this work, bold lowercase $(\mathbf{u},\mathbf{v},\mathbf{w},\dots)$ letters denotes row vectors of the real vector space $\mathbb{R}^n$ endowed with its canonical Euclidean inner product: $\langle \mathbf{x},\mathbf{y}\rangle := \sum x_i y_i$, and associated Euclidean metric $\|\mathbf{x}\| = \sqrt{\langle \mathbf{x},\mathbf{x}\rangle}$.

Spheres, Caps and Wedges. We define the following bodies in dimension $\mathbb{R}^n$:

– The unit sphere : $\mathcal{S}^n := \{\mathbf{x}\in\mathbb{R}^n \mid \|x\| = 1\}$
– The halfspace : $\mathcal{H}^n_{\mathbf{v},a} := \{\mathbf{x}\in\mathbb{R}^n \mid \langle \mathbf{x},\mathbf{v}\rangle \geq a\}$
– The spherical cap : $\mathcal{C}^n_{\mathbf{v},a} := \mathcal{S}^n \cap \mathcal{H}_{\mathbf{v},a}$
– The (symmetric) spherical wedge:[3] $\mathcal{W}^n_{\mathbf{v},\mathbf{w},a} := \mathcal{C}_{\mathbf{v},a}\cap\mathcal{C}_{\mathbf{w},a}$

where $\mathbf{v},\mathbf{w}\in\mathcal{S}^n$ and $a\in[0,1]$. Furthermore, we define the relative volume of caps and wedges as follows.

[3] The literature usually defines asymmetric spherical wedge with two different bounds a, b for the halfspace in directions $\mathbf{v}$ and $\mathbf{w}$. However the best choice appears to be $a = b$ [BDGL16,AGPS20].

- $\mathcal{C}^n(a) := \mathrm{Vol}(\mathcal{C}_{\mathbf{v},a}) / \mathrm{Vol}(\mathcal{S}^n)$ for any $\mathbf{v} \in \mathcal{S}^n$
- $\mathcal{W}^n(a,c) := \mathrm{Vol}(\mathcal{W}_{\mathbf{v},\mathbf{w},a}) / \mathrm{Vol}(\mathcal{S}^n)$ for any $\mathbf{v}, \mathbf{w} \in \mathcal{S}^n$ such that $\langle \mathbf{v}, \mathbf{w} \rangle = c$.

The dimension n being generally clear from context, the exponent n might be omitted in the rest of this document. For asymptotic analysis, we have the following lemma.

Lemma 1 ([BDGL16]). *For any fixed $a \in [0,1]$ and growing n,*

- $\mathcal{C}^n(a) = \left(1 - a^2\right)^{n/2} \cdot n^{O(1)}$
- $\mathcal{W}^n(a,c) = \left(1 - \frac{2a^2}{1+c}\right)^{n/2} \cdot n^{O(1)}$.

These quantities $\mathcal{C}^n(a)$ and $\mathcal{W}^n(a,c)$ can be efficiently computed precisely [AGPS20]; in particular $\mathcal{C}(a)$ directly relates to the incomplete beta function.

2.1 List-Decoding Sieve, Idealized

The idealized version of [BDGL16] proceeds by assuming that one is given random yet efficiently list decodable spherical code $F \subset \mathcal{S}^n$ of size M. More precisely, it is assumed that one can compute the set of codewords falling in a given spherical cap $F(\mathbf{v}, a) := F \cap \mathcal{C}_{\mathbf{v},a}$, and this efficiently, that is in time, say, $\#F(\mathbf{v}, a) + \mathsf{LDO}^+$ where $\mathsf{LDO}^+ = 2^{o(n)}$ denotes a sub-exponential additive overhead for list decoding. In the idealized model, this factor is assumed to be $\mathsf{LDO}^+ = 1$.

Given such a set and such an oracle, one proceeds with the search for reducing pairs as follows:

1. Compute $F(\mathbf{v}, a)$ for each $\mathbf{v} \in L$, and store $\mathbf{v}$ in buckets labelled by each $\mathbf{f} \in F(\mathbf{v}, a)$
2. For each $\mathbf{f} \in F$, and for each pair $\mathbf{v}, \mathbf{w}$ in the bucket labelled by $\mathbf{f}$, check whether $\langle \mathbf{v}, \mathbf{w} \rangle \leq 1/2$.

Assuming that each codeword $\mathbf{f} \in F$ and each lattice vector of the list $\mathbf{v} \in L$ is uniform and independent over the sphere, one expects each $\mathbf{v}$ to fall in $M \cdot \mathcal{C}(a)$ buckets, and each bucket to contain $N \cdot \mathcal{C}(a)$ vectors, leading to a time complexity of

$$N \cdot (n \cdot M \cdot \mathcal{C}(a) + \mathsf{LDO}^+) + n \cdot M \cdot (N \cdot \mathcal{C}(a))^2$$
$$\approx 2nM \quad \text{when } a = 1/2$$

and a memory complexity of $N \cdot M \cdot \mathcal{C}(a)$ for the procedure.

A reducing pair will be detected if and only if there is a codeword falling in the wedge $\mathcal{W}_{\mathbf{v},\mathbf{w},a} = \mathcal{C}_{\mathbf{v},a} \cap \mathcal{C}_{\mathbf{w},a}$. Given that $\mathbf{f} \in F$ are uniform and independent, this happens with probability $1 - (1 - \mathcal{W}(a, 1/2))^M \approx M \cdot \mathcal{W}(a, 1/2)$. One should therefore choose $M \approx 1/\mathcal{W}(a, 1/2)$ to find essentially all reducing pairs.

Recalling that $N \approx 1/\mathcal{C}(1/2) = (4/3)^{n/2} \cdot n^{O(1)}$, one finds that the optimal asymptotic time complexity of $\mathsf{T} = 1/\mathcal{W}(1/2, 1/2) = (3/2)^{n/2} \cdot n^{O(1)}$ is reached at $a = 1/2$, and with a similar memory complexity $\mathsf{M} = (3/2)^{n/2} \cdot n^{O(1)}$. In practice [BDGL16], a slightly smaller value of $a < 1/2$ seems preferable. However we note that after the update of [AGPS20] including the model correction of [MAT22] for BDGL cost, a slightly larger $a > 1/2$ now gives optimal gate count.

Low Memory Variant. Following an original remark of [BGJ15], [BDGL16] also propose a variant where the memory cost is dominated by that of the list of vectors, rather than by the buckets. One can simply choose a smaller value of $M = 1/\mathcal{C}(a)$, and to repeat the whole procedure with a fresh spherical code $R = M/\mathcal{W}(a, 1/2) = (9/8)^{n/2} \cdot n^{O(1)}$ many times. The new time complexity is now

$$RN \cdot (nM \cdot \mathcal{C}(a) + \mathsf{LDO}^+) + nRM \cdot (N \cdot \mathcal{C}(a))^2$$
$$\approx nRM(2 + \mathsf{LDO}^+) \quad \text{when } a = 1/2$$

which is similar to the above, up to an extra sub-exponential factor LDO^+. That is, we have traded an exponential factor $(9/8)^{n/2} \cdot n^{O(1)}$ on memory for a sub-exponential factor on time. Intermediate choices are also possible, ranging from $M_{\min} = 1/\mathcal{C}(a)$ to $M_{\max} = 1/\mathcal{W}(a, 1/2)$.

2.2 List-Decoding Sieve, Instantiated

It remains to replace the random spherical code by one that is structured enough to allow efficient list-decoding, while not affecting the success probability of detecting reducing pairs too much. This is an issue of independence. Indeed, consider for a second a code of size $M = 1/\mathcal{W}(a, 1/2)$, whose codewords would all be concentrated in a small region. The average number of codewords in a random wedge $\mathcal{W}^n_{\mathbf{v},\mathbf{w},a}$ would still be 1, yet most of the time a wedge will contain no codewords at all, while the remaining rare case is a wedge containing almost all of the M codewords. The desired situation is one there is always about 1 codeword in such a random wedge.

To do so, it is proposed in [BDGL16] to use a product of random codes in smaller dimensions. That is, F is constructed as the Cartesian product of m random spherical codes in dimension n/m, each of size $B = M^{1/m}$. For such a code, a decoding algorithm is devised [BDGL16, Sec. 5], running in time essentially

$$nB + mB \log B + m \cdot \#F(\mathbf{v}, a).$$

We will not describe the algorithm in detail (see [BDGL16, Sec. 5]), but briefly explain the three terms:

1. nB corresponds to the cost of computing B inner products in dimension n/m for each of the m subcode.
2. $mB \log B$ corresponds to sorting the m lists of inner products.
3. $m \cdot \#F(\mathbf{v}, a)$ corresponds to a tree enumeration without any backtracking, in a tree of depth m with $\#F(\mathbf{v}, a)$ leaves.

In practice, these costs can be tackled further [MLB17,DSvW21,MAT22], as will be discussed in Section 4.3, and a more optimistic model would be

$$mB + \#F(\mathbf{v}, a).$$

For now, one may simply consider that the additive overhead of the list decoder is $\mathsf{LDO}^{+} = nB = nM^{1/m}$. Furthermore, it is proved [BDGL16, Theorem 5.1] that such random product codes are not that far off perfectly random codes when $m = \log n$; more precisely, the success probability for detecting a reducing pair is only a sub-exponential factor $\mathsf{PO}^{\times} = 2^{\tilde{O}(\sqrt{n})}$ away of the idealized model:

$$\mathbb{P}_W[\#(F \cap W) \geq 1] = M \cdot \mathcal{W}(a, c)/\mathsf{PO}^{\times}$$

where W is a random wedge with parameter (a, c). This multiplicative probability overhead must be compensated for by repeating the algorithm $\mathsf{PO}^{\times}$ many times.

Low Memory Variant. For the low memory variant, one can ensure independence across the R repetitions by applying a fresh random rotation to the input of each repetition. This ensures that the overall probability overhead is the same as the individual ones.

3 Analyzing the List-Decoding Sieve Instantiation

3.1 Overheads and Trade-offs

We have identified three overheads between the idealized and the instantiated list-decoding sieving algorithm [BDGL16]: a cost overhead on the procedure $\mathsf{CO}^{\times}$ induced by the non-trivial cost of list-decoding, a memory overhead $\mathsf{MO}^{\times}$ for storing pointers to vectors in buckets, and a probabilistic overhead $\mathsf{PO}^{\times}$ induced by the independence defect of random product codes. The overall time overhead is given by $\mathsf{TO}^{\times} = \mathsf{CO}^{\times} \cdot \mathsf{PO}^{\times}$, and one may also consider the time-memory overhead $\mathsf{TMO}^{\times} = \mathsf{TO}^{\times} \cdot \mathsf{MO}^{\times}$.

The first two overheads $\mathsf{CO}^{\times}$ and $\mathsf{MO}^{\times}$ can be calculated from the algorithm parameter, though the exact formula might be bulky and hard to parse. For illustration, in a simple model ignoring constants, and assuming $a = 1/2$ and $N = 1/\mathcal{C}(1/2)$, we have:

$$\mathsf{TO}^{\times} \approx 1 + \frac{\mathsf{LDO}^{+}}{nM\mathcal{C}(a)} \approx 1 + \frac{M_{\min}}{M} \cdot M^{1/m} \text{ and } \mathsf{MO}^{\times} \approx 1 + \frac{M}{M_{\min}}.$$

The overhead $\mathsf{PO}^{\times}$ is however more problematic, and the author admits to having no clue on how to approach it analytically. In this position, one would be tempted to just ignore $\mathsf{PO}^{\times}$, and focus on the above; however such an analysis would result in essentially the same result as the idealized model: setting $M = M_{\min}$ and $m = \log_2(M) = \Theta(n)$ gives constants overheads $\mathsf{TO}^{\times}$ and

$\mathsf{MO}^\times$. In such an extreme regime, the BDGL algorithm starts resembling the hyperplane-LSH algorithm of Laarhoven [Laa15b], whose complexity is supposed to be exponentially worse, that is, we'd expect $\mathsf{PO}^\times = 2^{\Theta(n)}$.

In conclusion, to refine the cost analysis of [BDGL16] one has no choice but to estimate $\mathsf{PO}^\times$ some way or another. Before we explore how to experimentally measure such a quantity, let us briefly recapitulate how the parameters affect each overhead:

1. The parameter M can range from $M_{\min} = 1/\mathcal{C}(a)$ to $M_{\max} = 1/\mathcal{W}(a, 1/2)$. Straightforwardly, increasing M decreases memory overhead and increases time overhead. One may also guess that $\mathsf{PO}^\times$ grows with M; indeed, a larger value of $R = M_{\max}/M$ improves independence of the success events bringing us closer to the idealized model. This trend is confirmed by the experiments of Section 4.2.
2. The parameter m is a positive integer, and time overhead decrease with it; however we expect $\mathsf{PO}^\times$ to grow with m. This will also be confirmed by experiments of Section 4.2.
3. The parameter a may also affect both the base-line performance and the probabilistic overhead $\mathsf{PO}^\times$. Though it is not clear a-priori in which direction. Experiments of section 4.2 will show that for a fixed M, $\mathsf{PO}^\times$ increases as a decrease.

3.2 Measuring $\mathsf{PO}^\times$, Naively

In this section, we discuss how to experimentally measure $\mathsf{PO}^\times$, hopefully up to cryptographically relevant dimension. By definition, running the full sieve algorithm is not an option. When giving explicit complexity, we tacitly assume $a = 1/2$.

The Naive Approach. The naive approach consists in generating random reducing pair on the sphere $\mathbf{v}, \mathbf{w}$ such that $\langle \mathbf{v}, \mathbf{w} \rangle = 1/2$, and simply testing whether $F(\mathbf{v}, a) \cap F(\mathbf{w}, a)$ is non-empty.

Lemma 2. *There is a polynomial time algorithm that, given $a \in [0, 1]$ samples a uniform pair $\mathbf{v}, \mathbf{w} \in \mathcal{S}^n$ conditioned on $\langle \mathbf{v}, \mathbf{w} \rangle = a$.*

Proof. The algorithm follows:

1. Sample $\mathbf{v}$ uniformly on $\mathcal{S}$
2. Sample $\mathbf{x}$ uniformly on $\mathcal{S} \cap \mathbf{v}^\perp$, by sampling uniformly on $\mathcal{S}$, projecting orthogonally to $\mathbf{v}$, and renormalizing.
3. Set $\mathbf{w} = \sqrt{1 - a^2} \cdot \mathbf{x} + a \cdot \mathbf{v}$. □

Testing whether this pair is detected costs time $\mathsf{LDO}^+ + M/M_{\min}$ per trial, and memory $M/M_{\min}$. However, the success probability is only $M/(M_{\max}\mathsf{PO}^\times)$, so the experiments must be repeated$(M_{\max}\mathsf{PO}^\times)/M$ leading to a complexity of $(M_{\max}\mathsf{PO}^\times)/M_{\min} = (9/8)^{n/2+o(n)} = 2^{.085n+o(n)}$ to get a single success. And we may want to record up to a 1000 success for a decent estimate of the success probability.

3.3 Measuring $\mathsf{PO}^{\times}$, a First Speed-up

Because in this experimental set up we know in advance the reducing pair $\mathbf{v}, \mathbf{w}$ that the list-decoding is searching for, we can use this information to narrow down the search. In particular, consider the following lemma.

Lemma 3. *For any $a, c \in [0,1]$, and $\mathbf{v}, \mathbf{w} \in \mathcal{S}^n$ such that $\langle \mathbf{v}, \mathbf{w} \rangle = c$, we have the inclusion*

$$\mathcal{W}^n_{\mathbf{v},\mathbf{w},a} \subset \mathcal{C}^n_{\mathbf{z},2\cdot a/\sqrt{2+2c}}$$

where $\mathbf{z} = \frac{\mathbf{v}+\mathbf{w}}{\|\mathbf{v}+\mathbf{w}\|}$ is the midpoint of $\mathbf{v}, \mathbf{w}$ on the sphere $\mathcal{S}^n$.

Proof. Let $\mathbf{x}$ be in the wedge $\mathcal{W}^n_{\mathbf{v},\mathbf{w},a}$; by definition we have $\langle \mathbf{v}, \mathbf{x} \rangle \geq a$ and $\langle \mathbf{w}, \mathbf{x} \rangle \geq a$. Thus, it holds that $\langle \mathbf{v}+\mathbf{w}, \mathbf{x} \rangle \geq 2 \cdot a$, or equivalently that $\langle \mathbf{z}, \mathbf{x} \rangle \geq 2 \cdot a / \|\mathbf{v}+\mathbf{w}\|$. We conclude noting that $\|\mathbf{v}+\mathbf{w}\|^2 = \|\mathbf{v}\|^2 + \|\mathbf{w}\|^2 + 2\langle \mathbf{v}, \mathbf{w} \rangle = 2+2c$. $\square$

Further, we note that this inclusion $\mathcal{C}^n_{\mathbf{z},2\cdot a/\sqrt{2+2c}} \supset \mathcal{W}^n_{\mathbf{v},\mathbf{w},a}$ is rather a good over-approximation: the ratio $\mathsf{CW}(a,c) := \mathcal{C}(2a/\sqrt{2+2c})/\mathcal{W}(a,c)$ is not too large.

Lemma 4 ([BDGL16, App. A]). *For any $a, c \in [0,1)$, $\mathsf{CW}(a,c) = O(\sqrt{n})$.*

In our case, this implies that that $F(\mathbf{v},a) \cap F(\mathbf{w},a)$ is included in $F(\mathbf{z}, 2a/\sqrt{3})$; one can now test for each $\mathbf{f} \in F(\mathbf{z}, 2a/\sqrt{3})$ whether $\langle \mathbf{f}, \mathbf{v} \rangle \leq a$ and $\langle \mathbf{f}, \mathbf{w} \rangle \leq a$. This gives significant savings when $M/M_{\min} > \mathsf{LDO}^{+}$; in particular for $M = M_{\max}$ the time and memory complexity drop down to sub-exponential $(\mathsf{LDO}^{+} + \mathsf{CW}) \cdot \mathsf{PO}^{\times} = 2^{o(n)}$ per successful sample.

3.4 Measuring $\mathsf{PO}^{\times}$, a Second Speed-up

While the previous speed-up is appreciable in the high-memory regime, it is not very effective in the low memory regime, the main issue being that it inherently takes $(M_{\max}\mathsf{PO}^{\times})/M$ trials to get a success. And we are indeed most interested in the case where M is close to $M_{\min}$. To tackle it, we need not improve the algorithm, but instead design a different experiment.

Consider the following distribution $\mathcal{D} : \#(F(\mathbf{v},a) \cap F(\mathbf{w},a))$ where $\mathbf{v}, \mathbf{w}$ are uniform over the sphere conditioned on $\langle \mathbf{v}, \mathbf{w} \rangle = 1/2$. In other word, the distribution of the size of $W \cap F$ for a random $(a, 1/2)$-wedge. Let us call $d_i := \mathbb{P}_{j \leftarrow \mathcal{D}}[j = i]$ its density at i.

We know that the average size of $W \cap F$ is $M \cdot \mathcal{W}(a, 1/2)$, that is $\sum_{i \geq 0} d_i \cdot i = M \cdot \mathcal{W}(a, 1/2)$ is essentially equal to the success probability of when F is perfectly random. We are interested in the probability of success S, that $i \geq 1$ for $i \leftarrow \mathcal{D}$, i.e. $S = \sum_{i \geq 1} d_i$. An idea would be to design an experiment that focuses on the cases $i \geq 1$, *i.e.* an experiment that is conditioned on successful detection.

Conditioned sampling We start with a sampling procedure for generating pairs that are successfully detected by a given filter $\mathbf{f}$.

Lemma 5. *There is a polynomial time algorithm that, given $a, c \in [0,1]$ and $\mathbf{f} \in \mathcal{S}^n$ samples a uniform pair $\mathbf{v}, \mathbf{w} \in \mathcal{S}^n$ conditioned on $\langle \mathbf{v}, \mathbf{w} \rangle = c$ and $\mathcal{W}_{\mathbf{v},\mathbf{w},a} \ni \mathbf{f}$.*

Proof. Setting $\mathbf{z} = \frac{\mathbf{v}+\mathbf{w}}{\|\mathbf{v}+\mathbf{w}\|}$, we know by the previous lemma that $\mathbf{f} \in \mathcal{W}_{\mathbf{v},\mathbf{w},a}$ implies $\mathbf{f} \in \mathcal{C}_{\mathbf{z},2\cdot a/\sqrt{2+2c}}$, which is equivalent to $\mathbf{z} \in \mathcal{C}_{\mathbf{f},2\cdot a/\sqrt{2+2c}}$. Our strategy is therefore to

1. Sample $\mathbf{z}$ uniformly in $\mathcal{C}_{\mathbf{f},2\cdot a/\sqrt{2+2c}}$
2. Sample $\mathbf{v}, \mathbf{w}$ such that $\mathbf{z}$ is their midpoint, and $\langle \mathbf{v}, \mathbf{w} \rangle = c$
3. Return $(\mathbf{v}, \mathbf{w})$ if $\mathbf{f} \in \mathcal{W}_{\mathbf{v},\mathbf{w},a}$, otherwise restart.

For the first step, note that $r := \langle \mathbf{f}, \mathbf{z} \rangle$ is not determined, but constrained to $r \in [b := \frac{2\cdot a}{\sqrt{2+2c}}, 1]$. Defining $\beta = \cos^{-1}(b)$, $\rho = \cos^{-1}(r)$, we sample ρ uniformly in $[0, \beta]$ and use rejection sampling with acceptance probability $(\sin(\rho)/\sin(\beta))^{n-2}$. Finally, we choose $\mathbf{z} \in \mathcal{S}$ under the constraint $\langle \mathbf{f}, \mathbf{z} \rangle = r$.

The second step is easy, by first choosing $\mathbf{v}$ conditioned an inner product of $\frac{1+c}{\sqrt{2+2c}}$ with $\mathbf{z}$, and then setting $\mathbf{w}$ to be its reflection against the axis $\mathbf{z}$.

Regarding last step, we note that the accepting probability is $1/\mathsf{CW}(a,c) = 1/O(\sqrt{n})$ by Lemma 4. $\square$

An auxiliary distribution. We can now consider the following distribution $\mathcal{D}'$ of $\#(F(\mathbf{v},a) \cap F(\mathbf{w},a))$, where $\mathbf{f}$ is chosen uniformly from F, and $\mathbf{v}, \mathbf{w}$ chosen uniformly conditioned on $\langle \mathbf{v}, \mathbf{w} \rangle = 1/2$ and on $\mathcal{W}_{\mathbf{v},\mathbf{w},a} \ni \mathbf{f}$. By construction, it always holds that $i \geq 1$ for $i \leftarrow \mathcal{D}'$. In fact, the density at i is proportional to id_i, because there are i many ways to get to that same pair $(\mathbf{v}, \mathbf{w})$; that is, the density of $\mathcal{D}'$ is given by $d'_i = id_i / \sum_j jd_j$.

Conclusion. Now consider the expectation of $1/i$ for $i \leftarrow \mathcal{D}'$:

$$\mathbb{E}[1/i] = \frac{\sum_{i\geq 1} id_i/i}{\sum_j jd_j} = \frac{\sum_{i \geq 1} d_i}{\sum_j jd_j} = \frac{S}{M \cdot \mathcal{W}(a, 1/2)} = \frac{1}{\mathsf{PO}^{\times}}.$$

This means we can estimate $\mathsf{PO}^{\times}$ simply as the average of $\mathbb{E}[1/i]$ where $i \leftarrow \mathcal{D}'$.

The remaining question is how many sample do we need to get a precise estimate ? The variance of the empirical average grows as $\Theta(\mathbb{V}/k)$ using k samples and where $\mathbb{V}$ denotes the variance of individual samples; drawing $k = \Theta(\mathbb{V}/\mathbb{E}^2)$ samples one therefore reaches a relative error of $\sqrt{\mathbb{V}}/\mathbb{E} = O(1)$. Because the $1/i$ is supported by $[0,1]$, it holds that $\mathbb{V} \leq \mathbb{E}$, and we get $k \leq \Theta(1/\mathbb{E}) = O(\mathsf{PO}^{\times})$.

We therefore only need a sub-exponential amount $O(\mathsf{PO}^{\times})$ of sample in any regime, to be compared to $(M_{\max}\mathsf{PO}^{\times})/M$, a quantity as large as $(9/8)^{n/2+o(n)}$ in the low memory regime.

4 Implementation and Experiments

We implemented BDGL list-decoder in `python`, with the library `numpy` for vector operation, which makes most of the operation reasonably fast. The experiments to measure $\mathsf{PO}^\times$ are proudly parallel[4] over the many samples they require.

Only the tree enumeration is implemented without `numpy` acceleration, however this is reasonably mitigated by our first speed-up: the tree should be small on average for the regime we are interested in. We nevertheless experienced that a few instances had unreasonably large tree (enough to fill gigabytes worth of leaves); we therefore implemented a cap on the number of leaves at which the tree enumeration is halted, with a default value of 10^7. This might lead to under-estimating $\mathsf{PO}^\times$ in the experiments of Section 3.4, but no significantly unless $\mathsf{PO}^\times$ itself reaches similar order of magnitude. In particular, this is not significant in the regime of our experiments.

We also depend on the software of [AGPS20] for computing $\mathcal{C}$ and $\mathcal{W}$ exactly, as well as for computing baseline cost, that is the cost of idealized near neighboor search.

4.1 Consistency checks

In `consistency_check.py`, we implement some consistency checks for both speed-ups described in Section 3.3 and 3.4. For the first speed-up, we check that it is indeed the case that $F(\mathbf{v}, a) \cap F(\mathbf{w}, a)$ is included in $F(\mathbf{z}, 2a/\sqrt{3})$ where $\mathbf{z}$ is the spherical midpoint of $\mathbf{v}$ and $\mathbf{w}$.

For the second speed-up, we simply measure $\mathsf{PO}^\times$ using both methods, using 2^{16} samples, and check that both results are equal up to a 20% relative error.

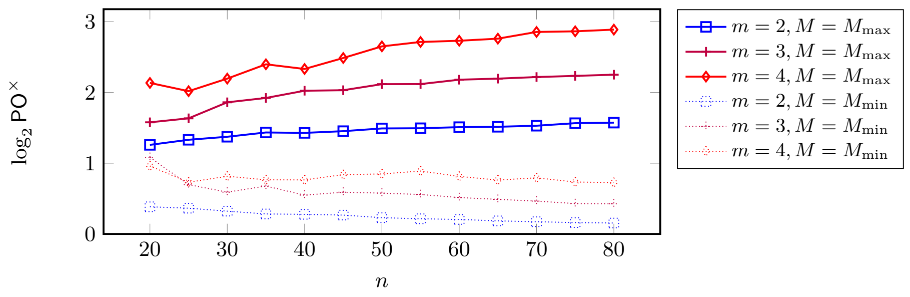
4.2 Trends

We plot the variation of $\mathsf{PO}^\times$ as a function of various parameters in Figure 1. These plots confirm that $\mathsf{PO}^\times$ indeed increases with m and M. Interestingly, for fixed m and either $M = M_{\text{min}}$ or $M = M_{\text{max}}$, $\mathsf{PO}^\times$ appears to be converging as n grows. We also note that $\mathsf{PO}^\times$ decrease with a for a fixed M.

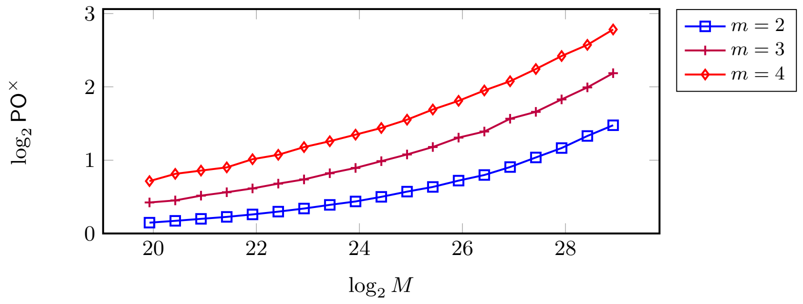
4.3 Concrete estimate in dimension 384

In this section, we will be more precise about costs, giving an gate count for time and a bit count for memory.

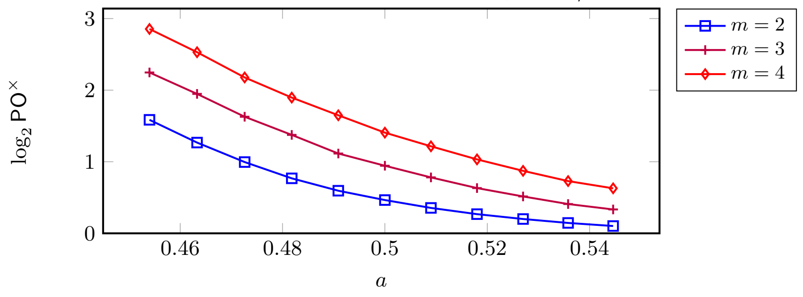
[4] The terminology was communicated to us by M. Albrecht, and is meant as a synonym of embarrassingly parallel.



Parameters: $a = 1/2$.



Parameters: $n = 80$, $a = 1/2$.



Parameters: $n = 80$, $M = 1/\mathcal{C}(0.55)$.

Fig. 1: Variations of $\mathsf{PO}^{\times}$ as a function of various parameters. Measured over 2^{14} samples per datapoint.

The [AGPS20] estimates. The original version of the software associated with [AGPS20] was costing list-decoding in the idealized model, that is, the cost of bucketing a vector was proportional to the number of buckets it falls into, following the formula[5]:

$$m \cdot M \cdot \mathcal{C}(a) \cdot \mathsf{C}_{\mathsf{ip}}(n) \tag{1}$$

[5] https://github.com/jschanck/eprint-2019-1161/blob/09d72d2125e75fdd35e49e54c35663a1affa212a/cost.py#L783

choosing $m = \log_2(n)$ and where $\mathsf{C}_{\mathsf{ip}}(n) = 2^{10} \cdot n$ denotes the cost of an inner product in dimension n in gates (computed at 32 bit precision). Not withstanding the idealization of ignoring the additive overhead LDO^+, this formula does not adequately reflects the cost stated in the original [BDGL16] paper; this was pointed out in [MAT22]. Indeed, in [BDGL16] the inner products are precomputed and reused through the tree enumeration, reducing the $\mathsf{C}_{\mathsf{ip}}(n)$ factor[6] to a $\mathsf{C}_{\mathsf{add}} = 160$ factor, the cost of a single 32 bit addition. Furthermore, the work of [MAT22] propose a variation on the tree enumeration order to tackle the m factor[7].

Following this report and further discussion on the NIST forum[8], this cost has been revised to[9]

$$\underbrace{m \cdot B \cdot \mathsf{C}_{\mathsf{ip}}(n/m) + mB \log_2 B(\mathsf{C}_{\mathsf{cp}} + \log_2 B)}_{\mathsf{LDO}^+} + M \cdot \mathcal{C}(a) \cdot (\mathsf{C}_{\mathsf{add}} + \log_2 B) \qquad (2)$$

where $\mathsf{C}_{\mathsf{cp}} = 32$ is the cost of a comparision (for sorting), and the additive terms $\log_2(B)$ are meant to account for moving and adressing pointers[10].

The addition of the LDO^+ term is in fact negligible, because the whole algorithm is costed in its *high-memory regime* $M = M_{\max}$, $R = 1$. Though, the memory consumption is not reported upon by the software, nor by other report using this software, nor in the reports of [AGPS20,GJ21,MAT22]. In the Kyber documentation (Round 3 version) [ABD+21], a memory cost of $2^{93.8}$ bits for sieving dimension 375, which corresponds to storing $N = 1/C(1/2)$ vectors of dimension n at 8-bit precision. That is, the memory was costed following the *low-memory regime*.

Costing time in the high memory regime, while costing memory in the low memory regime, all while ignoring the probabilistic overhead essentially corresponds to the idealized model for [BDGL16].

Further improvement on LDO^+. There has been some further improvement on implementing [BDGL16], in particular improving the LDO^+ term. In personal communication about [MLB17], Laarhoven mentioned that partial inner products need not be entirely sorted as only the top fraction is visited during the tree enumeration. In the low memory regime, one would expect that the tree visit about a single branch on average, so this sorting may be reduced down to

[6] Even without this precomputation, this cost should have been $\mathsf{C}_{\mathsf{ip}}(n/m)$.

[7] Though in the original [BDGL16] this factor comes from a *worst-case* analysis on the tree shape, and one would expect this factor to also vanish to 1 for large random trees without tweaking the tree enumeration.

[8] `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/Fm4cDfsx65s/m/m1vVrpoAAgAJ`

[9] `https://github.com/jschanck/eprint-2019-1161/blob/a4d3a53fe1f428fe3b4402bd63ee164ba6cc571c/cost.py#L801`

[10] We believe the last term should be $\log_2(M)$ rather than $\log(B)$ since there are M buckets to point to.

finding a single maximum. We will therefore ignore this cost which is dominated by the other LDO^+ term.

A second improvement comes from the implementation of [DSvW21], which replace explicit inner products with random vectors, by a sequence of implicit inner products using permutations and Hadamard matrices. This allows to decrease the cost from $m \cdot B \cdot \mathsf{C_{ip}}(n/m)$ to essentially

$$m \cdot \log_2 n/m \cdot B \cdot \mathsf{C_{add}} \tag{3}$$

where the additions are computed at 16-bit precision ($\mathsf{C_{add}} = 80$).

First Overhead Estimation. Having now costed the overheads following the state of the art [MLB17,DSvW21], and being equipped with an efficient method to measure the probability overhead, we may now provide an analysis of the previously neglected overheads. We fix parameters according to the optimization provided by the (revised) software of [AGPS20]:

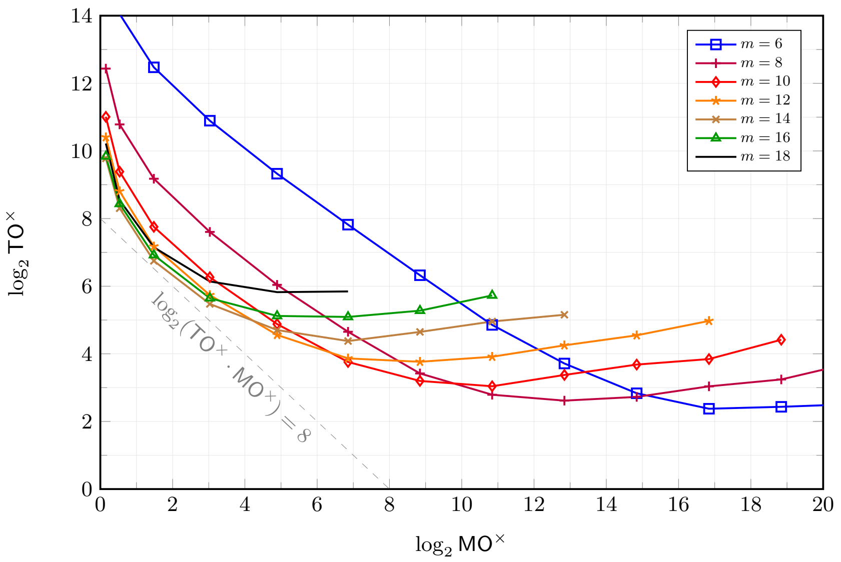
– Dimension is $n = 384$
– Xor-popcount parameters [FBB$^+$14,Duc18,ADH$^+$19] are done over 511 bits, with a threshold at 170, giving an positive detection rate of $\eta = 0.456$
– The number of vectors for the sieve is set to $N = 2/((1-\eta)C(1/2)) = 2^{86.0}$,
– The filter parameter is optimized at $a = 0.512$ by the revised script of [AGPS20], giving $M_{\min} = 2^{89.0}$ and $M_{\max} = 2^{127.6}$.

The revised script of [AGPS20] concludes on a cost of $\mathsf{T}_{\text{ideal}} = 2^{134.1}$ gates, while we need at least $\mathsf{M}_{\text{ideal}} = 8nN = 2^{97.6}$ bits of memory for storing all the sieve vectors. In Figure 2a we report on the time and memory overheads $(\mathsf{TO}^\times, \mathsf{MO}^\times)$ for various values of m and M.

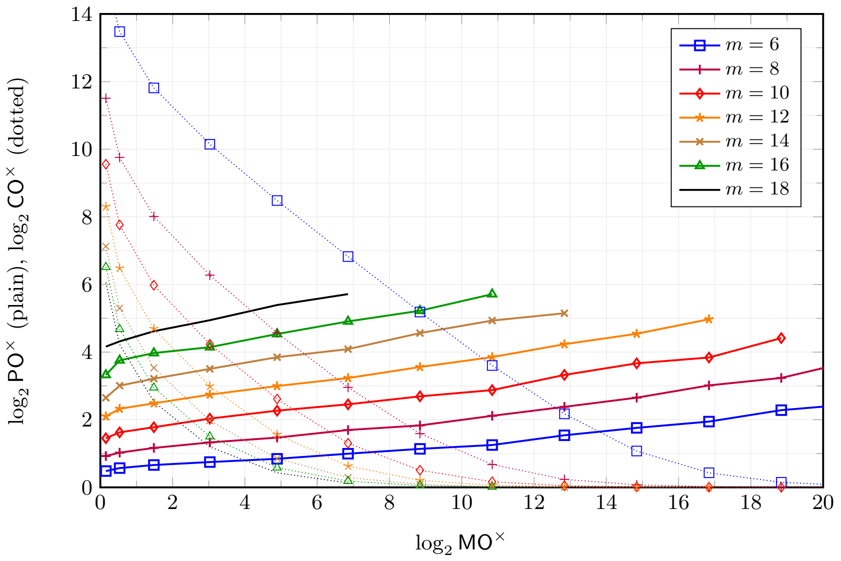
For each value of m, we first see time decrease as we increase the memory overhead, until a certain point, after which is starts increasing again. In other words, and perhaps surprisingly, the minimal time is not reached at the high-memory regime $M = M_{\max}$, but somewhere halfway $M_{\min} < M < M_{\max}$. A breakdown of the time overhead $\mathsf{TO}^\times = \mathsf{CO}^\times \cdot \mathsf{PO}^\times$ explains the phenomenon (Figure 2b): while $\mathsf{CO}^\times$ tends to 0, $\mathsf{PO}^\times$ increases at a steady rate; once $\mathsf{CO}^\times$ approaches 0, the decay of $\mathsf{CO}^\times$ gets lower than the increase of $\mathsf{PO}^\times$.

The plot shows that whatever the parametrization, the overhead on time-memory is $\mathsf{TO}^\times \cdot \mathsf{MO}^\times \geq 2^8$; furthermore, even for very large memory overhead the time overhead remains non-negligible.

One can also note that the number of vectors per bucket is $NC(a) \approx 1/8$, which is surprisingly low; only one bucket in 64 will actually have a pair to attempt reduction. This is understandable in the idealized model, because the cost of bucketing $\mathsf{C_{add}} + \log_2 B$ is significantly lower than a xor-popcount on 511 bits (costed at 3072 gates by [AGPS20]). But this could be sub-optimal when considering overheads. That is, to conclude, we first need to re-optimize the value of a with the overhead in the equation.

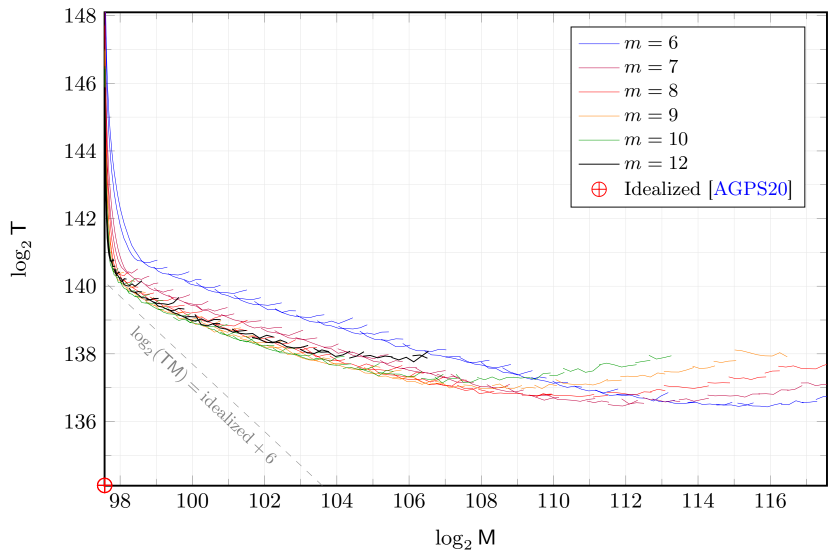


(a) Overall Time Overhead $\mathsf{TO}^\times$ as a function of the Memory Overhead $\mathsf{MO}^\times$.



(b) Breakdown of Time Overhead $\mathsf{TO}^\times = \mathsf{CO}^\times \cdot \mathsf{TO}^\times$ as a function of the Memory Overhead $\mathsf{MO}^\times$.
Each measure of $\mathsf{PO}^\times$ was done over 2^{12} samples. The computation took about 20 core-days.

Fig. 2: Overheads in dimension 384 for $a = 0.512$.



Each measure of $\mathsf{PO}^{\times}$ was done over 2^{13} samples. The computation took about 40 core-days.

Fig. 3: Cost in dimension 384 when optimizing a with overheads.

Reparametrizing, with Overheads. We now have three parameters to optimize over, m, M, and a, so we need to be mindful of the search space for the experiment to be feasible. We explore M by multiplicative increment of 2, and for each M, we use the previous curve to determine the relevant range for a. This explain the "hairy" appearance of our plot in Figure 3: each "thread" (for a fixed m) is in fact a union of curves for a fixed M and a small range of relevant a.

Qualitatively, the conclusion remain similar to that of the previous experiments, but quantitatively, the gap with the idealized cost is now a bit smaller near the low-memory regime (from 2^8 down to 2^6). Numerically, we can for example parametrize the algorithm to have time and memory complexity about ($\mathsf{T} = 2^{140.1}, \mathsf{M} = 2^{98}$), against a baseline of ($\mathsf{T}_{\text{ideal}} = 2^{134.1}, \mathsf{T}_{\text{ideal}} = 2^{97.6}$). The time-memory trade-off is however costly: from this point, decreasing the time by a factor 2^{-2} costs an extra factor of $2^{4.5}$ on memory, and gets worse as the curve flatten to minimal time around ($\mathsf{T} = 2^{136.5}, \mathsf{M} = 2^{112}$) .

5 Impact on Attacks

We clarify that the cost given all along this paper corresponds to the NNS task of finding all reducing pairs once inside sieving, and not that of the whole attack. In particular, we *can not* conclude that all attacks using sieving have their time-memory cost increased by a 2^6 factor. The most advanced attacks [GJ21,MAT22] have several stages, and reparametrizing them adequately should partially mitigate these overheads.

We also recall that there remain several known unknown in precisely modeling and costing lattice attacks, summarized in [ABD+21, Sec. 5.3]. This works resolves question Q2 (Idealized Near-Neighbors Search) of [ABD+21, Sec. 5.3]. We note that question Q7 (Refined BKZ Strategies) is now accounted for in the recent estimator of Albrecht [Alb22] under the label `bdd`.

We also warn against regressions on the accuracy of lattice attack estimates [GJ21,MAT22], such as the use of the Geometric-Series Assumption instead of a (progressive) BKZ simulator [CN11,DSDGR20,Alb22].

5.1 Mitigation inside Progressive-Sieve and Progressive-BKZ

Even in the simplest attack whose cost comes essentially from sieving, the overhead can also be mitigated. The reason is that the routine at hand is not only ran in the final sieving dimension (say $n = 384$) but also in dimension below it; for those calls in smaller dimension we can move on the time-memory curve.

More specifically, the simplest primal BKZ attack[11] makes about $d(i+1)$ calls in dimension $n-i$ where d is the total lattice dimension; these calls accumulates over progressive-sieving for each SVP oracle call [Duc18,ADH+19], and over progressive-BKZ tours. In the idealized model, this adds an extra dC^2 factor on time where $C = \sum_{i=0}^{\infty} 2^{-.292i} \approx 5.46$ stands for the progressivity overhead.

To correct the progressivity overhead using this strategy, we rely on the data collected at $n = 384$, and make the working assumption[12] that for small $i < 30$, the time-memory curve of Figure 3 is simply shifted by ($\mathsf{T} = 2^{-.292i}$, $\mathsf{M} = 2^{-.2075i}$). We aim to make optimal use of $\mathsf{M} = 2^{98}$ bits of memory throughout all calls to the inner routine in dimension $384 - i$.

For each i, we collect the minimal time T_i such that $\mathsf{M}_i < 2^{98}$, and finally compute $\sum(i+1)\mathsf{T}_i/\mathsf{T}_0 \approx 15.6$. This is to be compared with the idealized progressivity overhead squared $C^2 \approx 29.6$. That is, this strategy should mitigate the 2^6 overhead of the inner routine by a factor $2^{-0.9}$ on the whole primal BKZ attack, lowering the overhead down to $2^{5.1}$. Given the concavity of the curve, one would expect this mitigation works best in the low memory regime; for example, at $\mathsf{M} = 2^{100}$ we get $\sum(i+1)\mathsf{T}_i/\mathsf{T}_0 \approx 20.0$, i.e. a mitigation factor of $2^{-0.6}$.

For a single progressive sieve, we can also compare $\sum \mathsf{T}_i/\mathsf{T}_0 \approx 3.78$ to $C \approx 5.46$, and conclude on a mitigation factor of $2^{-0.5}$ at $\mathsf{M} = 2^{98}$.

6 Open Problems

The analysis of the overheads of the algorithm of [BDGL16] presented in this work is based on instantiating it is random product codes. While this is the only

[11] the BKZ blocksize β is here sligthly larger than n thanks to the dimensions for free of [Duc18].

[12] Collecting all those curves would be rather costly, about 40 core-days per curve. Furthermore, the software of [AGPS20] which we rely on is only set for n a multiple of 8.

proposed instantiation so far, the framework of [BDGL16] can also work with other efficiently list-decodable spherical codes. A natural open problem would therefore be to design spherical codes giving better trade-offs between $\mathsf{PO}^\times$, $\mathsf{CO}^\times$ and $\mathsf{MO}^\times$.

Another natural problem would be to find a theoretical model for the probabilistic overhead $\mathsf{PO}^\times$ as a function of the various parameters. We hope that the experimental data provided by the method of this work can be helpful for conjecturing or validating such a theoretical analysis.

References

ABD$^+$21. Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Vadim Lyubashevsky Tancrède Lepoint, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber, algorithm specifications and supporting documentation, 2021. Version 3.02. Available at `https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf`.

ADH$^+$19. Martin R Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 717–746. Springer, 2019.

AGPS20. Martin R Albrecht, Vlad Gheorghiu, Eamonn W Postlethwaite, and John M Schanck. Estimating quantum speedups for lattice sieves. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 583–613. Springer, 2020.

AKS01. Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 601–610. ACM, 2001.

Alb22. Martin Albrecht. Security estimates for lattice problems, 2022. Available at `https://github.com/malb/lattice-estimator`.

BDGL16. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 10–24. SIAM, 2016.

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

CN11. Yuanmi Chen and Phong Q Nguyen. Bkz 2.0: Better lattice security estimates. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2011.

DSDGR20. Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. Lwe with side information: attacks and concrete security estimation. In *Annual International Cryptology Conference*, pages 329–358. Springer, 2020.

DSvW21. Léo Ducas, Marc Stevens, and Wessel van Woerden. Advanced lattice sieving on gpus, with tensor cores. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 249–279. Springer, 2021.

Duc18. Léo Ducas. 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. Springer, 2018.

FBB$^+$14. Robert Fitzpatrick, Christian Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang. Tuning gausssieve for speed. In *International Conference on Cryptology and Information Security in Latin America*, pages 288–305. Springer, 2014.

GJ21. Qian Guo and Thomas Johansson. Faster dual lattice attacks for solving lwe with applications to crystals. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 33–62. Springer, 2021.

Laa15a. Thijs Laarhoven. Search problems in cryptography. 2015. Available at http://thijs.com/docs/phd-final.pdf.

Laa15b. Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *Annual Cryptology Conference*, pages 3–22. Springer, 2015.

LW15. Thijs Laarhoven and Benne de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In *International Conference on Cryptology and Information Security in Latin America*, pages 101–118. Springer, 2015.

MAT22. MATZOF. Report on the security of lwe: Improved dual lattice attack, 2022. Available at https://zenodo.org/record/6412487.

MLB17. Artur Mariano, Thijs Laarhoven, and Christian Bischof. A parallel variant of ldsieve for the svp on lattices. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 23–30. IEEE, 2017.

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