Faster Dual Lattice Attacks by Using Coding Theory

Abstract. We present a faster dual lattice attack on the Learning with Errors (LWE) problem, based on ideas from coding theory. Basically, it consists of revisiting the most recent dual attack of [MAT22] and replacing modulus switching by a decoding algorithm. This replacement achieves a reduction from small LWE to plain LWE with a very significant reduction of the secret dimension. We also replace the enumeration part of this attack by betting that the secret is zero on the part where we want to enumerate it and iterate this bet over other choices of the enumeration part. We estimate the complexity of this attack by making the optimistic, but realistic guess that we can use polar codes for this decoding task. We show that under this assumption the best attacks on Kyber and Saber can be improved by 1 and 6 bits.

Keywords: Learning with Errors, Dual attack, Fast Fourier Transform

1 Introduction

1.1 Background and Related Work

The LWE Problem. The Learning With Errors (LWE) problem was introduced by Regev [Reg05] and has since become a major ingredient for constructing basic and more advanced cryptographic primitives. It asks one to find $\mathbf{s}$ given $(\mathbf{A}, \mathbf{b})$ with $\mathbf{b} \equiv \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \bmod q$ where $\mathbf{e}$ has small entries. The variant where $\mathbf{s}$ is also small is called *small secret* LWE.[1] Its conjectured hardness against quantum computers further makes all these constructions supposedly post-quantum. It has become a central hardness assumption in cryptography and is frequently used to build cryptosystems whose security relies on it. For instance, in NIST's Post Quantum Standardization Process, two out of four selected algorithms are based on its conjectured hardness [SSTX09, LPR10].

Dual attacks. The most efficient cryptanalysis techniques against LWE(-like) problems are "primal" and "dual" lattice attacks, named depending on whether lattice reduction is performed on the "primal" lattice related to $\mathbf{A}$ or the "dual" lattice $\{\mathbf{x} \in \mathbb{Z}_q^m \mid \mathbf{x} \cdot \mathbf{A} \equiv \mathbf{0} \bmod q\}$. Until recently, dual attacks were generally considered less efficient for secrets $\mathbf{s}$ drawn from a sufficiently wide distribution.

[1] Because of the reduction given in [ACPS09], when we speak of small LWE, this generally means LWE where $\mathbf{s}$ is at least as short as $\mathbf{e}$.

They were introduced in [MR09]. In its simplest form, a dual attack is a distinguishing attack which is given either $(\mathbf{A}, \mathbf{A}\cdot\mathbf{s}+\mathbf{e})$ or $(\mathbf{A}, \mathbf{u})$ where $(\mathbf{A}, \mathbf{u})$ are uniform and $\mathbf{e}$ is short, and answers if we are in the first or second case. It starts by computing many short $\mathbf{x}_j$ such that $\mathbf{x}_j^\intercal \cdot \mathbf{A} \equiv \mathbf{0} \mod q$ and the associated $\langle \mathbf{x}_j, \mathbf{b}\rangle$'s. Then, we either obtain $\langle \mathbf{x}_j, \mathbf{b}\rangle = \mathbf{x}_j^\intercal \cdot \mathbf{A}\cdot\mathbf{s} + \langle \mathbf{x}_j, \mathbf{e}\rangle \equiv \langle \mathbf{x}_j, \mathbf{e}\rangle \mod q$ or $\langle \mathbf{x}_j, \mathbf{u}\rangle$. The former follows a distribution with small entries, *i.e.* the distribution of $|e_j|$ for $e_j \coloneqq \langle \mathbf{x}_j, \mathbf{e}\rangle$ is biased towards elements $< q/2$, and the latter follows a uniform distribution mod q.

Without loss of generality, by the reduction [ACPS09] of standard LWE (where $\mathbf{s}$ is uniformly distributed over $\mathbb{Z}_q$) to LWE where $\mathbf{s}$ and $\mathbf{e}$ follow the same distribution, we may assume that $\mathbf{s}$ is also short and we will consider this case from now on. In [ADPS16], the "normal form" of the dual attack was introduced which finds short $\mathbf{x}_j$ such that $\mathbf{x}_j^\intercal \cdot \mathbf{A} \equiv \mathbf{y}_j^\intercal \mod q$ with $\mathbf{y}_j$ which is also short. We then obtain

$$\langle \mathbf{x}, \mathbf{b}\rangle \equiv \mathbf{x}_j^\intercal \cdot \mathbf{A}\cdot\mathbf{s} + \langle \mathbf{x}_j, \mathbf{e}\rangle \equiv \langle \mathbf{y}_j, \mathbf{s}\rangle + \langle \mathbf{x}_j, \mathbf{e}\rangle \mod q \tag{1}$$

which follows a distribution with small entries when $\mathbf{y}_j, \mathbf{s}, \mathbf{x}_j$ and $\mathbf{e}$ are short.

There have been a sequence of developments in dual attacks which have shown in the end their ability to surpass (at least theoretically) primal attacks [GJ21, MAT22]. A first development [Alb17] was to combine these attacks with a guessing stage. The idea is to split $\mathbf{A} = [\mathbf{A}_{\text{en}} \;\; \mathbf{A}_{\text{lat}}]$ and the secret $\mathbf{s} = \begin{bmatrix}\mathbf{s}_{\text{en}} \\ \mathbf{s}_{\text{lat}}\end{bmatrix}$ accordingly, so that

$$\mathbf{b} \equiv \mathbf{A}_{\text{en}}\cdot\mathbf{s}_{\text{en}} + \mathbf{A}_{\text{lat}}\cdot\mathbf{s}_{\text{lat}} + \mathbf{e} \mod q. \tag{2}$$

We only look now for short vectors $\mathbf{x}$ and $\mathbf{y}_{\text{lat}}$ that are such that $\mathbf{x}^\intercal \cdot \mathbf{A} \equiv \mathbf{y}_{\text{lat}}^\intercal \mod q$ and define $\mathbf{y}_{\text{en}}$ by $\mathbf{y}_{\text{en}}^\intercal \stackrel{\triangle}{=} \mathbf{x}^\intercal \mathbf{A}_{\text{en}}$. The point is that looking for such vectors is faster because of the dimension reduction. We combine this with a guessing strategy for $\mathbf{s}_{\text{en}}$ and check whether the $\langle \mathbf{x}, \mathbf{b}\rangle - \langle \mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}}\rangle$'s are tilted towards small values or not. This comes from the fact that

$$\begin{aligned}\langle \mathbf{x}, \mathbf{b}\rangle &\equiv \mathbf{x}^\intercal \cdot \mathbf{A}_{\text{en}}\cdot\mathbf{s}_{\text{en}} + \mathbf{x}^\intercal \cdot \mathbf{A}_{\text{lat}}\cdot\mathbf{s}_{\text{lat}} + \langle \mathbf{x}, \mathbf{e}\rangle \mod q &(3)\\ &\equiv \langle \mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}}\rangle + \langle \mathbf{y}_{\text{lat}}, \mathbf{s}_{\text{lat}}\rangle + \langle \mathbf{x}, \mathbf{e}\rangle \mod q &(4)\end{aligned}$$

and here $\langle \mathbf{y}_{\text{lat}}, \mathbf{s}_{\text{lat}}\rangle + \langle \mathbf{x}, \mathbf{e}\rangle$ tends to take small values. In a sense, the gain comes from splitting the original task into two less complex subtasks. In [EJK20] this was generalized to more general secret distributions paired with additional improvements on the exhaustive search over $\mathbf{s}_{\text{en}}$. In [GJ21] further improvements were presented. In particular, the search over $\mathbf{s}_{\text{en}}$ is realized using a Fast Fourier Transform style algorithm and the search space is significantly reduced by roughly considering only the most significant symbols of $\mathbf{s}_{\text{en}}$. In [MAT22] this last step is replaced by "modulus switching" [BV11, AFFP14, KF15, GJS15a] which provides significant performance gains.[2] Overall, these newer iterations of

[2] Another significant gain reported in [MAT22] is due to an improvement to the lattice sieving algorithm from [BDGL16] but discussing this is out of scope of this work.

the dual attack relate the search space to the underlying secret in such a way that large dimensions can now be covered even when the norm of the secret vector is not very small (previous versions of the dual attack relied on, say, coefficients $\mathbf{s}_i \in \{-1, 0, 1\}$).

Reduction from LWE to small LWE with many samples but high noise. It is insightful to adopt the point of view of [CDMT22] that followed a very similar path as these improvements on dual attacks, but this time for decoding a linear code. Here too, small weight codewords (in the dual code) are used to produce inner products that are biased and the problem at hand is split in two or three pieces and the codewords are only required to be of low weight on one of the pieces, which as in the lattice case decreases significantly the running time for finding them. The algorithm is explained there by viewing the whole approach as reducing the decoding problem to a very noisy LPN problem but with an exponential number of samples which is then solved by a fast Fourier approach which is quite reminiscent of the [GJ21, MAT22] approach. It is easier to explain how this recent thread of work relates in some sense to a reduction to a new LWE problem, if we use the [MAT22] approach and forget for one moment about modulus switching. The matrix $\mathbf{A}$ is split here into three parts $\mathbf{A} = \begin{bmatrix}\mathbf{A}_{\text{en}} & \mathbf{A}_{\text{fft}} & \mathbf{A}_{\text{lat}}\end{bmatrix}$ and the secret $\mathbf{s}$ is split accordingly $\mathbf{s} = \begin{bmatrix}\mathbf{s}_{\text{en}} \\ \mathbf{s}_{\text{fft}} \\ \mathbf{s}_{\text{lat}}\end{bmatrix}$ so that

$$\mathbf{b} \equiv \mathbf{A}_{\text{en}} \cdot \mathbf{s}_{\text{en}} + \mathbf{A}_{\text{fft}} \cdot \mathbf{s}_{\text{fft}} + \mathbf{A}_{\text{lat}} \cdot \mathbf{s}_{\text{lat}} + \mathbf{e} \mod q. \tag{5}$$

We perform again a search of short vectors $\mathbf{x}$ and $\mathbf{y}_{\text{lat}}$ only on the $\mathbf{A}_{\text{lat}}$ part, that is such that $\mathbf{x}^\intercal \cdot \mathbf{A}_{\text{lat}} \equiv \mathbf{y}_{\text{lat}}^\intercal \mod q$ and define $\mathbf{y}_{\text{en}}, \mathbf{y}_{\text{fft}}$ by $\mathbf{y}_{\text{en}}^\intercal \stackrel{\triangle}{=} \mathbf{x}^\intercal \cdot \mathbf{A}_{\text{en}}$, $\mathbf{y}_{\text{fft}}^\intercal \stackrel{\triangle}{=} \mathbf{x}^\intercal \cdot \mathbf{A}_{\text{fft}}$. Similarly to (4), we have

$$\langle \mathbf{x}, \mathbf{b} \rangle \equiv \langle \mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}} \rangle + \langle \mathbf{y}_{\text{fft}}, \mathbf{s}_{\text{fft}} \rangle + \langle \mathbf{y}_{\text{lat}}, \mathbf{s}_{\text{lat}} \rangle + \langle \mathbf{x}, \mathbf{e} \rangle \mod q$$

Assume that we have guessed the right $\mathbf{s}_{\text{en}}$, then obviously if we define

$$\mathbf{a}' \stackrel{\triangle}{=} \mathbf{y}_{\text{fft}} \tag{6}$$

$$\mathbf{s}' \stackrel{\triangle}{=} \mathbf{s}_{\text{fft}} \tag{7}$$

$$b' \stackrel{\triangle}{=} \langle \mathbf{x}, \mathbf{b} \rangle - \langle \mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}} \rangle \tag{8}$$

$$e' \stackrel{\triangle}{=} \langle \mathbf{y}_{\text{lat}}, \mathbf{s}_{\text{lat}} \rangle + \langle \mathbf{x}, \mathbf{e} \rangle \tag{9}$$

then clearly the $(\mathbf{a}', b')$ form LWE samples associated to the secret $\mathbf{s}'$ (and the rows of the matrix $\mathbf{A}'$ of the new LWE problem are formed by the $\mathbf{a}'$'s) since

$$b' = \langle \mathbf{x}, \mathbf{b} \rangle - \langle \mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}} \rangle \tag{10}$$

$$\equiv \langle \mathbf{y}_{\text{fft}}, \mathbf{s}_{\text{fft}} \rangle + \langle \mathbf{y}_{\text{lat}}, \mathbf{s}_{\text{lat}} \rangle + \langle \mathbf{x}, \mathbf{e} \rangle \mod q \tag{11}$$

$$\equiv \langle \mathbf{a}', \mathbf{s}' \rangle + e' \mod q \tag{12}$$

and the e''s tend to be small. We are really in the situation, where we have many (actually exponentially many in the attacks) LWE samples coming from our search of short vectors $\mathbf{x}$ and $\mathbf{y}_{\mathrm{lat}}$, with a quite high noise on e' (these integers are only slightly tilted towards low values) and a short secret vector $\mathbf{s}' \stackrel{\triangle}{=} \mathbf{s}_{\mathrm{fft}}$, therefore precisely in the small LWE regime. Of course, if we have made the wrong guess on $\mathbf{s}_{\mathrm{en}}$, then the $(\mathbf{a}', b')$'s are independent and uniformly distributed. Our task after guessing $\mathbf{s}_{\mathrm{en}}$ is therefore really the task of distinguishing many small LWE samples which are very noisy from the uniform distribution. In [MAT22] this operation is performed by a modulo switching approach and a fast Fourier transform. However, this approach is not able to take into account that we are in the small LWE scenario here, namely that the secret $\mathbf{s}'$ is way shorter than $\mathbf{e}'$ now.

1.2 Contributions

A Coding Theoretic Approach. Our first contribution is a coding theoretic approach which is inspired by the coded-BKW approach [GJS15b] to switch from this small LWE problem to a standard LWE problem but with a huge reduction in the dimension (but also somewhat higher noise) and then solve it with an FFT technique combined or not with modulo switching (but where modulo switching is used for other reasons than in [MAT22]). The use of lattice codes is however different here. In [GJS15b], it was used to relax the collision condition in the BKW steps, here it is really used for transforming the small LWE problem into a standard LWE problem with a significant decrease in the dimension of the secret.

Basically, the idea is to look for a linear space $\mathscr{C}$ of dimension k_{fft} in $\mathbb{Z}_q^{n_{\mathrm{fft}}}$, where n_{fft} is the length of the vectors $\mathbf{s}_{\mathrm{fft}}$ and $\mathbf{y}_{\mathrm{fft}}$:

$$\mathscr{C} \stackrel{\triangle}{=} \left\{ \mathbf{G}\mathbf{u} \ : \ \mathbf{u} \in \mathbb{Z}_q^{k_{\mathrm{fft}}} \right\} \tag{13}$$

that

(i) we can *decode* efficiently, meaning that for all $\mathbf{y}_{\mathrm{fft}} \in \mathbb{Z}_q^{n_{\mathrm{fft}}}$ we are able to find efficiently $\mathbf{u} \in \mathbb{Z}_q^{k_{\mathrm{fft}}}$ such that $\mathbf{G}\mathbf{u}$ is close to $\mathbf{y}_{\mathrm{fft}}$,
(ii) the decoding distance, namely the typical/average Euclidean distance between the $\mathbf{G}\mathbf{u}$ we produce by this decoding algorithm and $\mathbf{y}_{\mathrm{fft}}$ is as small as possible. The product codes used in [BDGL16] or polar codes [Arı09, KU10, Şaş11] are examples where this is possible to achieve.

Note that here the operation $\mathbf{G}\mathbf{u}$ is performed over $\mathbb{Z}_q$ (namely modulo q).

To explain what we have in mind here, consider an LWE sample of the form $(\mathbf{y}_{\mathrm{fft}}, \langle \mathbf{y}_{\mathrm{fft}}, \mathbf{s}_{\mathrm{fft}} \rangle + e')$. We use our decoding algorithm on $\mathbf{y}_{\mathrm{fft}}$ and produce $\mathbf{G}\mathbf{u}$ such that $\mathbf{e}' \stackrel{\triangle}{=} (\mathbf{y}_{\mathrm{fft}} - \mathbf{G}\mathbf{u}) \bmod q$ is small. Notice now that

$$\begin{aligned} \langle \mathbf{y}_{\mathrm{fft}}, \mathbf{s}_{\mathrm{fft}} \rangle &\equiv \langle \mathbf{G}\mathbf{u} + \mathbf{e}', \mathbf{s}_{\mathrm{fft}} \rangle \bmod q \\ &\equiv \langle \mathbf{u}, \mathbf{G}^{\intercal}\mathbf{s}_{\mathrm{fft}} \rangle + \langle \mathbf{e}', \mathbf{s}_{\mathrm{fft}} \rangle \bmod q. \end{aligned}$$

Since the new term $\langle \mathbf{e}', \mathbf{s}_{\text{fft}} \rangle$ is also tilted towards small values, we have a new LWE problem given by samples $(\mathbf{a}", b")$ where

$$\mathbf{a}" \stackrel{\triangle}{=} \mathbf{u} \tag{14}$$

$$\mathbf{s}" \stackrel{\triangle}{=} \mathbf{G}^{\intercal}\mathbf{s}_{\text{fft}} \bmod q \tag{15}$$

$$b" \stackrel{\triangle}{=} \langle \mathbf{x}, \mathbf{b} \rangle - \langle \mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}} \rangle \bmod q \tag{16}$$

$$e" \stackrel{\triangle}{=} \langle \mathbf{y}_{\text{lat}}, \mathbf{s}_{\text{lat}} \rangle + \langle \mathbf{x}, \mathbf{e} \rangle + \langle \mathbf{e}', \mathbf{s}_{\text{fft}} \rangle \bmod q. \tag{17}$$

The secret is $\mathbf{G}^{\intercal}\mathbf{s}_{\text{fft}}$ which lives in $\mathbb{Z}_q^{k_{\text{fft}}}$ has much smaller dimension (but is not small anymore) and the noise term $e"$ is somewhat bigger because of the additional $\langle \mathbf{e}', \mathbf{s}_{\text{fft}} \rangle$ term.

It turns out that the modulo switching technique can also be viewed as using a code in this context, it is basically the lattice code $(q/p\mathbb{Z})^{n_{\text{fft}}}$ where p is an integer much smaller than q. It has a very simple decoding algorithm which consists of rounding, however also very poor performance in terms of the norm of the error vector $\mathbf{e}'$ we produce. This translates into a bigger noise $e"$ in the new LWE problem and worse performance for the distinguisher.

The optimal distance we can afford here is given by the lattice-theoretic analogue of the Gilbert-Varshamov quantity in coding theory, it corresponds to the radius ω of a ball B_ω in $\mathbb{R}^{n_{\text{fft}}}$ such that

$$\#\mathscr{C} \cdot \text{Vol}\left(B_\omega\right) = q^{n_{\text{fft}}}. \tag{18}$$

If we let ω_0 be the radius of a ball which is of volume 1 in $\mathbb{R}^{n_{\text{fft}}}$, that is $\omega_0 \approx \sqrt{\frac{n_{\text{fft}}}{2\pi e}}$, then this means that $\left(\frac{\omega}{\omega_0}\right)^{n_{\text{fft}}} = q^{n_{\text{fft}}-k_{\text{fft}}}$, that is

$$\omega \approx \sqrt{\frac{n_{\text{fft}}}{2\pi e}} q^{1-\frac{k_{\text{fft}}}{n_{\text{fft}}}}. \tag{19}$$

The new LWE problem is solved similarly to [MAT22] after performing modulus switching, namely computing a Fourier transform. In [MAT22] this is done by performing a Fourier transform over $\mathbb{Z}_p^{n_{\text{fft}}}$ in our case, this is performed by computing the Fourier transform over $\mathbb{Z}_q^{k_{\text{fft}}}$. Therefore it really makes sense to compare both approaches when $p^{n_{\text{fft}}} = q^{k_{\text{fft}}}$ and compare the decoding distance we obtain. After renormalizing by multiplying by a factor q/p, it turns out that the decoding distance ω_{Matzov} in the case of Matzov is of the form

$$\omega_{\text{Matzov}} \approx \sqrt{\frac{n_{\text{fft}}}{12}} q^{1-\frac{k_{\text{fft}}}{n_{\text{fft}}}}. \tag{20}$$

We gain with our approach a factor of $\sqrt{\frac{2\pi e}{12}} \approx 1.19$ in the decoding distance which has a definite impact.

Our approach can also be combined with modulus switching. But here the purpose is totally different. We choose a new modulus where performing the

FFT is less complex (for instance a power of two does the job) and to help finding a good choice for k_{fft}. Indeed because q is often very big we are often in a case where after optimizing the parameters of the attack we would like to choose a decoding distance ω for which the k_{fft} which satisfies (18) lies between two integer values k and $k+1$, but q^k is way too small and q^{k+1} way too big. In this case, changing the modulo q to a new modulo q' which is of the same order as q has two effects: (i) the added rounding noise is much smaller than in [MAT22], (ii) we have the latitude of choosing an integer for which the FFT is much more efficient and for which ${q'}^{k_{\text{fft}}}$ is closer to what the optimization of parameters would like.

The Prange bet. An important ingredient used for speeding up dual attacks in the coding context [CDMT22] is the Prange bet [Pra62]. It consists in making the bet that the error is much lower (and actually even non existent) on some parts of the word that we want to decode and using this bet to decode. Of course, the bet might be wrong and we have to start all over again until making the right guess, but we might hope to gain in complexity if the task we face by making the right bet is much easier than the original decoding task. In the case of [Pra62] it just amounts to solve a linear system. This idea can be used in our setting in a natural way. One of them is to use it in the enumeration part of the dual attack. It turns out that in this case the optimal bet we can make is that the secret is equal to zero on this part (see Section 3.5). It turns out that this results in a slightly improved attack.

Results. All in all, this approach gives in the end some improvement on the most recent dual attack of [MAT22]. We estimate the complexity of this attack by making the assumption that we can use polar codes for this decoding task. We show that under this assumption the best attacks on Kyber and Saber can be improved by 1 and 6 bits. We have validated this assumption experimentally by implementing the polar code we need here and running the corresponding decoder. Their decoding distance comes very close to the Gilbert Varshamov distance as shown in Section D of the appendix.

2 Notation and Preliminaries

2.1 Notation

Recall that $e^{ix} = \cos(x) + i\,\sin(x)$. For a complex number $z = a + ib$, $\Re(z)$ denotes its real part a. We write $[\![x, y]\!]$ for the interval $\{x, x+1, \ldots, y\} \subset \mathbb{Z}$. We denote matrices by bold uppercase letters, *e.g.* $\mathbf{A}$, and vectors by bold lowercase letters, *e.g.* $\mathbf{v}$. We treat vectors as column matrices. We write $\mathbf{v}^T$ for the transpose of $\mathbf{v}$.

For any $x \in \mathbb{Z}_q$, denote by $\hat{x} \in x + q\mathbb{Z}$ the unique integer such that $|\hat{x}| \leqslant \frac{q-1}{2}$. We extend this notion to vectors in $\mathbf{x} \in \mathbb{Z}_q^n$ componentwise. In other words, $\tilde{\mathbf{x}}$ is the lift from $\mathbb{Z}_q$ to $\mathbb{Z}$ centered on 0. We define $\|\mathbf{x}\|$ for $\mathbf{x} \in \mathbb{Z}_q^n$ as $\|\hat{\mathbf{x}}\|$.

For $x \in \mathbb{R}, [x]$ stands for the closest integer to x and $\{x\} \stackrel{\triangle}{=} x - [x]$ (by convention, we assume that for all $n \in \mathbb{Z}$, $\left[n + \frac{1}{2}\right] = n + 1$). Naturally, the notation is extended to vectors ; that is for any $\mathbf{x} \stackrel{\triangle}{=} (x_1, \cdots, x_n) \in \mathbb{R}^n$:

$$[\mathbf{x}] \stackrel{\triangle}{=} ([x_1], \cdots, [x_n])) \in \mathbb{Z}^n \tag{21}$$

$$\{\mathbf{x}\} \stackrel{\triangle}{=} (\{x_1\}, \cdots, \{x_n\}) \in \left[-\tfrac{1}{2}, \tfrac{1}{2}\right)^n \tag{22}$$

We let $\phi(x) \stackrel{\triangle}{=} \frac{1}{\sqrt{2\pi}} \exp(-x^2/2)$, respectively $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-t^2/2)\ \mathrm{d}t$ be the density and the cumulative distribution function (cdf) of the standard normal distribution and $\Phi^{-1}(x) : [0, 1) \to \mathbb{R}$ its inverse. We denote a random variable X of distribution D by $X \sim D$.

2.2 Lattices

A lattice $\mathcal{L}$ is a discrete subgroup of $\mathbb{R}^d$. We can represent it as $\{\sum x_i \cdot \mathbf{b}_i | x_i \in \mathbb{Z}\}$ where $\mathbf{b}_i$ are the columns of a matrix $\mathbf{B}$, we may write $\mathcal{L}(\mathbf{B})$. If $\mathbf{B}$ has full column rank, we call $\mathbf{B}$ a basis.

While the central object of this work, the dual attack, critically relies on lattice reduction, such as the BKZ algorithm, we mostly make blackbox use of these algorithms here. Thus, we refer the reader to *e.g.* [GJ21, MAT22] for details. In particular, the blackbox use we make of lattice reduction algorithms and, critically, lattice sieving algorithms is captured in Algorithm 1.

Algorithm 1: Short Vectors Sampling Procedure [GJ21]

Input: A basis $\mathbf{B} = \left[\mathbf{b}_0 \ \ldots \ \mathbf{b}_{d-1}\right]$ for a lattice and $2 \leq \beta_0, \beta_1 \in \mathbb{Z} \leqslant d$ and N.
Output: A list of N vectors from the lattice.

```
L={}.
for i ∈ [0, ⌈N/N_sieve(β_1)⌉ − 1] do
    Randomise the basis B.
    Run BKZ-β_0 to obtain a reduced basis b'_0, ..., b'_{d−1}.
    Run a sieve in dimension β_1 on the sublattice spanned by b'_0, ..., b'_{β_1−1} to
      obtain a list of N_sieve(β_1) vectors and add them to L.
return L
```

In Algorithm 1 the BKZ-β_0 call performs lattice reduction with parameter β_0 where the cost of the algorithm scales at least exponentially with β_0. The BKZ algorithm proceeds by making polynomially many calls to an SVP oracle. In this work, this oracle is instantiated using a lattice sieving algorithm which is also called explicitly in Algorithm 1 with parameter β_1. Such a sieving algorithm outputs $N_{\mathrm{sieve}}(\beta_1)$ short vectors in the lattice $\mathcal{L}(\mathbf{B})$ and has a cost exponential in β_1. The magnitude $N_{\mathrm{sieve}}(\beta_1)$ also grows exponentially with β_1 but slower than the cost of sieving. We will write $T_{\mathrm{BKZ}}(d, \beta_0)$ for the cost of running BKZ-β_0

in dimension d and $T_{\text{sieve}}(\beta_1)$ for the cost of sieving in dimension β_1. We may instantiate the lattice sieve with a classical algorithm [BDGL16] which has a cost of $2^{0.292\,\beta_1+o(\beta_1)}$. Thus, according to the best known algorithms we have $T_{\text{BKZ}}(d,\beta_0) \in \mathsf{poly}(d)\cdot 2^{\Theta(\beta_0)}$ and $T_{\text{sieve}}(\beta_1) \in 2^{\Theta(\beta_1)}$. More specifically, we take these complexities from [MAT22, Lemma 4.1, Assumption 7.3].

Lemma 2.1 (Short Vectors Sampling Complexity). *Let* $\mathbf{B}$ *be a basis of a* d*-dimensional lattice. Then, the running time* T_{sample} *of Algorithm 1 for outputting at least* N *short vectors is:*

$$T_{\text{sample}}(d,\beta_0,\beta_1,N) = \left\lceil \frac{N}{N_{\text{sieve}}(\beta_1)} \right\rceil \cdot \left(T_{\text{BKZ}}(d,\beta_0) + T_{\text{sieve}}(\beta_1)\right) \tag{23}$$

$$\text{where } T_{\text{BKZ}}(d,\beta_0) = C_{\text{prog}}^2 \cdot (d-\beta_0+1)\cdot T_{\text{NNS}}(\beta_0^{\text{eff}}) \tag{24}$$

$$\text{and } T_{\text{sieve}}(\beta_1) = C_{\text{prog}}\cdot T_{\text{NNS}}(\beta_1) \tag{25}$$

where $N_{\text{sieve}}(\beta) = \left(\sqrt{\frac{4}{3}}\right)^{\beta}$ *is the expected number of sieve results,* $T_{\text{NNS}}(\beta)$ *is the time complexity for finding all close pairs in dimension* β *(see [AGPS20] with improvment of Matzov [MAT22, Section 6]),* $C_{\text{prog}} = 1/\left(1-2^{-0.292}\right)$ *is the number of close pairs search to run and* β^{eff} *is the optimal sieve dimension to use for solving* SVP *for lattices in dimension* β*.*

Note that in [Duc18], β^{eff} is estimated as $\beta^{\text{eff}} = \beta - \frac{\beta\log(4/3)}{\log(\beta/(2\pi e))}$ whereas in [ADH+19], it is estimated as $\beta^{\text{eff}} = \beta - (0.0757\beta + 11.46)$ for a certain range of dimensions.

The lengths of the short vectors produced by Algorithm 1. Assuming that the Gaussian Heuristic (GH) and the Geometric Series Assumption (GSA) [Sch03] hold for a an m-dimensional lattice, applying BKZ-β to it produces vectors $\mathbf{x}$ of length [Che13]

$$\|\mathbf{x}\| \approx \delta_0^m \cdot \text{Vol}(\mathcal{L})^{\frac{1}{m}}, \tag{26}$$

where $\delta(\beta) = \left(\frac{\beta}{2\pi e}(\pi\beta)^{\frac{1}{\beta}}\right)^{\frac{1}{2(\beta-1)}}$ is the root-Hermite factor. With these assumptions, the expected length of the short vectors produced by Algorithm 1 is given by

Lemma 2.2 (Short Vectors Sampling Procedure's Quality and Correctness [MAT22, Lemma 4.2]). *Let* Λ *be a* d*-dimensional lattice. Then, Algorithm 1 outputs at least* N *vectors of length* ℓ *given by*

$$\ell \triangleq \det(\Lambda)^{1/d}\cdot N_{\text{sieve}}(\beta_1)^{1/\beta_1}\cdot\sqrt{\frac{\beta_1}{2\pi e}\cdot(\pi\beta_1)^{1/\beta_1}}\cdot\delta(\beta_0)^{d-\beta_1} \tag{27}$$

with $N_{\text{sieve}}(\beta) = \left(\sqrt{\frac{4}{3}}\right)^{\beta}$.

2.3 Learning with Errors

The Learning with Errors problem (LWE) is defined as follows.

Definition 2.3 (LWE). *Let $n, m, q \in \mathbb{N}$, and let χ_s, χ_e be distributions over $\mathbb{Z}_q$. Denote by $\mathrm{LWE}_{n,m,\chi_s,\chi_e}$ the probability distribution on $\mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^m$ obtained by sampling the coordinates of the matrix $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ independently and uniformly over $\mathbb{Z}_q$, sampling the coordinates of $\mathbf{s} \in \mathbb{Z}_q^n$, $\mathbf{e} \in \mathbb{Z}_q^m$ independently from χ_s and χ_e respectively, setting $\mathbf{b} \coloneqq \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \bmod q$ and outputting $(\mathbf{A}, \mathbf{b})$.*

We define two problems:

- Decision-LWE. Distinguish the uniform distribution over $\mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^m$ from $\mathrm{LWE}_{n,m,\chi_s,\chi_e}$.
- Search-LWE. Given a sample from $\mathrm{LWE}_{n,m,\chi_s,\chi_e}$, recover $\mathbf{s}$.

A popular distribution for χ_s and χ_e is the centered binomial distribution whose definition is

Definition 2.4 (centered binomial distribution). *The centered binomial distribution Z_s of parameter s is defined as:*

$$Z \sim \sum_{i=1}^{s} X_i - Y_i \tag{28}$$

Where the X_i's and Y_i's are identically and independently distributed according to the uniform distribution over $\{0, 1\}$.

2.4 Discrete Gaussian Distribution

Let $\sigma > 0$. For any $\mathbf{x} \in \mathbb{R}^d$, we let $\rho_\sigma(\mathbf{x}) \coloneqq \exp(-\|\mathbf{x}\|^2 / 2\sigma^2)$. Note that this is different from the other (also commonly used) definition, where $\frac{1}{2}$ is replaced by π in the exponent. This change is inconsequential to our results. We extend the definition of $\rho_\sigma(\cdot)$ to sets of vectors $\mathcal{S}$ by letting $\rho_\sigma(\mathcal{S}) \coloneqq \sum_{\mathbf{x}\in\mathcal{S}} \rho_\sigma(\mathbf{x})$. For any lattice $\mathcal{L} \subset \mathbb{R}^d$, we denote by $D_{\mathcal{L},\sigma}$ the discrete Gaussian distribution over $\mathcal{L}$, defined by $D_{\mathcal{L},\sigma}(\mathbf{x}) = \rho_\sigma(\mathbf{x})/\rho_\sigma(\mathcal{L})$ for all $\mathbf{x} \in \mathcal{L}$. Observing that $D_{\mathbb{Z}^n,\sigma}(\mathbf{x})$ only depends on $\|\mathbf{x}\|$, we abuse notation and for $\ell = \|\mathbf{x}\|$ write $\rho_\sigma(\ell) = \exp(-\ell^2/2\sigma^2)$ and $D_{\mathbb{Z}^n,\sigma}(\ell) = \rho_\sigma(\ell)/\rho_\sigma(\mathbb{Z}^n)$.

We will also make use of the modular discrete Gaussian distribution. For any $q \in \mathbb{N}$, we denote by $D_{\mathbb{Z}_q^d,\sigma}$ the modular discrete Gaussian distribution over $\mathbb{Z}_q^d$ defined by

$$D_{\mathbb{Z}_q^d,\sigma}(\mathbf{x}) = \frac{\rho_\sigma(\mathbf{x} + q\mathbb{Z}^d)}{\rho_\sigma(\mathbb{Z}^d)}. \tag{29}$$

Note that the distribution $D_{\mathbb{Z}_q^d,\sigma}$ is isomorphic to the distribution $D^d_{\mathbb{Z}_q,\sigma}$, a fact that we will use often implicitly.

2.5 Lattice codes and quantization

To analyze the performance of our coding technique, it is convenient to view it as a lattice code through Construction A that we now recall.

Definition 2.5 (Construction A). *Let $\mathscr{C}$ be a linear code of dimension k and length n over $\mathbb{Z}_q$ (where q is prime), that is a subspace of dimension k in $\mathbb{Z}_q^n$. The lattice $\mathcal{L}$ obtained by Construction A applied to $\mathscr{C}$ is given by*

$$\mathcal{L}(\mathscr{C}) \stackrel{\triangle}{=} \{\mathbf{x} \in \mathbb{R}^n \ : \ \mathbf{x} = (\mathbf{c} \mod q), \ \mathbf{c} \in \mathscr{C}\}. \tag{30}$$

Clearly finding the closest point in Euclidean distance to some $\mathbf{y} \in \mathbb{Z}_q^n$ in $\mathscr{C}$ also amounts to find the closest lattice point in $\mathcal{L}(\mathscr{C})$ of $\mathbf{y}$. The algorithm for performing this task when $\mathbf{y}$ belongs to $\mathbb{R}^n$ is known as a mean-squared-error (MSE) quantizer for $\mathcal{L}(\mathscr{C})$. To analyze its performance, let $\mathrm{Vor}(\mathcal{L})$ be the fundamental Voronoi region of $\mathcal{L}$, that is $V \stackrel{\triangle}{=} \{\mathbf{v} \in \mathbb{R}^n : \|\mathbf{v}\| \leq \|\mathbf{v} - \mathbf{x}\|, \ \forall \mathbf{x} \in \mathcal{L}\}$. Note that the volume of the Voronoi region V of a lattice $\mathcal{L}(\mathscr{C})$ associated to code $\mathscr{C}$ of dimension k over $\mathbb{Z}_q^n$ is given by $\mathrm{Vol}(V) = q^{n-k}$ [CS88]. The average distance provided by the mean-square quantizer for $\mathcal{L}$ can be assessed by the normalized second moment $G = G(\mathcal{L})$, which is defined as

$$G \stackrel{\triangle}{=} \frac{1}{\mathrm{Vol}(V)^{\frac{2}{n}} \cdot n} \int_V \frac{\|\mathbf{v}\|^2}{\mathrm{Vol}(V)} d\mathbf{v}. \tag{31}$$

It is known that

$$G \geq \frac{1}{2\pi e} \tag{32}$$

and is achieved asymptotically for lattices generated by Construction A from q-ary random codes when q gets big [ZF96]. It really corresponds to the case when the square $\omega^2 \stackrel{\triangle}{=} \int_V \frac{\|\mathbf{v}\|^2}{\mathrm{Vol}(V)} d\mathbf{v}$ of the average decoding distance ω is the lattice analogue of the Gilbert-Varshamov, *i.e.* meets equality in (18) and corresponds therefore to (19). Notice that this normalized second moment is much worse for $\mathbb{Z}^n$, since in this case $G(\mathbb{Z}^n) \approx \frac{1}{12}$ which explains the rather bad performance of modulus switching (20).

3 The algorithm

We assume in the whole section that we deal here with an LWE instance $(\mathbf{A}, \mathbf{b} = \mathbf{As} + \mathbf{e})$ where $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, $\mathbf{b} \in \mathbb{Z}_q^m$, $\mathbf{s} \in \mathbb{Z}_q^n$, *i.e.* we have m LWE samples and the secret is of size n. We assume that the secret distribution χ_s is of variance σ_s^2 whereas the distribution of the error is of variance σ_e^2. We start by giving an overview of the whole algorithm.

Algorithm 2: Skeleton of the dual attack

Input: parameters for Algorithm 3, real numbers $\beta_0,\ \beta_1$, integers R, N, n_{en}, n_{code}, n_{lat}, a set $\text{Bet} \subseteq \mathbb{Z}_q^{n_{\text{en}}}$, and an LWE pair $(\mathbf{A}, \mathbf{b})$.

Choose once for all $I_{\text{lat}} \subset [\![1, n]\!]$ of size n_{lat}

Compute the matrix $\mathbf{B} = \begin{bmatrix} \alpha \mathbf{I}_m & \mathbf{0} \\ \mathbf{A}_{\text{lat}}^T & q\mathbf{I}_{n_{\text{lat}}} \end{bmatrix}$ where $\alpha = \frac{\sigma_e}{\sigma_s}$.

Run Algorithm 1 on $\mathbf{B}$ with parameters β_0, β_1, N to get a list $\mathcal{L}$ of N short vectors.

repeat R **times**

  Choose n_{en} positions among $I \setminus I_{\text{lat}}$ to form I_{en}.

  for every value $\tilde{\mathbf{s}}_{\text{en}} \in \text{Bet}$ **do**

    Check the validity of $\tilde{\mathbf{s}}_{\text{en}}$ with the help of $\mathcal{L}$.

    if $\tilde{\mathbf{s}}_{\text{en}}$ is valid **then**

      return $(\tilde{\mathbf{s}}_{\text{en}}, I_{\text{en}})$

return $\perp$

3.1 Overview of the algorithm

Skeleton of the whole algorithm. The algorithm proceeds by partitioning the set of coordinates $I \stackrel{\triangle}{=} [\![1, n]\!]$ of the secret $\mathbf{s}$ into I_{en}, I_{code} and I_{lat} of respective sizes n_{en}, n_{code} and n_{lat}. The way this is done is explained by the procedure describe by Algorithm 2 which gives the skeleton of the whole algorithm.

Roughly speaking, we perform lattice reduction in order to find short vectors and make then several bets of what the secret $\mathbf{s}$ could be on I_{en}, where the bet Bet is basically a small subset of vectors of very small norm of $\mathbb{Z}_q^{n_{\text{en}}}$. In other words, we check here for R different subsets I_{en} if the the secret $\mathbf{s}$ could be of very small norm on I_{en}. Of course, the choice of R is made in such a way that with constant probability there should be such a subset of positions for which the corresponding $\mathbf{s}_{\text{en}}$ should belong to the restricted set of possibilities Bet. The point is that for most secret distributions of interest the most likely secret error vectors are vectors of very small norm (even if they have an exponentially small probability to happen). Let us explain now the rationale behind checking that a certain choice of $\mathbf{s}_{\text{en}}$ is correct or not.

Checking the validity of the bet on $\tilde{\mathbf{s}}_{\text{en}}$ in its naive version. The columns of $\mathbf{A}$ are also partitioned according to I_{en}, I_{code} and I_{lat} to get $\mathbf{A}_{\text{en}}, \mathbf{A}_{\text{code}}, \mathbf{A}_{\text{lat}}$. It follows that

$$\mathbf{b} \equiv \mathbf{A}_{\text{en}}\mathbf{s}_{\text{en}} + \mathbf{A}_{\text{code}}\mathbf{s}_{\text{code}} + \mathbf{A}_{\text{lat}}\mathbf{s}_{\text{lat}} + \mathbf{e} \mod q. \tag{33}$$

By lattice reduction and sieving, we got a list $\mathcal{L}$ of N pairs of short vectors $(\alpha\mathbf{x}_j, \mathbf{y}_{j,\text{lat}})$ where $\alpha \stackrel{\triangle}{=} \frac{\sigma_e}{\sigma_s}$ and $\mathbf{y}_{j,\text{lat}} \stackrel{\triangle}{=} \mathbf{A}_{\text{lat}}^{\intercal}\mathbf{x}_j$. For all $j \in [\![1, N]\!]\}$, we also let

$$\mathbf{y}_{j,\text{code}} \stackrel{\triangle}{=} \mathbf{A}_{\text{code}}^{\intercal}\mathbf{x}_j, \qquad \mathbf{y}_{j,\text{en}} \stackrel{\triangle}{=} \mathbf{A}_{\text{en}}^{\intercal}\mathbf{x}_j.$$

The verification that we have found the right value for $\mathbf{s}_{\text{en}}$ basically relies on the following observation

$$\underbrace{\langle \mathbf{x}_j, \mathbf{b}\rangle - \langle \mathbf{y}_{j,\text{en}}, \mathbf{s}_{\text{en}}\rangle}_{\text{known}} \equiv \langle \mathbf{y}_{j,\text{code}}, \mathbf{s}_{\text{code}}\rangle + \underbrace{\langle \mathbf{y}_{j,\text{lat}}, \mathbf{s}_{\text{lat}}\rangle + \langle \mathbf{x}_j, \mathbf{e}\rangle}_{\text{modular Gaussian}} \mod q. \quad (34)$$

In other words as explained in the introduction we have now N new LWE samples associated to a secret $\mathbf{s}_{\text{code}} \in \mathbb{Z}_q^{n_{\text{code}}}$ living in a much smaller space. One naive way to perform this task is to compute for all $\mathbf{z} \in \mathbb{Z}_q^{n_{\text{code}}}$ the corresponding vector $\mathbf{t}(\mathbf{z}) = (\{\langle \mathbf{x}_j, \mathbf{b}\rangle - \langle \mathbf{y}_{j,\text{en}}, \mathbf{s}_{\text{en}}\rangle - \langle \mathbf{y}_{j,\text{code}}, \mathbf{z}\rangle\} \mod q)_{1\le j\le N}$ and check whether there is a vector of unusually small norm. This would cost $Nq^{n_{\text{code}}}$. There is a way to amortize these computations with an FFT approach which costs only $q^{n_{\text{code}}+1}\log q$ and which is suggested in [MAT22]. There, the task of verifying whether $\mathbf{s}_{\text{en}} = \tilde{\mathbf{s}}_{\text{en}}$ is performed by computing the real part of the Fourier transform, where $\mathbf{z}$ ranges over $\mathbb{Z}_q^{n_{\text{code}}}$

$$F_{\text{plain}}(\tilde{\mathbf{s}}_{\text{en}}, \mathbf{z}) \stackrel{\triangle}{=} \Re\left(\sum_{j=1}^{N} e^{(\langle \mathbf{x}_j, \mathbf{b}\rangle - \langle \mathbf{y}_{j,\text{en}}, \tilde{\mathbf{s}}_{\text{en}}\rangle - \langle \mathbf{y}_{j,\text{code}}, \mathbf{z}\rangle)\frac{2i\pi}{q}}\right). \quad (35)$$

If we have made the wrong guess, the Fourier transform behaves like those of a uniform distribution, whereas if we made the right guess for $\mathbf{s}_{\text{en}}$, this function of $\mathbf{z}$ should have an unusually high value at $\mathbf{z} = \mathbf{s}_{\text{code}}$. Therefore the distinguisher just relies on the test:
Plain distinguisher test for the guess $\tilde{\mathbf{s}}_{\text{en}}$: Does there exist a value $\mathbf{z} \in \mathbb{Z}_q^{n_{\text{code}}}$ with $F_{\text{plain}}(\tilde{\mathbf{s}}_{\text{en}}, \mathbf{z}) > C$ where C is some well chosen cutoff value?

This approach is justified by the following lemmas, a proof of which is given in the Appendix A.

Lemma 3.1. *Let $(X_j)_{j\in[\![1,N]\!]}$ be N i.i.d random variables drawn according to a modular Gaussian over $\mathbb{Z}_q$ of mean 0 and variance σ^2. Then $\Re\left(\sum_{j=1}^{N} e^{\frac{2i\pi X_j}{q}}\right)$ is approximately normally distributed with*

$$\mathbb{E}\left(\Re\left(\sum_{j=1}^{N} e^{\frac{2i\pi X_j}{q}}\right)\right) \ge Ne^{-2\left(\frac{\pi\sigma}{q}\right)^2} \quad (36)$$

and if $\sigma \ge \sqrt{\frac{q\log(2)}{8\pi^2}}$, then we have

$$\mathbb{V}\left(\Re\left(\sum_{j=1}^{N} e^{\frac{2i\pi X_j}{q}}\right)\right) \le \frac{N}{2}\left(1 + 2e^{-8\left(\frac{\pi\sigma}{q}\right)^2}\right) \quad (37)$$

Lemma 3.2. *Let $(X_j)_{j\in[\![1,N]\!]}$ be N i.i.d random variables drawn according to a uniform distribution over $\mathbb{Z}_q$. Then $\Re\left(\sum_{j=1}^{N} e^{\frac{2i\pi X_j}{q}}\right)$ is approximately normally distributed with*

$$\mathbb{E}\left(\Re\left(\sum_{j=1}^{N} e^{\frac{2i\pi X_j}{q}}\right)\right) = 0 \tag{38}$$

and

$$\mathbb{V}\left(\Re\left(\sum_{j=1}^{N} e^{\frac{2i\pi X_j}{q}}\right)\right) = \frac{N}{2} \tag{39}$$

The problem with this approach is we cannot afford to take large values of n_{code} because of the $\mathcal{O}\big(q^{n_{\text{code}}+1}\log q\big)$ complexity of the whole algorithm and this is unfortunate since we are hardly reducing the dimension of the lattice on which we perform the reduction. [MAT22] circumvents this problem by choosing to perform modulus switching to reduce by a large amount the value of q. Let us explain what this means here.

Modulus switching. Modulus switching basically relies on the following idea. Start again with (34) and rewrite it slightly as an equality over integers, *i.e.* for any $(\alpha\mathbf{x}, \mathbf{y}_{\text{lat}}) \in \mathcal{L}$, there exists $k \in \mathbb{Z}$ such that

$$\langle \mathbf{x}, \mathbf{b}\rangle - \langle \mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}}\rangle = \langle \mathbf{y}_{\text{code}}, \mathbf{s}_{\text{code}}\rangle + e + kq \tag{40}$$

$$e \sim D_{\mathbb{Z}_q,\sigma} \tag{41}$$

We can basically rewrite (34) as an equality of the same type, but this time for another modulo p and with a new noise which is the sum of the rounding noise with the original noise. This can be verified by noticing that the previous equality can be multiplied on both sides by p/q so that

$$\left\langle \frac{p}{q}\mathbf{x}, \mathbf{b}\right\rangle - \left\langle \frac{p}{q}\mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}}\right\rangle = \left\langle \frac{p}{q}\mathbf{y}_{\text{code}}, \mathbf{s}_{\text{code}}\right\rangle + \frac{p}{q}e + kp \tag{42}$$

By writing that $\frac{p}{q}\mathbf{y}_{\text{code}} = \left[\frac{p}{q}\mathbf{y}_{\text{code}}\right] + \left\{\frac{p}{q}\mathbf{y}_{\text{code}}\right\}$ and rearranging terms a little bit we obtain

$$\left\langle \frac{p}{q}\mathbf{x}, \mathbf{b}\right\rangle - \left\langle \frac{p}{q}\mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}}\right\rangle = \left\langle \left[\frac{p}{q}\mathbf{y}_{\text{code}}\right], \mathbf{s}_{\text{code}}\right\rangle + e' + kp \tag{43}$$

$$\text{where} \quad e' \triangleq \left\langle \left\{\frac{p}{q}\mathbf{y}_{\text{code}}\right\}, \mathbf{s}_{\text{code}}\right\rangle + \frac{p}{q}e \tag{44}$$

This can be considered as to be "almost" an LWE problem (mod p), where

$$\mathbf{a} \triangleq \left[\frac{p}{q}\mathbf{y}_{\text{code}}\right] \tag{45}$$

$$b \stackrel{\triangle}{=} \left\langle \frac{p}{q}\mathbf{x}, \mathbf{b} \right\rangle - \left\langle \frac{p}{q}\mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}} \right\rangle \tag{46}$$

$$b = \langle \mathbf{a}, \mathbf{s}_{\text{code}} \rangle + e' + kp \tag{47}$$

What makes this different from an LWE problem, is the fact the b part of the sample $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e')$ is not necessarily an integer anymore. The new rounding term $\left\langle \left\{ \frac{p}{q}\mathbf{y}_{\text{code}} \right\}, \mathbf{s}_{\text{code}} \right\rangle$ has negligible effect when p is close to q, but is significant in the case considered by Matzov, namely when $p \ll q$, because of the rescaling of the samples by the multiplicative factor p/q.

Verifying whether the guess $\tilde{\mathbf{s}}_{\text{en}}$ for $\mathbf{s}_{\text{en}}$ is correct can be done with a similar distinguisher as above, but this time with the Fourier transform being taken over $\mathbb{Z}_p^{n_{\text{code}}}$:

$$F_{\text{mod. switching}}(\tilde{\mathbf{s}}_{\text{en}}, \mathbf{z}) \stackrel{\triangle}{=} \Re \left(\sum_{j=1}^{N} e^{\left(\langle \frac{p}{q}\mathbf{x}_j, \mathbf{b} \rangle - \langle \frac{p}{q}\mathbf{y}_{j,\text{en}}, \tilde{\mathbf{s}}_{\text{en}} \rangle - \langle \left[\frac{p}{q}\mathbf{y}_{j,\text{code}} \right], \mathbf{z} \rangle \right) \frac{2i\pi}{p}} \right). \tag{48}$$

Here $\mathbf{z} \in \mathbb{Z}_p^{n_{\text{code}}}$ and the cost of FFT is only $\mathcal{O}\left(p^{n_{\text{code}}+1} \log p\right)$. This is precisely the [MAT22] approach. We will proceed similarly here, but for completely other reasons. We switch to a modulus $p = 2^s$ which is about the same order as q, in order to get a significant saving in the FFT. However, we do not perform directly the FFT over $\mathbb{Z}_p^{n_{\text{code}}}$ but perform a dimension reduction with a code approach as we now explain.

Reduction of the dimension with a coding approach. The idea is to pick up a code $\mathscr{C}$ of dimension n_{fft} over $\mathbb{Z}_p^{n_{\text{code}}}$ generated by a matrix $\mathbf{G} \in \mathbb{Z}_p^{n_{\text{code}} \times n_{\text{fft}}}$, that is

$$\mathscr{C} \stackrel{\triangle}{=} \left\{ \mathbf{G}\mathbf{u} \ : \ \mathbf{u} \in \mathbb{Z}_p^{n_{\text{fft}}} \right\} \tag{49}$$

such that

(i) we can *decode* efficiently, meaning that for all $\mathbf{y} \in \mathbb{Z}_p^{n_{\text{code}}}$ we are able to find efficiently $\mathbf{u} \in \mathbb{Z}_p^{n_{\text{fft}}}$ such that $\mathbf{G}\mathbf{u}$ is close to $\mathbf{y}$,

(ii) the decoding distance, namely the typical/average Euclidean distance between the $\mathbf{G}\mathbf{u}$ we produce by this decoding algorithm and $\mathbf{y}$ is as small as possible. The product codes used in [BDGL16] or polar codes [Arı09, KU10, Şaş11] are examples where this is possible to achieve.

We use such a code to decode

$$\left[\frac{p}{q}\mathbf{y}_{\text{code}} \right] \equiv \mathbf{G}\mathbf{u}_{\text{code}} + \mathbf{t}_{\text{code}} \mod p \tag{50}$$

where $\mathbf{t}_{\text{code}}$ is of small norm, so that finally, after using this in (47) we obtain

$$\underbrace{\left\langle \frac{p}{q}\mathbf{x}, \mathbf{b} \right\rangle - \left\langle \frac{p}{q}\mathbf{y}_{\text{en}}, \mathbf{s}_{\text{en}} \right\rangle}_{b} = \langle \mathbf{G}\mathbf{u}_{\text{code}} + \mathbf{t}_{\text{code}}, \mathbf{s}_{\text{code}} \rangle + e' + k'p \tag{51}$$

$$= \langle \mathbf{G}\mathbf{u}_{\text{code}}, \mathbf{s}_{\text{code}} \rangle + \langle \mathbf{t}_{\text{code}}, \mathbf{s}_{\text{code}} \rangle + e' + k'p \quad (52)$$

$$= \left\langle \underbrace{\mathbf{u}_{\text{code}}}_{\text{new } \mathbf{a}}, \underbrace{\mathbf{G}^\intercal \mathbf{s}_{\text{code}}}_{\text{new secret}} \right\rangle + \underbrace{\langle \mathbf{t}_{\text{code}}, \mathbf{s}_{\text{code}} \rangle + e'}_{e''} + k'p. \quad (53)$$

In other words, we have here a new "approximate" (because b is not necessarily an integer) LWE problem where the new secret is only of size n_{fft} over $\mathbb{Z}_p$ for which we use now the previous distinguisher, but this time we perform an FFT just over $\mathbb{Z}_p^{n_{\text{fft}}}$:

$$F(\tilde{\mathbf{s}}_{\text{en}}, \mathbf{z}) \stackrel{\triangle}{=} \Re \left(\sum_{j=1}^{N} e^{\left(\langle \frac{p}{q}\mathbf{x}_j, \mathbf{b} \rangle - \langle \frac{p}{q}\mathbf{y}_{j,\text{en}}, \tilde{\mathbf{s}}_{\text{en}} \rangle - \langle \mathbf{u}_{j,\text{code}}, \mathbf{z} \rangle \right) \frac{2i\pi}{p}} \right) \quad (54)$$

where $\mathbf{u}_{j,\text{code}}$ is the result of decoding $\left[\frac{p}{q}\mathbf{y}_{j,\text{code}}\right]$, that is $\left[\frac{p}{q}\mathbf{y}_{j,\text{code}}\right] \equiv \mathbf{G}\mathbf{u}_{j,\text{code}} + \mathbf{t}_{j,\text{code}} \mod p$ where $\mathbf{t}_{j,\text{code}}$ is small.

Wrapping everything up in an algorithm. If we sum up the whole discussion, the distinguisher we use to check the validity of $\tilde{\mathbf{s}}_{\text{en}}$ is given by:
Final distinguisher test for a bet $\tilde{\mathbf{s}}_{\text{en}}$: Does there exist a value $\mathbf{z} \in \mathbb{Z}_p^{n_{\text{fft}}}$ where $F(\tilde{\mathbf{s}}_{\text{en}}, \mathbf{z}) > C$ where C is some well chosen cutoff value?

The corresponding algorithm is given by:

Algorithm 3: Verification of $\tilde{\mathbf{s}}_{\text{en}}$ (Instruction 7 in Algorithm 2)

Input: $\tilde{\mathbf{s}}_{\text{en}} \in \mathbb{Z}_q^{n_{\text{en}}}$, LWE parameters $(n, m, q, \chi_s, \chi_e)$, an LWE pair $(\mathbf{A}, \mathbf{b}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m$ and a list $\mathcal{L}$ of N pairs of short vectors $(\alpha\mathbf{x}, \mathbf{y}_{\text{lat}})$ where $\alpha \stackrel{\triangle}{=} \frac{\sigma_e}{\sigma_s}$ and such that $\mathbf{y}_{\text{lat}} \stackrel{\triangle}{=} \mathbf{A}_{\text{lat}}^\intercal \mathbf{x}$.

Parameters: integers $n_{\text{en}}, n_{\text{fft}}, n_{\text{lat}}$ such that $n_{\text{en}} + n_{\text{code}} + n_{\text{lat}} = n$, an integer $p \stackrel{\triangle}{=} 2^s \leqslant q$, an integer $n_{\text{fft}} \leqslant n_{\text{code}}$, integer N, a real number C

Output: `true` if $\tilde{\mathbf{s}}_{\text{en}}$ is the right bet, `false` otherwise.

Choose a code $\mathbf{G} \in \mathbb{Z}_p^{n_{\text{code}} \times n_{\text{fft}}}$
Initialize a table T of dimensions $\underbrace{p \times p \times \cdots p}_{n_{\text{fft}} \text{ times}}$
for every short vector $(\alpha\mathbf{x}, \mathbf{y}_{\text{lat}})$ in $\mathcal{L}$ **do**
    Compute $\mathbf{y}_{\text{code}} = \mathbf{A}_{\text{code}}^\intercal \mathbf{x}$.
    Compute $\mathbf{y}_{\text{en}} = \mathbf{A}_{\text{en}}^\intercal \mathbf{x}$.
    Decode $\left[\frac{p}{q}\mathbf{y}_{\text{code}}\right]$ as $\mathbf{G}\mathbf{u}_{\text{code}} + \mathbf{t}_{\text{code}}$.
    Add $\exp\left(\frac{2i\pi}{p}\left(\left\langle \frac{p}{q}\mathbf{x}, \mathbf{b} \right\rangle - \left\langle \frac{p}{q}\mathbf{y}_{\text{en}}, \tilde{\mathbf{s}}_{\text{en}} \right\rangle\right)\right)$ to cell $\mathbf{u}_{\text{code}}$ of T.
Perform FFT on T
if the real part of $T[\tilde{\mathbf{s}}_{\text{fft}}^{\mathbf{G}}]$ is larger than C for every $\tilde{\mathbf{s}}_{\text{fft}}^{\mathbf{G}} \in \mathbb{Z}_p^{n_{\text{fft}}}$ **then**
    return `true`.
return `false`

3.2 Correctness of the algorithm

Let us first summarize in Table 1 all parameters of the algorithm and quantities which will appear in this section.

Table 1. Dual attack parameters.

parameters	explanation
n, m, χ_s, χ_e	LWE parameters as in Definition 2.3.
β_0, β_1	BKZ block size β_0 and sieving dimension β_1.
p	modulus switching target modulus ($p \stackrel{\triangle}{=} 2^s \leq q$).
μ	the target success probability $0 < \mu < 1$.
σ_s^2, σ_e^2	variances of χ_s and χ_e respectively.
$\mathbf{s}_{\text{en}}, \mathbf{s}_{\text{code}}, \mathbf{s}_{\text{lat}}$	components of $\mathbf{s}$ covered by exhaustive search, FFT and lattice reduction.
$\tilde{\mathbf{s}}_{\text{en}}, \tilde{\mathbf{s}}_{\text{code}}, \tilde{\mathbf{s}}_{\text{fft}}^{\mathbf{G}}$	guesses for respectively $\mathbf{s}_{\text{en}} \bmod q$, $\mathbf{s}_{\text{code}} \bmod p$ and $\mathbf{s}_{\text{fft}}^{\mathbf{G}} \stackrel{\triangle}{=} \mathbf{G}^{\intercal}\mathbf{s}_{\text{fft}} \bmod p$.
$N_{\text{en}}(\mathbf{s}_{\text{en}})$	number of vector $\tilde{\mathbf{s}}_{\text{en}}$ to enumerate.
$n_{\text{en}}, n_{\text{fft}}, n_{\text{lat}}$	dimension of $\mathbf{s}_{\text{en}}, \mathbf{s}_{\text{code}}$ or $\mathbf{s}_{\text{lat}}$ respectively.
$\mathbf{A}_{\text{en}}, \mathbf{A}_{\text{fft}}, \mathbf{A}_{\text{lat}}$	$\mathbf{A} \cdot \mathbf{s} = \mathbf{A}_{\text{en}} \cdot \mathbf{s}_{\text{en}} + \mathbf{A}_{\text{fft}} \cdot \mathbf{s}_{\text{code}} + \mathbf{A}_{\text{lat}} \cdot \mathbf{s}_{\text{lat}}$.
α	scaling/normalization factor $\alpha \stackrel{\triangle}{=} \sigma_e/\sigma_s$.
$\mathcal{L}, N$	list/number of short vectors returned by sieving oracle $N \coloneqq \sharp\,\mathcal{L}$.
R	the number of different subsets of size n_{en} we test for the bet $\mathbf{s}_{\text{en}} \in \text{Bet}$.
$\psi\left(\mathbf{s}_{\text{code}}\right)$	$= \exp(\frac{2\pi i c_{q'}}{p} \sum_t s_t)$ where $q' = q/\gcd(p,q)$, $c_{q'} := 0$ when q' is odd and $c_{q'} \coloneqq 1/2q'$ when q' is even.
C	FFT cutoff value for scoring function.
Φ	the cumulative distribution function of the standard normal distribution $\Phi(x) \stackrel{\triangle}{=} \frac{1}{2} + \frac{1}{2}\text{erf}\left(\frac{x}{\sqrt{2}}\right)$.
$\phi_{\text{fp}}(\mu)$, $\phi_{\text{fn}}(\mu)$	$= \Phi^{-1}\left(1 - \frac{\mu}{2 \cdot N_{\text{en}}(\mathbf{s}_{\text{en}}) \cdot p^{n_{\text{fft}}}}\right)$, $\Phi^{-1}\left(1 - \frac{\mu}{2}\right)$.
N_{eq}	exponential factor coming from the original LWE error.
N_{round}	exponential factor coming from the rounding when performing modulus switching.
N_{code}	exponential factor coming from the code based treatment.
N_{arg}	$\approx 1/2$, improvement factor coming from considering the complex argument of the Fourier coefficient rather than only the magnitude.
N_{fpfn}	a polynomial factor controlling false positives and negatives.
$\lvert\mathbf{x}\rvert_0$	number of entries of $\mathbf{x}$ equal to 0.

Under assumptions which are similar to Assumption 5.8 in [MAT22] plus additional assumptions on the decoding, we can prove the correctness of our algorithm.

Assumption 3.3

– *All assumptions in [MAT22, Assumption 5.8] hold.*
– *for a uniformly random vector* $\mathbf{y}$*, if we decode it as* $\mathbf{Gu}+\mathbf{t}$ *then* $\mathbf{u}$ *is uniform and* $\mathbf{t}$ *is (approximately) uniform in a ball of radius* d *where* d *is the radius* ω *of a ball attaining equality in* (18)*, that is* $\mathrm{Vol}\left(B_\omega\right) = q^{n_{\mathrm{code}}-n_{\mathrm{fft}}}$*. We call* d *the decoding distance.*
– *The vectors* $\mathbf{t}_{j,\mathrm{code}}$ *are approximately independent from* $\mathbf{x}_j, \mathbf{y}_{j,\mathrm{lat}}, \mathbf{y}_{j,\mathrm{code}}$ *and* $\mathbf{y}_{j,\mathrm{en}}$*.*
– *The vectors* $\mathbf{y}_{j,\mathrm{code}}$ *are approximately uniform modulo* q*.*
– *The coordinates of* $\mathbf{t}_{j,\mathrm{code}}$ *are approximately independent and distributed according to a modular Gaussian of standard deviation* $\frac{d}{\sqrt{n_{\mathrm{code}}}}$ *where* d *is the aforementioned decoding distance of* $\mathbf{G}$*.*

Under these assumptions, plus an additional assumption that we give in the proof which follows (see Assumption 3.6), we give now a theorem proving that when $\mathbf{s}_{\mathrm{en}}$ belongs to the betting set Bet, then the algorithm will return it with a certain probability that we control

Theorem 3.4. *Let* $(n, m, q, \chi_s, \chi_e)$ *be* LWE *parameters, let* $(\beta_0,\ \beta_1,\ n_{\mathrm{en}},\ n_{\mathrm{fft}},\ n_{\mathrm{lat}},\ n_{\mathrm{code}},\ \mathrm{Bet},\ p,\ C,\ D)$ *be a tuple of parameters for Algorithm 3, and let* $0 < \mu < 1$ *be the desired failure probability. Fix* $(\mathbf{s}, \mathbf{e}) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^m$*. If* $\mathbf{s}_{\mathrm{en}} \in \mathrm{Bet}$ *then Algorithm 3 with parameters*

$$N \geqslant N_{\mathrm{eq}} \cdot \Re\left(\psi(\mathbf{s}_{\mathrm{code}})\right)^{-2} N_{\mathrm{round}} \cdot N_{\mathrm{code}} \cdot N_{\mathrm{arg}} \cdot N_{\mathrm{fpfn}} \tag{55}$$

$$C = \phi_{\mathrm{fp}} \sqrt{N \cdot N_{\mathrm{arg}}} \tag{56}$$

returns $\mathbf{s}_{\mathrm{en}}$ *with probability at least* $1-\mu$*, otherwise it returns* $\perp$ *with probability at least* $1 - \mu/2$*. Above, we have*

$$N_{\mathrm{round}}^{-\frac{1}{2}} \triangleq \prod_{\substack{t=1 \\ (\mathbf{s}_{\mathrm{code}})_t \neq 0}}^{n_{\mathrm{code}}} \frac{\sin\left(\frac{\pi(\mathbf{s}_{\mathrm{code}})_t}{p}\right)}{\frac{\pi(\mathbf{s}_{\mathrm{code}})_t}{p}}, \quad N_{\mathrm{eq}}^{-\frac{1}{2}} \triangleq e^{-2\left(\frac{\pi\tau_{\mathrm{eq}}}{q}\right)^2}, \quad \tau_{\mathrm{eq}} \triangleq \frac{\alpha^{-2}\left\|\mathbf{e}\right\|^2 + \left\|\mathbf{s}_{\mathrm{lat}}\right\|^2}{m + n_{\mathrm{lat}}} \ell^2,$$

$$N_{\mathrm{arg}} \triangleq \frac{1}{2}\left(1 + 2e^{-8\left(\frac{\pi\tau_{\mathrm{eq}}}{q}\right)}\right), \quad N_{\mathrm{code}}^{-\frac{1}{2}} \triangleq e^{-2\left(\frac{\pi\tau_{\mathrm{code}}}{p}\right)^2}, \quad \tau_{\mathrm{code}} \triangleq \left\|\mathbf{s}_{\mathrm{code}}\right\| \frac{p^{1-\frac{n_{\mathrm{fft}}}{n_{\mathrm{code}}}}}{\sqrt{2\pi e}},$$

$$N_{\mathrm{fpfn}} = \left(\phi_{\mathrm{fp}} + \phi_{\mathrm{fn}}\right)^2, \quad \phi_{\mathrm{fn}} = \Phi^{-1}\left(1 - \tfrac{\mu}{2}\right), \quad \phi_{\mathrm{fp}} = \Phi^{-1}\left(1 - \tfrac{\mu}{2 \cdot |\,\mathrm{Bet}\,| \cdot p^{n_{\mathrm{fft}}}}\right),$$

$$\psi(\mathbf{s}_{\mathrm{code}}) \triangleq e^{\frac{2i\pi c_{q'}}{p} \sum_{j=1}^{n_{\mathrm{code}}} (\mathbf{s}_{\mathrm{code}})_j}, \quad c_{q'} \triangleq \begin{cases} 0 & \text{if } q' \text{ odd} \\ \frac{1}{2q'} & \text{otherwise} \end{cases}, \quad q' \triangleq \frac{q}{\gcd(p,q)},$$

Ω_s is the universe of χ_s and ℓ is the average length of the vectors given by the short vectors sampling algorithm.

Proof. In this proof, we index j all variables that depend on the short vector $(\mathbf{x}_j, \mathbf{y}_{j,\mathrm{lat}})$, namely $\mathbf{y}_{j,\mathrm{code}}$, $\mathbf{y}_{j,\mathrm{en}}$, $\mathbf{u}_{j,\mathrm{code}}$ and $\mathbf{t}_{j,\mathrm{code}}$. Recall that for every guess $(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}})$, the algorithm computes the sum

$$F(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}}) \triangleq \Re\left(\sum_{j=1}^{N} e^{\left(\langle \frac{p}{q}\mathbf{x}_j, \mathbf{b}\rangle - \langle \frac{p}{q}\mathbf{y}_{j,\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{en}}\rangle - \langle \mathbf{u}_{j,\mathrm{code}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}}\rangle\right)\frac{2i\pi}{p}}\right)$$

via the FFT on T. When $(\tilde{\mathbf{s}}_{\text{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\text{fft}}) = (\mathbf{s}_{\text{en}}, \mathbf{s}^{\mathbf{G}}_{\text{fft}})$, we have

$$F(\mathbf{s}_{\text{en}}, \mathbf{s}^{\mathbf{G}}_{\text{fft}}) = \Re\left(\sum_{j=1}^{N} e^{\left(\langle \mathbf{t}_{\text{code}}, \mathbf{s}_{\text{code}}\rangle + \langle \left\{\frac{p}{q}\mathbf{y}_{j,\text{code}}\right\}, \mathbf{s}_{\text{code}}\rangle + \langle \frac{p}{q}\mathbf{y}_{j,\text{lat}}, \mathbf{s}_{\text{lat}}\rangle + \langle \frac{p}{q}\mathbf{x}_j, \mathbf{e}\rangle\right)\frac{2i\pi}{p}}\right)$$
$$= \sum_{j=1}^{N} \Re\left(\varepsilon_{j,\text{eq}} \cdot \varepsilon_{j,\text{round}} \cdot \varepsilon_{j,\text{code}}\right)$$

where

$$\varepsilon_{j,\text{eq}} = e^{(\langle \mathbf{y}_{j,\text{lat}}, \mathbf{s}_{\text{lat}}\rangle + \langle \mathbf{x}_j, \mathbf{e}\rangle)\frac{2i\pi}{q}}, \qquad \varepsilon_{j,\text{round}} = e^{\left(\langle \left\{\frac{p}{q}\mathbf{y}_{j,\text{code}}\right\}, \mathbf{s}_{\text{code}}\rangle\right)\frac{2i\pi}{p}}$$

and

$$\varepsilon_{j,\text{code}} = e^{(\langle \mathbf{t}_{j,\text{code}}, \mathbf{s}_{\text{code}}\rangle)\frac{2i\pi}{p}}. \tag{57}$$

By Assumption 3.3, those are independent and identically distributed variables so when N is large, the distribution of $F(\mathbf{s}_{\text{en}}, \mathbf{s}^{\mathbf{G}}_{\text{fft}})$ is approximately normal with mean and variance determined by the distribution of $\varepsilon_{j,\text{eq}}, \varepsilon_{j,\text{round}}$ and $\varepsilon_{j,\text{code}}$.

We now remark that the definition of $\varepsilon_{j,\text{eq}}$ and $\varepsilon_{j,\text{round}}$ is exactly the same as in [MAT22], except that we did not include the correcting factor $\frac{1}{\psi(\mathbf{s}_{\text{code}})}$ in $\varepsilon_{j,\text{round}}$, as well as the definition of $\mathbf{x}_j$, $\mathbf{y}_{j,\text{lat}}$ and $\mathbf{y}_{j,\text{code}}$. Therefore we apply the results of [MAT22] directly. Specifically, by [MAT22, Lemma 5.3],

$$\mathbb{E}[\varepsilon_{j,\text{eq}}] \geqslant e^{-2\left(\frac{\pi\tau_{\text{eq}}}{q}\right)^2} \triangleq N_{\text{eq}}^{-\frac{1}{2}}, \qquad \mathbb{E}\left[\varepsilon_{j,\text{eq}}^2\right] \leqslant 2e^{-8\left(\frac{\pi\tau_{\text{eq}}}{q}\right)^2}$$

where $\tau_{\text{eq}} = \frac{\alpha^{-2}\|\mathbf{e}\|^2 + \|\mathbf{s}_{\text{lat}}\|^2}{m+n_{\text{lat}}}\ell^2$ and ℓ is the average length of the vectors given by the short vectors sampling algorithm. Note that $\varepsilon_{j,\text{eq}}$ has real expected value. Also by [MAT22, Lemma 5.4], recall that our $\varepsilon_{j,\text{round}}$ did not include the correcting factor $\frac{1}{\psi(\mathbf{s}_{\text{code}})}$ in $\varepsilon_{j,\text{round}}$,

$$\mathbb{E}\left[\frac{\varepsilon_{j,\text{round}}}{\psi(\mathbf{s}_{\text{code}})}\right] \geqslant \prod_{t=1:\mathbf{s}_t\neq 0}^{n_{\text{fft}}} \frac{\sin\left(\frac{\pi(\mathbf{s}_{\text{code}})_t}{p}\right)}{\frac{\pi(\mathbf{s}_{\text{code}})_t}{p}} \triangleq N_{\text{round}}^{-\frac{1}{2}}.$$

Note in particular that $\frac{\varepsilon_{j,\text{round}}}{\psi(\mathbf{s}_{\text{code}})}$ has real expected value. Therefore,

$$\Re\left(\mathbb{E}[\varepsilon_{j,\text{round}}]\right) = \Re\left(\psi(\mathbf{s}_{\text{code}})\mathbb{E}\left[\frac{\varepsilon_{j,\text{round}}}{\psi(\mathbf{s}_{\text{code}})}\right]\right) \geqslant \Re\left(\psi(\mathbf{s}_{\text{code}})\right) N_{\text{round}}^{-\frac{1}{2}}.$$

Finally, for the lattice-code part, we have the following. (See proof in Appendix B)

Lemma 3.5. $\mathbb{E}[\varepsilon_{j,\text{code}}] \geqslant e^{-2\left(\frac{\pi\tau_{\text{code}}}{p}\right)^2} \triangleq D_{\text{code}}^{-\frac{1}{2}}$ *and* $\mathbb{E}\left[\varepsilon_{j,\text{code}}^2\right] \leqslant 2e^{-8\left(\frac{\pi\tau_{\text{code}}}{p}\right)^2}$ *where* $\tau_{\text{code}} = \|\mathbf{s}_{\text{code}}\| \frac{p^{1-\frac{n_{\text{fft}}}{n_{\text{code}}}}}{\sqrt{2\pi e}}$.

With this lemma, plus the following additional assumption

Assumption 3.6 *$\varepsilon_{j,\mathrm{round}}$ and $\varepsilon_{j,\mathrm{code}}$ are independent.*

we have that

$$\begin{aligned}
\mathbb{E}\left[F(\mathbf{s}_{\mathrm{en}}, \mathbf{s}_{\mathrm{fft}}^{\mathbf{G}})\right] &= \sum_{j=1}^{N} \mathbb{E}[\Re\left(\varepsilon_{j,\mathrm{eq}} \cdot \varepsilon_{j,\mathrm{round}} \cdot \varepsilon_{j,\mathrm{code}}\right)] \\
&= \sum_{j=1}^{N} \Re\left(\mathbb{E}[\varepsilon_{j,\mathrm{eq}}] \cdot \mathbb{E}[\varepsilon_{j,\mathrm{round}}] \cdot \mathbb{E}[\varepsilon_{j,\mathrm{code}}]\right) \\
&= \sum_{j=1}^{N} \mathbb{E}[\varepsilon_{j,\mathrm{eq}}] \cdot \Re\left(\mathbb{E}[\varepsilon_{j,\mathrm{round}}]\right) \cdot \mathbb{E}[\varepsilon_{j,\mathrm{code}}] \\
&\geqslant N \cdot \left(N_{\mathrm{eq}} \cdot \Re\left(\psi(\mathbf{s}_{\mathrm{code}})\right)^{-2} N_{\mathrm{round}} \cdot N_{\mathrm{code}}\right)^{-\frac{1}{2}}
\end{aligned}$$

where we have used that $\varepsilon_{j,\mathrm{eq}}$ and $\varepsilon_{j,\mathrm{code}}$ have real expected value. Recall that this computation was for a correct guess, *i.e.* $(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}}) = (\mathbf{s}_{\mathrm{en}}, \mathbf{s}_{\mathrm{fft}}^{\mathbf{G}})$. On the other hand, if $\tilde{\mathbf{s}}_{\mathrm{en}} \neq \mathbf{s}_{\mathrm{en}}$, *i.e.* an incorrect guess, we write $\tilde{\mathbf{s}}_{\mathrm{en}} = \mathbf{s}_{\mathrm{en}} + \Delta\mathbf{s}_{\mathrm{en}}$ and $\tilde{\mathbf{s}}_{\mathrm{code}} = \mathbf{s}_{\mathrm{code}} + \Delta\mathbf{s}_{\mathrm{code}}$. It follows that

$$\begin{aligned}
&F(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}}) \\
&= \Re\left(\sum_{j=1}^{N} e^{\left(\langle \frac{p}{q}\mathbf{x}_j, \mathbf{b}\rangle - \langle \frac{p}{q}\mathbf{y}_{j,\mathrm{en}}, (\mathbf{s}_{\mathrm{en}} + \Delta\mathbf{s}_{\mathrm{en}})\rangle - \langle \mathbf{u}_{j,\mathrm{code}}, (\tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}} + \mathbf{G}^{\intercal}\Delta\mathbf{s}_{\mathrm{code}})\rangle\right)\frac{2i\pi}{p}}\right) \\
&= \sum_{j=1}^{N} \Re\left(e^{\left(\langle \mathbf{t}_{\mathrm{code}}, \mathbf{s}_{\mathrm{code}}\rangle + \langle \left\{\frac{p}{q}\mathbf{y}_{j,\mathrm{code}}\right\}, \mathbf{s}_{\mathrm{code}}\rangle + \langle \frac{p}{q}\mathbf{y}_{j,\mathrm{lat}}, \mathbf{s}_{\mathrm{lat}}\rangle + \langle \frac{p}{q}\mathbf{x}_j, \mathbf{e}\rangle - \langle \frac{p}{q}\mathbf{y}_{j,\mathrm{en}}, \Delta\mathbf{s}_{\mathrm{en}}\rangle - \langle \mathbf{u}_{j,\mathrm{code}}, \mathbf{G}^{\intercal}\Delta\mathbf{s}_{\mathrm{code}}\rangle\right)\frac{2i\pi}{p}}\right) \\
&= \sum_{j=1}^{N} \Re\left(e^{-(\langle \mathbf{y}_{j,\mathrm{en}}, \Delta\mathbf{s}_{\mathrm{en}}\rangle)\frac{2i\pi}{q}} e^{2i\pi W_j}\right)
\end{aligned}$$

where W_j is independent of $\mathbf{y}_{j,\mathrm{en}}$ by Assumption 3.3. Furthermore, by Assumption 3.3, $\mathbf{y}_{j,\mathrm{en}}$ is uniform mod q and $\Delta\mathbf{s}_{\mathrm{en}}$ is nonzero so $\mathbf{y}_{j,\mathrm{en}}^{T}\Delta\mathbf{s}_{\mathrm{en}}$ is uniform mod q, hence has expected value 0. This shows that

$$\mathbb{E}\left[F(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}})\right] = 0 \qquad \text{when } \tilde{\mathbf{s}}_{\mathrm{en}} \neq \mathbf{s}_{\mathrm{en}}.$$

We have shown that $\mathbb{E}\left[F(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}})\right]$ is zero for wrong guesses and bounded away from zero for good guesses. We now need to study the variance of this quantity to quantify how many samples we need to distinguish between those two cases. Let $\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}}$ be arbitrary. Again write $\tilde{\mathbf{s}}_{\mathrm{en}} = \mathbf{s}_{\mathrm{en}} + \Delta\mathbf{s}_{\mathrm{en}}$ and $\tilde{\mathbf{s}}_{\mathrm{fft}} = \mathbf{s}_{\mathrm{code}} + \Delta\mathbf{s}_{\mathrm{code}}$ for some (possibly zero) $\Delta\mathbf{s}_{\mathrm{en}}$ and $\Delta\mathbf{s}_{\mathrm{code}}$. The computation above shows that

$$F(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}})$$

$$= \Re\left(\sum_{j=1}^{N} e^{\left(\langle \mathbf{t}_{\text{code}}, \mathbf{s}_{\text{code}}\rangle + \langle \left\{\frac{p}{q}\mathbf{y}_{j,\text{code}}\right\}, \mathbf{s}_{\text{code}}\rangle + \langle \frac{p}{q}\mathbf{y}_{j,\text{lat}}, \mathbf{s}_{\text{lat}}\rangle + \langle \frac{p}{q}\mathbf{x}_j, \mathbf{e}\rangle - \langle \frac{p}{q}\mathbf{y}_{j,\text{en}}, \Delta\mathbf{s}_{\text{en}}\rangle - \langle \mathbf{u}_{j,\text{code}}, \mathbf{G}^{\intercal}\Delta\mathbf{s}_{\text{code}}\rangle\right)\frac{2i\pi}{p}}\right)$$

$$= \sum_{j=1}^{N} \Re\left(e^{-\left(\langle \frac{p}{q}\mathbf{y}_{j,\text{lat}}, \mathbf{s}_{\text{lat}}\rangle + \langle \frac{p}{q}\mathbf{x}_j, \mathbf{e}\rangle\right)\frac{2i\pi}{q}} e^{2i\pi Z_j}\right)$$

$$= \sum_{j=1}^{N} \Re\left(\varepsilon_{j,\text{eq}} \cdot e^{2i\pi Z_j}\right)$$

where Z_j is independent of $\varepsilon_{j,\text{eq}}$ by Assumption 3.3. Furthermore, note that Z_j is a real number. Therefore,

$$\begin{aligned}
\text{Var}\left(F(\tilde{\mathbf{s}}_{\text{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\text{fft}})\right) &= \sum_{j=1}^{N} \text{Var}\left(\Re\left(\varepsilon_{j,\text{eq}} \cdot e^{2i\pi Z_j}\right)\right) \\
&\leqslant \sum_{j=1}^{N} \mathbb{E}\left[\Re\left(\varepsilon_{j,\text{eq}} \cdot e^{2i\pi Z_j}\right)^2\right] \\
&= \sum_{j=1}^{N} \mathbb{E}\left[\left(\frac{\varepsilon_{j,\text{eq}} \cdot e^{2i\pi Z_j} + \overline{\varepsilon_{j,\text{eq}}} \cdot e^{-2i\pi Z_j}}{2}\right)^2\right] \\
&= \sum_{j=1}^{N} \mathbb{E}\left[\frac{\varepsilon^2_{j,\text{eq}} \cdot e^{4i\pi Z_j} + 2 + \overline{\varepsilon^2_{j,\text{eq}}} \cdot e^{-4i\pi Z_j}}{4}\right] \\
&= \sum_{j=1}^{N} \frac{1}{2}\left(1 + \Re\left(\mathbb{E}\left[\varepsilon^2_{j,\text{eq}}\right]\mathbb{E}\left[e^{4i\pi Z_j}\right]\right)\right) \\
&\leqslant \sum_{j=1}^{N} \frac{1}{2}\left(1 + \left|\mathbb{E}\left[\varepsilon^2_{j,\text{eq}}\right]\right| \cdot \left|\mathbb{E}\left[e^{4i\pi Z_j}\right]\right|\right) \\
&\leqslant \sum_{j=1}^{N} \frac{1}{2}\left(1 + 2e^{-8\left(\frac{\pi\tau_{\text{eq}}}{q}\right)}\right) \qquad \text{since } |e^{4i\pi Z_j}| = 1 \\
&= N \cdot N_{\text{arg}}
\end{aligned}$$

where $N_{\text{arg}} = \frac{1}{2}\left(1 + 2e^{-8\left(\frac{\pi\tau_{\text{eq}}}{q}\right)}\right)$. In summary, we have shown about $F(\tilde{\mathbf{s}}_{\text{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\text{fft}})$ that:

- for an incorrect guess $\tilde{\mathbf{s}}_{\text{en}} \neq \mathbf{s}_{\text{en}}$, it has expected value 0,
- for a correct guess $(\tilde{\mathbf{s}}_{\text{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\text{fft}}) = (\mathbf{s}_{\text{en}}, \mathbf{s}^{\mathbf{G}}_{\text{fft}})$, it has expected value at least

$$D \cdot \left(D_{\text{eq}} \cdot \Re\left(\psi(\mathbf{s}_{\text{code}})\right) D_{\text{round}} \cdot D_{\text{code}}\right)^{-\frac{1}{2}}$$

- it has variance bounded by $D \cdot D_{\text{arg}}$.

Furthermore, $F(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\mathrm{fft}})$ is the sum of many independent and identically distributed variables and thus approximately distributed according to a normal distribution with mean and variance described just above. As a result, we have (for good guesses) that

$$\begin{aligned}
&\Pr\left[F(\mathbf{s}_{\mathrm{en}}, \mathbf{s}^{\mathbf{G}}_{\mathrm{fft}}) \leqslant C\right] \\
&\leqslant \Pr\left[\mathcal{N}(N \cdot (N_{\mathrm{eq}} \cdot \Re\left(\psi(\mathbf{s}_{\mathrm{code}})\right) N_{\mathrm{round}} \cdot N_{\mathrm{code}})^{-\frac{1}{2}}, N \cdot N_{\mathrm{arg}}) \leqslant C\right] \\
&= 1 - \Phi\left(\frac{N \cdot (N_{\mathrm{eq}} \cdot \Re\left(\psi(\mathbf{s}_{\mathrm{code}})\right) N_{\mathrm{round}} \cdot N_{\mathrm{code}})^{-\frac{1}{2}} - C}{\sqrt{N \cdot N_{\mathrm{arg}}}}\right) \\
&\leqslant 1 - \Phi\left(\sqrt{N_{\mathrm{fpfn}}} - \frac{C}{\sqrt{N \cdot N_{\mathrm{arg}}}}\right) \\
&= 1 - \Phi(\phi_{\mathrm{fn}}) \\
&= \frac{\mu}{2}
\end{aligned}$$

where we have used our assumption that $N \geqslant N_{\mathrm{eq}} \cdot \Re\left(\psi(\mathbf{s}_{\mathrm{code}})\right)^{-2} N_{\mathrm{round}} \cdot N_{\mathrm{code}} \cdot N_{\mathrm{arg}} \cdot N_{\mathrm{fpfn}}$ and the last two equalities are exactly the same as in [MAT22, Lemma 5.7]. On the other hand, for a bad guess $(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\mathrm{fft}})$ with $\tilde{\mathbf{s}}_{\mathrm{en}} \neq \mathbf{s}_{\mathrm{en}}$, the mean is zero so

$$\begin{aligned}
\Pr\left[F(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\mathrm{fft}}) > C\right] &\leqslant \Pr[\mathcal{N}(0, N \cdot N_{\mathrm{arg}}) > C] \\
&= 1 - \Phi\left(\frac{C}{\sqrt{N \cdot N_{\mathrm{arg}}}}\right) \\
&= 1 - \Phi(\phi_{\mathrm{fp}}) \\
&= \frac{\mu}{2 \cdot |\,\mathrm{Bet}\,| \cdot p^{n_{\mathrm{fft}}}}
\end{aligned}$$

again by the same computation as in [MAT22, Lemma 5.7].

Assume that $\mathbf{s}_{\mathrm{en}} \in \mathrm{Bet}$, *i.e.* that the code will eventually consider a good guess $(\mathbf{s}_{\mathrm{en}}, \mathbf{s}^{\mathbf{G}}_{\mathrm{fft}})$. Then for the algorithm to return a wrong value, it must either fail to pass the check when considering $(\mathbf{s}_{\mathrm{en}}, \mathbf{s}^{\mathbf{G}}_{\mathrm{fft}})$, or wrongly accepts a bad guess $(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\mathrm{fft}})$ that appears before $(\mathbf{s}_{\mathrm{en}}, \mathbf{s}^{\mathbf{G}}_{\mathrm{fft}})$ in the enumeration. In total, the algorithm considers at most $|\,\mathrm{Bet}\,| \cdot p^{n_{\mathrm{fft}}}$ pairs $(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\mathrm{fft}})$. Therefore, by a union bound, we have

$$\Pr[\text{the algorithm does not return } \mathbf{s}_{\mathrm{en}}] \leqslant \frac{\mu}{2} + \frac{\mu \cdot |\,\mathrm{Bet}\,| \cdot p^{n_{\mathrm{fft}}}}{2 \cdot |\,\mathrm{Bet}\,| \cdot p^{n_{\mathrm{fft}}}} = \mu.$$

Now assume that $\mathbf{s}_{\mathrm{en}} \in \mathrm{Bet}$ does not hold. Then the algorithm should return $\perp$ and all pairs $(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}^{\mathbf{G}}_{\mathrm{fft}})$ should fail. Therefore the algorithm fails if it wrongly accepts a bad guess. In total, the algorithm considers at most $|\,\mathrm{Bet}\,| \cdot p^{n_{\mathrm{fft}}}$ pairs

$(\tilde{\mathbf{s}}_{\mathrm{en}}, \tilde{\mathbf{s}}_{\mathrm{fft}}^{\mathbf{G}})$. Therefore, by a union bound, we have

$$\Pr[\text{the algorithm does not return } \perp] \leqslant \frac{\mu \cdot |\operatorname{Bet}| \cdot p^{n_{\mathrm{fft}}}}{2 \cdot |\operatorname{Bet}| \cdot p^{n_{\mathrm{fft}}}} = \frac{\mu}{2}.$$

□

To analyze now the whole algorithm, the following lemma analyzing the whole success probability over the R iterations of Algorithm 2 will be helpful.

Lemma 3.7. *Let $(n, m, q, \chi_s, \chi_e)$ be* LWE *parameters, let (β_0, β_1, n_{en}, n_{fft}, n_{lat}, n_{code},* Bet*, p, C, N, R) be a tuple of parameters for Algorithm 2, and let $0 < \nu < 1$ be the desired failure probability. Fix $(\mathbf{s}, \mathbf{e}) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^m$. Let $\mathcal{T}(\mathbf{s}) = \{\, \text{permutation } \tau \text{ of } [\![1, n_{\mathrm{en}} + n_{\mathrm{code}}]\!] : \mathbf{s}_{\mathrm{en}}^{\tau} \in \operatorname{Bet} \,\}$. We assume that we have chosen these parameters so that if for one of the R iterations of Algorithm 2 (that is Instruction 6) we have $\mathbf{s}_{\mathrm{en}} \in \operatorname{Bet}$, then the probability that $\mathbf{s}_{\mathrm{en}}$ is output by Algorithm 3 is at least $1 - \mu$. If $|\mathcal{T}| > 0$ then Algorithm 2 with parameters as in Theorem 3.4 and*

$$R \geqslant \frac{\ln(\frac{\nu}{2})}{\ln(1 - p_\tau) + \ln(1 - \mu)}, \qquad \mu \leqslant \frac{\nu p_\tau}{\nu p_\tau + 2 - \nu}, \qquad p_\tau \triangleq \frac{|\mathcal{T}(\mathbf{s})|}{(n_{\mathrm{en}} + n_{\mathrm{code}})!}$$

returns $\mathbf{s}_{\mathrm{en}}^{\tau}$ with probability at least $1 - \nu$. If $|\mathcal{T}| = 0$ then Algorithm 2 with parameters as in Theorem 3.4 and

$$R \leqslant \frac{\ln(1 - \frac{\nu}{2})}{\ln(1 - \mu)}, \qquad \mu \text{ arbitrary}$$

returns $\perp$ with probability at least $1 - \frac{\nu}{2}$.

Proof. Assume that $|\mathcal{T}| > 0$. Then at each round of the algorithm, the random permutation τ has a probability p_τ to be such that $\mathbf{s}_{\mathrm{en}}^{\tau} \in \operatorname{Bet}$. By Theorem 3.4, when this is the case, the algorithm will return $\mathbf{s}_{\mathrm{en}}^{\tau}$ with probability at least $1 - \mu$. When this is not the case, the algorithm will return $\perp$ with probability at least $1 - \mu$ and continue to the next round. Therefore the probability that the algorithm returns the correct value within R iterations is at least

$$\begin{aligned} p_{\mathrm{succ}} &\triangleq \sum_{k=0}^{R-1} \left(\underbrace{\overbrace{(1 - p_\tau)}^{\text{bad } \tau \text{ go to next iteration}} (1 - \mu)}_{:= \alpha} \right)^k \overbrace{p_\tau (1 - \mu)}^{\text{good } \tau \text{ pass}} \\ &= p_\tau (1 - \mu) \frac{1 - \alpha^R}{1 - \alpha} \\ &= \frac{p_\tau (1 - \mu)}{p_\tau (1 - \mu) + \mu} (1 - \alpha^R). \end{aligned}$$

Let $x = \frac{\nu}{2 - \nu}$ and assume that $\mu \leqslant \frac{x p_\tau}{1 + x p_\tau} = \frac{\nu p_\tau}{\nu p_\tau + 2 - \nu}$. Then a straightforward calculation shows that $\frac{p_\tau (1 - \mu)}{p_\tau (1 - \mu) + \mu} \geqslant 1 - \frac{\nu}{2}$. It follows that

$$p_{\mathrm{succ}} \geqslant (1 - \tfrac{\nu}{2}) \left(1 - x^R\right) \geqslant (1 - \tfrac{\nu}{2})(1 - \tfrac{\nu}{2}) \geqslant 1 - \nu$$

when $R \geqslant \frac{\ln(\frac{\nu}{2})}{\ln(1-p_\tau)+\ln(1-\mu)}$.

Assume now that $|\mathcal{T}| = 0$, then the algorithm should return $\perp$. For this to happen, all the calls to Algorithm 3 must return $\perp$ at each of the R iterations. By Theorem 3.4, since the condition $\mathbf{s}_{\text{en}}^{\tau} \in \text{Bet}$ is never satisfied, Algorithm 3 returns $\perp$ with probability at least $1-\mu$. Therefore, the algorithm will correctly return $\perp$ with probability at least

$$(1-\mu)^R = e^{R\ln(1-\mu)} \geqslant 1 - \tfrac{\nu}{2}$$

when $R \leqslant \frac{\ln(1-\frac{\nu}{2})}{\ln(1-\mu)}$. □

Finally, if we put everything together, we obtain the following result analyzing the success probability of Algorithm 2. We give a proof in the Appendix C for reference.

Theorem 3.8. *Let $(n, m, q, \chi_s, \chi_e)$ be* LWE *parameters, (β_0, β_1, n_{en}, n_{fft}, n_{lat}, n_{code}, Bet, p, C, N, R, μ) be a partial tuple of parameters and $0 < \nu < 1$ be the desired failure probability. Then Algorithm 2 with parameters*

$$\begin{aligned}
N &\geqslant \tilde{N}_{\text{eq}} \cdot \tilde{N}_{\text{round}} \cdot \tilde{N}_{\text{code}} \cdot \tilde{N}_{\text{arg}} \cdot \tilde{N}_{\text{fpfn}},\\
C &= \tilde{\phi}_{\text{fp}} \sqrt{D \cdot \tilde{N}_{\text{arg}}},\\
R &\geqslant \frac{\ln(\frac{\nu}{2})}{\ln(1-\tilde{p}_\tau)+\ln(1-\mu)},\\
\mu &\leqslant \frac{\nu\tilde{p}_\tau}{\nu\tilde{p}_\tau + 2 - \nu}
\end{aligned}$$

returns $\mathbf{s}_{\text{en}}$ with probability at least $\frac{1-\nu}{4}$. Above, we have

$$\tilde{N}_{\text{round}} \triangleq \prod_{\bar{s}\neq 0} \left(\frac{\sin\left(\frac{\pi\bar{s}}{p}\right)}{\frac{\pi\bar{s}}{p}} \right)^{-2n_{\text{fft}}\chi_s(\bar{s})}, \qquad \tilde{N}_{\text{eq}} \triangleq e^{4\left(\frac{\pi\sigma_s\ell}{q}\right)^2}, \qquad \tilde{N}_{\text{arg}} \approx \frac{1}{2},$$

$$\tilde{N}_{\text{code}} \triangleq \exp\left(2\frac{\pi\sigma_s^2}{e} n_{\text{code}} \cdot p^{-\frac{n_{\text{fft}}}{n_{\text{code}}}}\right),$$

$$\tilde{N}_{\text{fpfn}} = \left(\tilde{\phi}_{\text{fp}} + \tilde{\phi}_{\text{fn}}\right)^2, \quad \tilde{\phi}_{\text{fn}} = \Phi^{-1}\left(1 - \tfrac{\mu}{2}\right), \quad \tilde{\phi}_{\text{fp}} = \Phi^{-1}\left(1 - \tfrac{\mu}{2\cdot|\text{Bet}|\cdot p^{n_{\text{fft}}}}\right),$$

$$\tilde{p}_\tau \text{ such that } \Pr_{\mathbf{s}}[p_\tau(\mathbf{s}) \geqslant \tilde{p}_\tau] \geqslant 3/4, \qquad p_\tau(\mathbf{s}) = \frac{|\mathcal{T}(\mathbf{s})|}{(n_{\text{en}}+n_{\text{code}})!}$$

$$\mathcal{T}(\mathbf{s}) = \{ \text{ permutation } \tau \text{ of } [\![1, n_{\text{en}}+n_{\text{code}}]\!] : \mathbf{s}_{\text{en}}^{\tau} \in \text{Bet} \}$$

Ω_s is the universe of χ_s and ℓ is the average length of the vectors given by the short vectors sampling algorithm.

3.3 Which Codes Should We Use?

In terms of the decoding distance alone, the answer would be, just use a random code of dimension n_{fft} in $\mathbb{Z}_p^{n_{\text{code}}}$. In this case, we would obtain the decoding distance $d \approx p^{1-\frac{n_{\text{fft}}}{n_{\text{code}}}} \sqrt{\frac{n_{\text{code}}}{2\pi e}}$ (see (19)) attaining the bound (32) or (18). However, the decoding algorithm we could use in this case would be too complex for our purpose. We could instead use a product code structure as in [BDGL16]. However, contrary to what happens in the latter case, where spherical codes can be used, we are in a situation where more stuctured codes could do the job better. A natural answer that comes to mind, would be to use polar codes [Arı09] for this purpose. When $p = 2$, such codes are known to be asymptotically optimal [KU10] for lossy source coding (basically a version of our problem where the alphabet is binary and we take the Hamming distance instead of the Euclidean distance). Such a result is expected to carry over for other prime size alphabets, basically because

- The proof that polar codes are optimal for this problem relies on the same crucial argument (namely the proof of the polarization phenomenon for the successive cancellation decoder) as proving that polar codes attain the capacity of any symmetric binary input discrete memoryless channel.
- Non-binary polar codes defined over any prime alphabets have been found out to attain capacity (see for instance [Şaş11, Ch.5]).

Remarkably, the construction of polar codes generalizes trivially to other alphabets, be it prime or not, be it a finite field or a ring. In particular, it applies to $\mathbb{Z}_p$ where $p \stackrel{\triangle}{=} 2^s$. Moreover, the successive cancellation decoder which is used to decode them in the error correction scenario can be turned with a simple modification into an algorithm for finding a close codeword (but not necessarily the closest one) [KU10]. This is precisely what is needed in our context. It needs a noise model to instantiate it, and this can be done for our Euclidean metric by using the Gaussian noise model. Essentially, the complexity of this type of decoding is of order $n_{\text{code}} \log_2 n_{\text{code}}$ times twice the complexity of the FFT over $\mathbb{Z}_p$, because the most consuming part of the algorithm consists of computing the convolution of two probability distributions over $\mathbb{Z}_p$ and this $n_{\text{code}} \log_2 n_{\text{code}}$ times. This explains the optimistic cost we give for decoding in Subsection 3.4. namely $T_{\text{dec}}(N, p, n) = 3 \cdot C_{\text{mul}} \cdot N \cdot p \log_2(p) \cdot n \log_2(n)$.

In Appendix D, we give more details about our polar code construction over $\mathbb{Z}_p$. We also give some experimental results that show polar codes are perfectly suitable in our case.

3.4 Complexity of Algorithm 2

Theorem 3.9. *Let* $(n, m, q, \chi_s, \chi_e)$ *be* LWE *parameters,* $(\beta_0, \beta_1, n_{\text{en}}, n_{\text{fft}}, n_{\text{lat}}, n_{\text{code}}, \text{Bet}, p)$ *be a partial tuple of parameters for Algorithm 2, and let* $0 < \nu < 1$. *Choosing the parameters* C, N, R, μ *according to Theorem 3.8, Algorithm 2*

outputs $\mathbf{s}_{\mathrm{en}}^{\tau}$ *with probability at least* $1-\nu$ *in expected time*

$$O\left(T_{\mathrm{samp}}\left(m+n_{\mathrm{lat}},\beta_0,\beta_1,N\right)+R|\operatorname{Bet}|\left(T_{\mathrm{fft}}\left(p,n_{\mathrm{code}}\right)+T_{\mathrm{tab}}\left(N\right)+T_{\mathrm{dec}}\left(N,p,n_{\mathrm{fft}}\right)\right)\right)$$

where

- $T_{\mathrm{samp}}\left(d,\beta_0,\beta_1,N\right)$ *is the running time for outputting at least* N *short vectors (see Lemma 2.1),*
- $T_{\mathrm{fft}}(2^u,d)=C_{\mathrm{mul}}\cdot d\cdot u\cdot 2^{d\cdot u}$ *is the cost of performing the FFT over* $\mathbb{Z}/2^u\mathbb{Z}$ *on a table of dimension* $(2^u)^d$ *(see [Knu97, pp. 306-311]),*
- $T_{\mathrm{tab}}\left(N\right)\stackrel{\triangle}{=}4\cdot C_{\mathrm{add}}\cdot N$ *is the table generation time,*
- $T_{\mathrm{dec}}\left(N,p,n\right)=3\cdot C_{\mathrm{mul}}\cdot N\cdot p\log_2(p)\cdot n\log_2(n)$ *is the cost of decoding* N *word in a polar code of length* n *over* $\mathbb{Z}_p$*,*
- C_{add} *and* C_{mul} *are respectively the cost of an addition and a multiplication (with wordsize of 4 bytes,* $C_{\mathrm{add}}=5\cdot 32$ *and* $C_{\mathrm{mul}}=32^2$*).*

3.5 Optimal Betting Strategy

The optimal betting strategy is in our case straightforward to derive since here betting is just used to decrease the cost N_{en} of enumerating all likely $\mathbf{s}_{\mathrm{en}}$ in $\mathbb{Z}_q^{n_{\mathrm{en}}}$. For $\mathbf{x}\in\mathbb{Z}_q^{n_{\mathrm{en}}}$, let $p(\mathbf{x})\stackrel{\triangle}{=}\Pr[\mathbf{s}_{\mathrm{en}}=\mathbf{x}]$ and denote accordingly the probability of a subset $\mathcal{E}$ of $\mathbb{Z}_q^{n_{\mathrm{en}}}$: $p(\mathcal{E})\stackrel{\triangle}{=}\sum_{\mathbf{x}\in\mathcal{E}}p(\mathbf{x})$. Let us assume that we strive for a probability of say $\frac{1}{2}$ of finding the right $\mathbf{s}_{\mathrm{en}}$ with this approach. The number R of times we have to check whether for the I_{en} we have chosen $\mathbf{s}_{\mathrm{en}}$ is in Bet should satisfy $\sum_{i=1}^{R}(1-\pi)^{i-1}\pi=\frac{1}{2}$ where $\pi=p(\mathrm{Bet})$. Since $\sum_{i=1}^{R}(1-\pi)^{i-1}\pi=\pi\frac{1-(1-\pi)^R}{\pi}=1-(1-\pi)^R$, we deduce that we should have $(1-\pi)^R=\frac{1}{2}$, that is $R=-\frac{\ln 2}{\ln(1-\pi)}\approx\frac{\ln 2}{\pi}$, when we assume that π is small. The complexity of finding $\mathbf{s}_{\mathrm{en}}$ with probability $\frac{1}{2}$ for the betting strategy is therefore $R|\operatorname{Bet}|\approx\frac{\ln 2|\operatorname{Bet}|}{p(\mathrm{Bet})}$. The ensemble Bet which minimizes this complexity is therefore necessarily the set (or a subset of it) of all $\mathbf{x}$ of maximal probability. In the case of the distributions χ_s of $\mathbf{s}$ we have examined, this was always given by the single all-zero vector: $\mathrm{Bet}_{\mathrm{optim}}=\{\mathbf{0}\}$.

4 Application

In this section we give estimates for the impact of our algorithm on the cost of solving lattice parameters from the literature. In particular, we consider NIST PQC Round 3 candidate Saber [DKR+20] and NIST PQC to-be-standardised candidate Kyber [SAB+20].

The cost given in Theorem 3.9 is the sum of two costs: lattice reduction and the search for $\mathbf{s}_{\mathrm{en}}$. The optimal cost is obtained by balancing the two summands. For the first summand, various cost models are available which we describe below.

Table 2. The $\log_2$ complexity of original Matzov

Scheme	C0	CC	CN
Kyber 512	114.8	138.5	133.7
Kyber 768	173.1	195.7	190.4
Kyber 1024	240.7	261.4	255.4
LightSaber	113.1	137.1	132.3
Saber	178.3	201.1	195.1
FireSaber	242.8	263.6	257.7

Table 3. The $\log_2$ complexity of our method

without using the Prange bet

Scheme	C0	CC	CN
Kyber 512	114.0	137.8	133.0
Kyber 768	170.2	192.5	187.2
Kyber 1024	235.7	256.2	250.5
LightSaber	112.3	136.8	131.5
Saber	177.0	199.7	194.9
FireSaber	239.4	259.9	254.4

using the Prange bet

Scheme	C0	CC	CN
Kyber 512	113.9	137.5	132.6
Kyber 768	169.8	191.9	186.7
Kyber 1024	235.5	255.5	249.5
LightSaber	112.2	136.7	131.8
Saber	176.9	199.0	193.8
FireSaber	239.0	259.3	253.9

Table 4. Parameters

	Scheme	Attack	m	β_0	β_1	n_{en}	n_{fft}	n_{code}	p
C0:	Kyber 512	113.9	501	390	390	5	14	42	64
	Kyber 768	169.8	665	581	581	11	19	76	128
	Kyber 1024	235.5	844	806	806	12	22	106	512
	LightSaber	112.2	535	384	384	4	12	34	128
	Saber	176.9	731	605	605	9	23	58	64
	FireSaber	239	898	818	818	11	33	101	64

	Scheme	Attack	m	β_0	β_1	n_{en}	n_{fft}	n_{code}	p
CC:	Kyber 512	137.5	491	379	383	10	14	45	128
	Kyber 768	191.9	658	573	567	13	19	83	256
	Kyber 1024	255.5	839	799	781	13	31	124	128
	LightSaber	136.7	527	376	381	8	12	39	256
	Saber	199	726	599	591	11	22	67	128
	FireSaber	259.3	894	813	794	13	24	109	512

	Scheme	Attack	m	β_0	β_1	n_{en}	n_{fft}	n_{code}	p
CN:	Kyber 512	132.6	493	381	385	11	9	43	1024
	Kyber 768	186.7	660	575	568	12	15	89	1024
	Kyber 1024	249.5	841	801	783	14	19	126	2048
	LightSaber	131.8	530	378	382	8	9	35	1024
	Saber	193.8	728	601	594	12	16	62	512
	FireSaber	253.9	896	816	798	10	36	102	64

CC Cost estimates in a classical circuit model [AGPS20, SAB+20, MAT22] for Algorithm 1 using [BDGL16] as the sieving oracle. We derive these estimates by implementing the cost estimates from [MAT22], those tagged "asymptotic" cf. [MAB+22]. This is the most detailed cost estimate available in the literature. However, we caution that these estimates, too, ignore the cost of memory access and thus may significantly underestimate the true cost. That is RAM access is not "free", cf. [MAB+22]. This cost model is called "list_decoding-classical" in [AGPS20].

CN Cost estimates in a query model for Algorithm 1 using [BDGL16] as the sieving oracle. We include this cost model for completeness. This cost model is called "list_decoding-naive_classical" in [AGPS20].

C0 Cost estimates in the "Core-SVP" cost model [ADPS16] for Algorithm 1 using [BDGL16] as the sieving oracle. This model assumes a single SVP call suffices to reduce a lattice. It furthermore assumes that all lower-order terms in the exponent are zero.

We give the source code for and results of the comparison in Appendix E.

References

ACPS09. Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 595–618. Springer, Heidelberg, August 2009.

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 Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 717–746, Cham, 2019. Springer International Publishing.

ADPS16. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 327–343. USENIX Association, August 2016.

AFFP14. Martin R. Albrecht, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Lazy modulus switching for the BKW algorithm on LWE. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 429–445. Springer, Heidelberg, March 2014.

AGPS20. Martin R. Albrecht, Vlad Gheorghiu, Eamonn W. Postlethwaite, and John M. Schanck. Estimating quantum speedups for lattice sieves. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 583–613. Springer, Heidelberg, December 2020.

Alb17. Martin R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 103–129. Springer, Heidelberg, April / May 2017.

APS15. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

Arı09. Erdal Arıkan. Channel polarization: a method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Trans. Inform. Theory*, 55(7):3051–3073, 2009.

BDGL16. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, *27th SODA*, pages 10–24. ACM-SIAM, January 2016.

BV11. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, October 2011.

CDMT22. Kevin Carrier, Thomas Debris-Alazard, Charles Meyer-Hilfiger, and Jean-Pierre Tillich. Statistical decoding 2.0: Reducing decoding to LPN. Cryptology ePrint Archive, Paper 2022/1000, 2022. `https://eprint.iacr.org/2022/1000`.

Che13. Yuanmi Chen. *Réduction de réseau et sécurité concrète du chiffrement complètement homomorphe*. PhD thesis, Paris 7, 2013.

Chi14. Mao-Ching Chiu. Non-binary polar codes with channel symbol permutations. In *2014 International Symposium on Information Theory and its Applications*, pages 433–437, 2014.

CS88. John H. Conway and Neil J. A. Sloane. *Sphere Packings, Lattices and Groups*, volume 290 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1988.

DKR+20. Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`.

Duc18. Léo Ducas. Shortest vector from lattice sieving: A few dimensions for free. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 125–145. Springer, Heidelberg, April / May 2018.

EJK20. Thomas Espitau, Antoine Joux, and Natalia Kharchenko. On a dual/hybrid approach to small secret LWE - A dual/enumeration technique for learning with errors and application to security estimates of FHE schemes. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *INDOCRYPT 2020*, volume 12578 of *LNCS*, pages 440–462. Springer, Heidelberg, December 2020.

GJ21. Qian Guo and Thomas Johansson. Faster dual lattice attacks for solving LWE with applications to CRYSTALS. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021, Part IV*, volume 13093 of *Lecture Notes in Computer Science*, pages 33–62. Springer, 2021.

GJS15a. Qian Guo, Thomas Johansson, and Paul Stankovski. Coded-BKW: Solving LWE using lattice codes. In Gennaro and Robshaw [GR15], pages 23–42.

GJS15b. Qian Guo, Thomas Johansson, and Paul Stankovski. Coded-BKW: Solving LWE using lattice codes. In *Advances in Cryptology - CRYPTO 2015*, volume 9215 of *LNCS*, pages 23–42, Santa Barbara, CA, USA, August 2015. Springer.

GR15. Rosario Gennaro and Matthew J. B. Robshaw, editors. *CRYPTO 2015, Part I*, volume 9215 of *LNCS*. Springer, Heidelberg, August 2015.

KF15. Paul Kirchner and Pierre-Alain Fouque. An improved BKW algorithm for LWE with applications to cryptography and lattices. In Gennaro and Robshaw [GR15], pages 43–62.

Knu97. Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

KU10. Satish B. Korada and Rüdiger Urbanke. Polar codes are optimal for lossy source coding. *IEEE Trans. Inform. Theory*, 56(4):1751–1768, 2010.

LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

MAB+22. Matzov, Daniel Apon, Daniel J. Bernstein, Carl Mitchell, Léo Ducas, Martin Albrecht, and Chris Peikert. Improved Dual Lattice Attack. `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/Fm4cDfsx65s`, 2022.

MAT22. MATZOV. Report on the Security of LWE: Improved Dual Lattice Attack. Available at `https://doi.org/10.5281/zenodo.6412487`, April 2022.

MR09. Daniele Micciancio and Oded Regev. Lattice-based cryptography. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 147–191. Springer, Heidelberg, Berlin, Heidelberg, New York, 2009.

Pra62. Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery.

SAB+20. Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`.

Şaş11. Eren Şaşoğlu. Polarization and polar codes. *Foundations and Trends in Communications and Information Theory*, 8(4):259–381, 2011.

Sav21. Valentin Savin. Non-binary polar codes for spread-spectrum modulations. In *2021 11th International Symposium on Topics in Coding (ISTC)*, pages 1–5, 2021.

Sch03. Claus Peter Schnorr. Lattice reduction by random sampling and birthday methods. In Helmut Alt and Michel Habib, editors, *STACS 2003*, pages 145–156, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

SSTX09. Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 617–635. Springer, Heidelberg, December 2009.

STA09. Eren Sasoglu, Emre Telatar, and Erdal Arikan. Polarization for arbitrary discrete memoryless channels. pages 144 – 148, 11 2009.

ZF96. Ram Zamir and Meir Feder. On lattice quantization noise. *IEEE Trans. Inform. Theory*, 42(4):1152–1159, 1996.

Appendices

A Proof of Lemma 3.1 and Lemma 3.2

Lemma 3.1. *Let $(X_j)_{j\in[\![1,N]\!]}$ be N i.i.d random variables drawn according to a modular Gaussian over $\mathbb{Z}_q$ of mean 0 and variance σ^2. Then $\Re\left(\sum_{j=1}^N e^{\frac{2i\pi X_j}{q}}\right)$ is approximately normally distributed with*

$$\mathbb{E}\left(\Re\left(\sum_{j=1}^N e^{\frac{2i\pi X_j}{q}}\right)\right) \geq Ne^{-2\left(\frac{\pi\sigma}{q}\right)^2}$$

and if $\sigma \geq \sqrt{\frac{q\log(2)}{8\pi^2}}$, then we have

$$\mathbb{V}\left(\Re\left(\sum_{j=1}^N e^{\frac{2i\pi X_j}{q}}\right)\right) \leq \frac{N}{2}\left(1+2e^{-8\left(\frac{\pi\sigma}{q}\right)^2}\right)$$

Proof. We denote ρ the distribution function of the X_j's. Then we have

$$\begin{aligned}\mathbb{E}\left(\Re\left(\sum_{j=1}^N e^{\frac{2i\pi X_j}{q}}\right)\right) &= N\cdot\Re\left(\sum_{j\in\mathbb{Z}_q}\left(e^{\frac{2i\pi j}{q}}\rho(j)\right)\right)\\ &= N\cdot\Re\left(\widehat{\rho}(1)\right)\\ &\geq Ne^{-2\left(\frac{\pi\sigma}{q}\right)^2}\end{aligned}$$

The last inequality holds using [MAT22, Lemma 2.5].

On the other hand, using the same trick as in the proof of Lemma 3.2, we have

$$\mathbb{V}\left(\Re\left(\sum_{j=1}^N e^{\frac{2i\pi X_j}{q}}\right)\right) = \frac{N}{2}\cdot\left(1+\mathbb{E}\left(\Re\left(e^{\frac{4i\pi X_j}{q}}\right)\right)\right)$$

and so, we end the proof using again [MAT22, Lemma 2.5].

Lemma 3.2. *Let $(X_j)_{j\in[\![1,N]\!]}$ be N i.i.d random variables drawn according to a uniform distribution over $\mathbb{Z}_q$. Then $\Re\left(\sum_{j=1}^N e^{\frac{2i\pi X_j}{q}}\right)$ is approximately normally distributed with*

$$\mathbb{E}\left(\Re\left(\sum_{j=1}^N e^{\frac{2i\pi X_j}{q}}\right)\right) = 0$$

and

$$\mathbb{V}\left(\Re\left(\sum_{j=1}^N e^{\frac{2i\pi X_j}{q}}\right)\right) = \frac{N}{2}$$

Proof. First of all, we have

$$\begin{aligned}
\mathbb{E}\left(\Re\left(e^{\frac{2i\pi X_j}{q}}\right)\right) &= \Re\left(\mathbb{E}\left(e^{\frac{2i\pi X_j}{q}}\right)\right) \\
&= \Re\left(\sum_{j=-\lfloor q/2\rfloor}^{\lceil q/2\rceil-1} \frac{1}{q} e^{\frac{2i\pi j}{q}}\right) \\
&= \Re\left(\frac{e^{\frac{-2i\pi\lfloor q/2\rfloor}{q}}}{q} \cdot \sum_{j=0}^{q-1} e^{\frac{2i\pi j}{q}}\right) \\
&= \Re\left(\frac{e^{\frac{-2i\pi\lfloor q/2\rfloor}{q}}}{q} \cdot \left(\frac{1-\left(e^{\frac{2i\pi}{q}}\right)^q}{1-e^{\frac{2i\pi}{q}}}\right)\right) \\
&= 0
\end{aligned}$$

and with the same arguments:

$$\mathbb{E}\left(\Re\left(e^{\frac{4i\pi X_j}{q}}\right)\right) = 0$$

So, on the one hand

$$\mathbb{E}\left(\Re\left(\sum_{j=1}^{N} e^{\frac{2i\pi X_j}{q}}\right)\right) = \sum_{j=1}^{N} \mathbb{E}\left(\Re\left(e^{\frac{2i\pi X_j}{q}}\right)\right) = 0$$

and on the other hand

$$\begin{aligned}
\mathbb{V}\left(\Re\left(\sum_{j=1}^{N} e^{\frac{2i\pi X_j}{q}}\right)\right) &= \sum_{j=1}^{N} \mathbb{V}\left(\Re\left(e^{\frac{2i\pi X_j}{q}}\right)\right) \\
&= N \cdot \mathbb{E}\left(\Re\left(e^{\frac{2i\pi X_j}{q}}\right)^2\right) \\
&= \frac{N}{4} \cdot \mathbb{E}\left(\left(e^{\frac{2i\pi X_j}{q}}\right)^2 + 2\left|e^{\frac{2i\pi X_j}{q}}\right|^2 + \left(\overline{e^{\frac{2i\pi X_j}{q}}}\right)^2\right) \\
&= \frac{N}{2} \cdot \left(1 + \mathbb{E}\left(\Re\left(e^{\frac{4i\pi X_j}{q}}\right)\right)\right) \\
&= \frac{N}{2}
\end{aligned}$$

B Proof of Lemma 3.5

Lemma 3.5. $\mathbb{E}[\varepsilon_{j,\mathrm{code}}] \geqslant e^{-2\left(\frac{\pi\tau_{\mathrm{code}}}{p}\right)^2} \triangleq D_{\mathrm{code}}^{-\frac{1}{2}}$ *and* $\mathbb{E}\left[\varepsilon_{j,\mathrm{code}}^2\right] \leqslant 2e^{-8\left(\frac{\pi\tau_{\mathrm{code}}}{p}\right)^2}$ *where* $\tau_{\mathrm{code}} = \|\mathbf{s}_{\mathrm{code}}\| \frac{p^{1-\frac{n_{\mathrm{fft}}}{n_{\mathrm{code}}}}}{\sqrt{2\pi e}}$.

Proof (Proof of Lemma 3.5). By Assumption 3.3, the decoding distance d is given by the equation $\mathrm{Vol}_{n_{\mathrm{code}}}(B_d) = p^{n_{\mathrm{code}}-n_{\mathrm{fft}}}$, where B_d is the volume of a ball of radius d in $\mathbb{R}^{n_{\mathrm{code}}}$. Since $\mathrm{Vol}_n(B_d) = \frac{\pi^{n/2}d^n}{\Gamma\left(\frac{n}{2}+1\right)}$ we obtain

$$\begin{aligned} d &= \frac{p^{1-\frac{n_{\mathrm{fft}}}{n_{\mathrm{code}}}}\left(\Gamma\left(\frac{n_{\mathrm{code}}}{2}+1\right)\right)^{\frac{1}{n_{\mathrm{code}}}}}{\sqrt{\pi}} \\ &\sim p^{1-\frac{n_{\mathrm{fft}}}{n_{\mathrm{code}}}}\sqrt{\frac{n_{\mathrm{code}}}{2\pi e}}. \end{aligned}$$

We apply this to the vector $\mathbf{z}_j \triangleq \left\lceil \frac{p}{q}\mathbf{y}_{j,\mathrm{code}} \right\rfloor$. By Assumption 3.3, the distribution of $\mathbf{y}_{j,\mathrm{code}}$ is approximately uniform modulo q so $\mathbf{z}_j$ is approximately uniform modulo p. By Assumption 3.3, the distribution of $\mathbf{t}_{j,\mathrm{code}} = \mathbf{z}_j - \mathbf{G}\mathbf{u}_{\mathrm{code}}$ is approximately uniform in an (integer) ball of radius d as defined above. By Assumption 3.3, the components of $\mathbf{z}_j$ are therefore approximately independent and identically distributed according to a modular Gaussian of standard deviation

$$\sigma_{\mathrm{code}} \triangleq \frac{d}{\sqrt{n_{\mathrm{code}}}} = \frac{p^{1-\frac{n_{\mathrm{fft}}}{n_{\mathrm{code}}}}}{\sqrt{2\pi e}}.$$

Now the exponent of $\varepsilon_{j,\mathrm{code}}$ is a weighted sum of approximately independent modular Gaussian, hence by Assumption 3.3, it is also approximately distributed according to a modular Gaussian of standard deviation τ_{code}. Furthermore, we have that

$$\begin{aligned} \tau_{\mathrm{code}}^2 &= \mathrm{Var}\left(\sum_{i=1}^{n_{\mathrm{code}}} (\mathbf{t}_{j,\mathrm{code}})_i (\mathbf{s}_{\mathrm{code}})_i\right) \\ &= \sum_{i=1}^{n_{\mathrm{code}}} (\mathbf{s}_{\mathrm{code}})_i^2 \,\mathrm{Var}\left((\mathbf{t}_{j,\mathrm{code}})_i\right) \\ &= \sum_{i=1}^{n_{\mathrm{code}}} (\mathbf{s}_{\mathrm{code}})_i^2 \sigma_{\mathrm{code}}^2 \\ &= \|\mathbf{s}_{\mathrm{code}}\|^2 \sigma_{\mathrm{code}}^2. \end{aligned}$$

It follows by [MAT22, Theorem 2.4 and Lemma 2.5] that

$$\mathbb{E}[\varepsilon_{j,\mathrm{code}}] \geqslant e^{-2\left(\frac{\pi\tau_{\mathrm{code}}}{p}\right)^2}, \qquad \mathbb{E}\left[\varepsilon_{j,\mathrm{code}}^2\right] \leqslant 2e^{-8\left(\frac{\pi\tau_{\mathrm{code}}}{p}\right)^2}.$$

□

C Proof of the success probability of Algorithm 2

Theorem 3.8. *Let $(n, m, q, \chi_s, \chi_e)$ be* LWE *parameters, (β_0, β_1, n_{en}, n_{fft}, n_{lat}, n_{code},* Bet*, p, C, N, R, μ) be a partial tuple of parameters and $0 < \nu < 1$ be the desired failure probability. Then Algorithm 2 with parameters*

$$N \geqslant \tilde{N}_{\mathrm{eq}} \cdot \tilde{N}_{\mathrm{round}} \cdot \tilde{N}_{\mathrm{code}} \cdot \tilde{N}_{\mathrm{arg}} \cdot \tilde{N}_{\mathrm{fpfn}},$$

$$
\begin{aligned}
C &= \tilde{\phi}_{\mathrm{fp}} \sqrt{D \cdot \tilde{N}_{\mathrm{arg}}}, \\
R &\geqslant \frac{\ln(\frac{\nu}{2})}{\ln(1-\tilde{p}_\tau) + \ln(1-\mu)}, \\
\mu &\leqslant \frac{\nu \tilde{p}_\tau}{\nu \tilde{p}_\tau + 2 - \nu}
\end{aligned}
$$

returns $\mathbf{s}_{\mathrm{en}}$ *with probability at least* $\frac{1-\nu}{4}$*. Above, we have*

$$
\tilde{N}_{\mathrm{round}} \stackrel{\triangle}{=} \prod_{\bar{s} \neq 0} \left(\frac{\sin\left(\frac{\pi \bar{s}}{p}\right)}{\frac{\pi \bar{s}}{p}} \right)^{-2 n_{\mathrm{fft}} \chi_s(\bar{s})}, \qquad \tilde{N}_{\mathrm{eq}} \stackrel{\triangle}{=} e^{4\left(\frac{\pi \sigma_s \ell}{q}\right)^2}, \qquad \tilde{N}_{\mathrm{arg}} \approx \frac{1}{2},
$$

$$
\tilde{N}_{\mathrm{code}} \stackrel{\triangle}{=} \exp\left(2 \frac{\pi \sigma_s^2}{e} n_{\mathrm{code}} \cdot p^{-\frac{n_{\mathrm{fft}}}{n_{\mathrm{code}}}} \right),
$$

$$
\tilde{N}_{\mathrm{fpfn}} = \left(\tilde{\phi}_{\mathrm{fp}} + \tilde{\phi}_{\mathrm{fn}} \right)^2, \quad \tilde{\phi}_{\mathrm{fn}} = \Phi^{-1}\left(1 - \tfrac{\mu}{2}\right), \quad \tilde{\phi}_{\mathrm{fp}} = \Phi^{-1}\left(1 - \tfrac{\mu}{2 \cdot |\,\mathrm{Bet}\,| \cdot p^{n_{\mathrm{fft}}}}\right),
$$

$$
\tilde{p}_\tau \text{ such that } \Pr_{\mathbf{s}}[p_\tau(\mathbf{s}) \geqslant \tilde{p}_\tau] \geqslant 3/4, \qquad p_\tau(\mathbf{s}) = \frac{|\mathcal{T}(\mathbf{s})|}{(n_{\mathrm{en}} + n_{\mathrm{code}})!}
$$

$$
\mathcal{T}(\mathbf{s}) = \{ \text{ permutation } \tau \text{ of } [\![1, n_{\mathrm{en}} + n_{\mathrm{code}}]\!] : \mathbf{s}_{\mathrm{en}}^\tau \in \mathrm{Bet} \}
$$

Ω_s is the universe of χ_s and ℓ is the average length of the vectors given by the short vectors sampling algorithm.

Proof. By Lemma 3.7, if $|\mathcal{T}(\mathbf{s})| > 0$ the algorithm succeeds with probability at least $1 - \nu$ when

$$
\begin{aligned}
N &\geqslant N_{\mathrm{eq}} \cdot \Re\left(\psi(\mathbf{s}_{\mathrm{code}})\right)^{-2} N_{\mathrm{round}} \cdot N_{\mathrm{code}} \cdot N_{\mathrm{arg}} \cdot N_{\mathrm{fpfn}} \\
C &= \phi_{\mathrm{fp}} \sqrt{N \cdot N_{\mathrm{arg}}} \\
R &\geqslant \frac{\ln(\frac{\nu}{2})}{\ln(1-p_\tau) + \ln(1-\mu)} \stackrel{\triangle}{=} R_{\max}(p_\tau), \\
\mu &\leqslant \frac{\nu p_\tau}{\nu p_\tau + 2 - \nu} \stackrel{\triangle}{=} \mu_{\max}(p_\tau), \\
p_\tau &= \frac{|\mathcal{T}(\mathbf{s})|}{(n_{\mathrm{en}} + n_{\mathrm{code}})!}
\end{aligned}
$$

and the other parameters are specified in Theorem 3.4.

We assume that $\log(N_{\mathrm{eq}} \cdot \Re\left(\psi(\mathbf{s}_{\mathrm{code}})\right)^{-2} N_{\mathrm{round}} \cdot N_{\mathrm{code}} \cdot N_{\mathrm{arg}} \cdot N_{\mathrm{fpfn}})$ is approximately normally distributed (over the randomness of $\mathbf{s}$ and $\mathbf{e}$) and so it is greater than its expectation with probability $\approx \frac{1}{2}$. It therefore remains to compute each term of this expectation.

Observe that $\tilde{N}_{\mathrm{round}}$, $\tilde{N}_{\mathrm{arg}}$ and $\tilde{N}_{\mathrm{eq}}$ are defined exactly as $\tilde{D}_{\mathrm{round}}$, $\tilde{D}_{\mathrm{arg}}$ and $\tilde{D}_{\mathrm{eq}}$ in [MAT22, Theorem 5.9] so we do not reproduce the proof.

Recall that $N_{\text{code}}^{-\frac{1}{2}} = e^{-2\left(\frac{\pi\tau_{\text{code}}}{p}\right)^2}$ where $\tau_{\text{code}} = \|\mathbf{s}_{\text{code}}\| \frac{p^{1-\frac{n_{\text{fft}}}{n_{\text{code}}}}}{\sqrt{2\pi e}}$. Since each coordinate of $\mathbf{s}_{\text{code}}$ is independently distributed, $\log(N_{\text{code}})$ is approximately normal with expectation

$$\begin{aligned}\mathbb{E}[\log N_{\text{code}}] &= \frac{4\pi^2}{p^2}\mathbb{E}\left[\|\mathbf{s}_{\text{code}}\|^2\right]\frac{p^{2-2\frac{n_{\text{fft}}}{n_{\text{code}}}}}{2\pi e} \\ &= \frac{2\pi}{e}\sigma_s^2 n_{\text{code}} \cdot p^{-\frac{n_{\text{fft}}}{n_{\text{code}}}} \\ &= \log \tilde{N}_{\text{code}}.\end{aligned}$$

Recall that $\psi(\mathbf{s}_{\text{code}}) = e^{\frac{2i\pi c_{q'}}{p}\sum_{j=1}^{n_{\text{code}}}(\mathbf{s}_{\text{code}})_j}$ where $c_{q'} = 0$ if q' is odd and $c_{q'} = \frac{1}{2q'}$ if q' is even, where $q' \triangleq \frac{q}{\gcd(p,q)}$. Assuming that χ_s is a centered distribution, we have

$$\mathbb{E}\left[\sum_{j=1}^{n_{\text{code}}}(\mathbf{s}_{\text{code}})_j\right] = 0.$$

Furthermore, since the coordinates of $\mathbf{s}_{\text{code}}$ are independent,

$$\text{Var}\left(\sum_{j=1}^{n_{\text{code}}}(\mathbf{s}_{\text{code}})_j\right) = \sum_{j=1}^{n_{\text{code}}}\text{Var}\left((\mathbf{s}_{\text{code}})_j\right) = n_{\text{code}}\sigma_s^2.$$

For n_{code} not too small, $\sum_{j=1}^{n_{\text{code}}}(\mathbf{s}_{\text{code}})_j$ is a sum of independent variables so is approximately normal with mean and deviation computed above. As a result, for any threshold a,

$$\Pr\left[\left|\sum_{j=1}^{n_{\text{code}}}(\mathbf{s}_{\text{code}})_j\right| > a\right] = \Pr\left[|\mathcal{N}(0, n_{\text{code}}\sigma_s^2)| > a\right] \leqslant 1 - \Phi\left(\frac{a}{n_{\text{code}}\sigma_s^2}\right)$$

Pick $a = q/100$: for all reasonable choices of distributions and n_{code}, we will have that $\frac{q}{100 n_{\text{code}}\sigma_s^2}$ is big enough to make the above probability lower than, say, 1%. When this is the case, we have that

$$\Re(\psi(\mathbf{s}_{\text{code}})) \geqslant \cos(2\pi\frac{qc_{q'}}{100p}) = \cos(2\pi\frac{q\gcd(p,q)}{200pq}) \geqslant \cos(\frac{2\pi}{200}) \geqslant 0.999.$$

As a result, for at least 99% of the secret, we have that $\Re(\psi(\mathbf{s}_{\text{code}}))^{-2} \leqslant \tilde{\psi} \approx 1.001$. In other words, we have that

$$\mathbb{E}\left[\log\left(\Re\left(\psi(\mathbf{s}_{\text{code}})\right)^{-2}\right)\right] \approx 0.$$

For $\tilde{N}_{\text{fpfn}}$, we do repeat the analysis of [MAT22, Theorem 5.9] where we replace $N_{\text{en}}(\mathbf{s}_{\text{en}})$ by $|\,\text{Bet}\,|$. We conclude in the same way as [MAT22].

We now focus on the value of R and ν. By our assumption, for a fraction $3/4$ of the secret, we have that

$$p_\tau(\mathbf{s}) \geqslant \tilde{p}_\tau.$$

Now observe that the map $\mu_{\max}$ is increasing, therefore if $p_\tau \geqslant \tilde{p}_\tau$ then $\mu_{\max}(p_\tau) \geqslant \mu_{\max}(\tilde{p}_\tau)$. As a result, the inequality $\mu \leqslant \mu_{\max}(p_\tau)$ is satisfied for all those instances as soon as $\mu_{\max}(\tilde{p}_\tau)$. Similarly, the map $R_{\max}$ is decreasing so if $R \geqslant R_{\max}(\tilde{p}_\tau)$ then $R \geqslant R_{\max}(p_\tau)$ as well.

In conclusion, overall, the algorithm succeeds when the inequality on N, R and μ are satisfied. The inequality on N is satisfied for a fraction $1/2$ of the secrets. The inequalities on R and μ are satisfied for a fraction $3/4$ of the secrets. Hence, the fraction of secret for which all inequalities hold is at least $1/4$. This means the success probability overall is at least $\frac{1-\nu}{4}$. □

D Distortion properties of polar codes

In this appendix we give more details about the construction of polar codes over $\mathbb{Z}_q$ that is mentioned in Subsection 3.3. We then verify that, when decoding random words, it is possible, to achieve a typical decoding distance which is very close to the lattice analogue of the Gilbert-Varshamov distance that we recall to be

$$\omega \approx \sqrt{\frac{n}{2\pi e}} q^{1-\frac{k}{n}}. \tag{58}$$

where n and k are respectively the length and the dimension of the polar code.

Construction of polar codes over $\mathbb{Z}_q$. Let assume a codeword in $\mathbb{Z}_q^n$ of which each symbol is transmitted through a Gaussian channel of standard deviation $\sigma \stackrel{\triangle}{=} \frac{\omega}{\sqrt{n}}$. The polar code construction basically consists of transforming those n Gaussian channels into n virtual channels that are, for most of them, either of maximal or minimal entropy. The idea then is to fix (or in other words *freeze*) the information that will transit via the bad channels and involve the good channels for the k symbols of useful information.

Our construction essentially follows the papers [STA09, Chi14, Sav21]. We refer to those articles for more details about polar codes. It is a recursive construction which can be described as follows.

Definition D.1 ($(U+V, \alpha U)$-construction). *Let U and V be two linear codes of the same length n over $\mathbb{Z}_q$ and let $\alpha \in \mathbb{Z}_q^*$ be an invertible scalar. The $(U+V, \alpha V)$ is a $\mathbb{Z}_q$-linear code of length $2n$ defined by*

$$(U+V, \alpha U) \stackrel{\triangle}{=} \{(\mathbf{u}+\mathbf{v}, \alpha\mathbf{u}) : \mathbf{u} \in U \text{ and } \mathbf{v} \in V\} \tag{59}$$

A polar code of length $n \stackrel{\triangle}{=} 2^m$ and dimension k is then defined by

Definition D.2 (polar code). *Let F be a subset of $\{0,1\}^m$ of size $2^m - k$ and let α be a function mapping the binary words of length $< m$ to $\mathbb{Z}_q^*$. The polar code of length 2^m associated to F and α is defined recursively by (we denote by ε the empty binary word)*

$$C \triangleq U_\varepsilon$$

where the $U_{\mathbf{x}}$ are codes of length 1 for all $\mathbf{x} \in \{0,1\}^m$ and are given by

$$U_{\mathbf{x}} = \begin{cases} \{0\} & \text{if } \mathbf{x} \in F \\ \mathbb{Z}_q & \text{otherwise} \end{cases} \tag{60}$$

and the other $U_{\mathbf{x}}$'s where $\mathbf{x}$ is a binary word of length $< m$ are defined recursively by

$$U_{\mathbf{x}} \triangleq \left(U_{0||\mathbf{x}} + U_{1||\mathbf{x}}, \alpha(\mathbf{x}) U_{0||\mathbf{x}}\right).$$

Thus, a polar code is fully defined by the set F of *frozen* positions and the $\alpha(\mathbf{x})$'s. In [Chi14], it is admitted that choosing the $\alpha(\mathbf{x})$'s uniformly at random in $\mathbb{Z}_q^*$ is good enough. However, in [Sav21], it is shown that those coefficients can be optimized. Thereafter, we do not use the optimization technique from [Sav21] but simply try several polar codes then choose the best of them. On another hand, we classically determine the optimal frozen positions F using Monte-Carlo simulation: we run many times a genie-aided decoder for estimating the probability distribution of each virtual channel then selecting the worst of them; that are the $2^m - k$ virtual channels for which the error probabilities are the highest.

Decoding polar codes over $\mathbb{Z}_q$. The Successive Cancelation (SC) decoding algorithm (see [STA09, Chi14, Sav21]) can be described as a recursive decoding algorithm. For each code $U_{\mathbf{x}} \subseteq \mathbb{Z}_q^{2^{m-t}}$ such that $\mathbf{x} \in \{0,1\}^t$ and $t \in [\![0, m-1]\!]$, we decode a noisy codeword in this code by using recursively the decoders of $U_{0||\mathbf{x}}$ and $U_{1||\mathbf{x}}$.

Let $\mathbf{c} \triangleq (c_1, \cdots, c_{2^{m-t}}) \triangleq (\mathbf{u} + \mathbf{v}, \alpha(\mathbf{x})\mathbf{u})$ be a codeword in $U_{\mathbf{x}}$; *i.e.* $\mathbf{u} \triangleq (u_1, \cdots, u_{2^{m-t-1}})$ and $\mathbf{v} \triangleq (v_1, \cdots, v_{2^{m-t-1}})$ are respectively in $U_{0||\mathbf{x}}$ and $U_{1||\mathbf{x}}$. Let assume that $\mathbf{c}$ is transmitted through a channel $W^{(\mathbf{x})}$; let

$$\mathbf{y} \triangleq (y_1, \cdots, y_{2^{m-t}}) \triangleq (\mathbf{y}_\ell, \mathbf{y}_r) \in \mathbb{Z}_q^{2^{m-t-1}} \times \mathbb{Z}_q^{2^{m-t-1}}$$

be the received word. We assume that for each position $i \in [\![1, 2^{m-t}]\!]$ and symbol $s \in \mathbb{Z}_q$, we know the probability that the transmitted symbol is s knowing that the received one is y_i:

$$\Pi_i^{(\mathbf{x})}(s) \triangleq \Pr[c_i = s | y_i] \tag{61}$$

Instead of decoding directly $\mathbf{y}$, we decode first $\mathbf{y}_\ell - \alpha(\mathbf{x})^{-1}\mathbf{y}_r$ expecting to find $\mathbf{v} \in U_{1||\mathbf{x}}$. The virtual channel through which $\mathbf{v}$ has transited is then the serialization of two $W^{(\mathbf{x})}$ channels that we denote by $W^{(1||\mathbf{x})}$. Thus for each coordinate $i \in [\![1, 2^{m-t-1}]\!]$ and symbol $s \in \mathbb{Z}_q$, we have the probability

$$\Pi_i^{(1||\mathbf{x})}(s) \triangleq \Pr[v_i = s | y_i, y_{i+2^{m-t-1}}] \tag{62}$$

$$= \sum_{s' \in \mathbb{Z}_q} \Pi_i^{(\mathbf{x})}(s + s') \cdot \Pi_{i+2^{m-t-1}}^{(\mathbf{x})}(\alpha(\mathbf{x}) \cdot s') \tag{63}$$

$$= \sum_{s' \in \mathbb{Z}_q} \Pi_i^{(\mathbf{x})}(s - s') \cdot \Pi_{i+2^{m-t-1}}^{(\mathbf{x})}(-\alpha(\mathbf{x}) \cdot s') \tag{64}$$

$$= \left(\Pi_i^{(\mathbf{x})} * \overline{\Pi}_{i+2^{m-t-1}}^{(\mathbf{x})} \right)(s) \tag{65}$$

where $\overline{\Pi}_i^{(\mathbf{x})}(s) \triangleq \Pi_i^{(\mathbf{x})}(-\alpha(\mathbf{x}) \cdot s)$.

On another hand, let us assume that the decoding of $\mathbf{y}_\ell - \alpha(\mathbf{x})^{-1}\mathbf{y}_r$ has led us to the vector $\widetilde{\mathbf{v}}$ that we expect to be $\mathbf{v}$ (for the genie-aided decoder used for the construction of the code, we actually take $\widetilde{\mathbf{v}} = \mathbf{v}$, regardless of the result of the decoding of $\mathbf{y}_\ell - \alpha(\mathbf{x})^{-1}\mathbf{y}_r$). We now have two independent noisy versions of the same vector $\mathbf{u}$ that are $\alpha(\mathbf{x})^{-1}\mathbf{y}_r$ and $\mathbf{y}_\ell - \widetilde{\mathbf{v}}$. In other words, supposing $\widetilde{\mathbf{v}} = \mathbf{v}$, the vector $\mathbf{u}$ has been sent twice through the channel $W^{(\mathbf{x})}$; we denote by $W^{(0||\mathbf{x})}$ the resulting channel and for each coordinate $i \in [\![1, 2^{m-t-1}]\!]$ and symbol $s \in \mathbb{Z}_q$, we have the probability

$$\Pi_i^{(0||\mathbf{x})}(s) \triangleq \Pr[u_i = s | y_i, y_{i+2^{m-t-1}}] \tag{66}$$

$$= \frac{1}{\eta} \cdot \Pi_i^{(\mathbf{x})}(s + \widetilde{v}_i) \cdot \Pi_{i+2^{m-t-1}}^{(\mathbf{x})}(\alpha(\mathbf{x}) \cdot s) \tag{67}$$

where $\eta \triangleq \sum_{s' \in \mathbb{Z}_q} \Pi_i^{(\mathbf{x})}(s' + \widetilde{v}_i) \cdot \Pi_{i+2^{m-t-1}}^{(\mathbf{x})}(\alpha(\mathbf{x}) \cdot s')$ is a normalization factor.

Finally, for decoding a received word $\mathbf{y} \in \mathbb{Z}_q^{2^m}$ in the code U_ε that has been sent through a Gaussian channel of standard deviation σ, one essentially has to compute recursively the vector probabilities $\Pi_i^{(\mathbf{x})}$ for all $t \in [\![1, m]\!]$, $\mathbf{x} \in \{0,1\}^t$ and $i \in [\![1, 2^{m-t}]\!]$ using the Equations (65) and (67). Note that the initial channel $W^{(\varepsilon)}$ is the original Gaussian channel; so for all $i \in [\![1, 2^m]\!]$ and $s \in \mathbb{Z}_q$, we have

$$\Pi_i^{(\varepsilon)}(s) = D_{\mathbb{Z}_q,\sigma}(y_i - s) \tag{68}$$

where $D_{\mathbb{Z}_q,\sigma}$ is given in Subsection 2.4.

When arriving to the codes on the leaves – that are the codes $U_{\mathbf{x}}$ such that $\mathbf{x} \in \{0,1\}^m$ – then we can exhaustively decode $U_{\mathbf{x}}$:

1. if $\mathbf{x} \in F$ (meaning the corresponding symbol is *frozen*) then the only possible codeword in $U_{\mathbf{x}}$ is the symbol 0,
2. if $\mathbf{x} \notin F$, then we choose the maximum likelihood codeword in $U_{\mathbf{x}}$ that is the symbol s for which $\Pi_1^{(\mathbf{x})}(s)$ is the greatest.

The running time of Successive Cancelation decoding presented above is given by the following lemma:

Lemma D.3 (Complexity of the SC decoder). *Assuming q is a power of 2. The running time for decoding a word in a polar code of length 2^m and dimension k over $\mathbb{Z}_q$ is:*

$$T_{\text{polar}} \leq 3 \cdot C_{\text{mul}} \cdot q \cdot \log_2(q) \cdot m \cdot 2^m \tag{69}$$

where C_{mul} is the cost of one multiplication.

Proof. For all $t \in [\![1, m]\!]$, $\mathbf{x} \in \{0,1\}^t$ and $i \in [\![1, 2^{m-t}]\!]$ – *i.e.* for $m \cdot 2^m$ triplets $(t, \mathbf{x}, i)$ – we can compute the vector of probabilities $\Pi_i^{(\mathbf{x})}$ with at most $3 \cdot q \cdot \log_2(q)$ multiplications. Indeed, we either have to compute Equation (65) or Equation (67). In the first case, it is a convolution; this can be done with the help of three fast Fourier transforms, each costing $q \cdot \log_2(q)$ multiplications (see [Knu97, pp. 306-311]). In the second case, we only have to do $2.q$ multiplications and q additions, which is less than the cost of a convolution.

Remark D.4. We could reduce the cost of the SC decoder by considering the vectors of LLR (*Log Likelihood Ratio*) instead of the vectors of probabilities. This trick allows to transform multiplications into additions.

Punctured polar code. The polar codes construction above is about codes of length that are a power of 2. In our case, we may require codes of other length. A simple way for reducing the length of a code without changing its dimension is to puncture it. Let n, k be two positive integers. We build a linear code of length n and dimension k by puncturing a polar code of length 2^m and dimension k where $m \triangleq \lceil \log_2(n) \rceil$. Let denote by $\ell \triangleq 2^m - n$ the number of symbol to puncture. The puncturing operation essentially consists of ignoring the ℓ first symbols of the codeword; that is equivalent to suppose that the ℓ first physical channels through which transit the codewords are of maximal entropy:

$$\Pi_i^{(\varepsilon)}(s) \triangleq \frac{1}{q} \qquad \forall i \in [\![1, \ell]\!] \tag{70}$$

Note that we made this assumption both for the decoder and also for the genie-aided decoder used to determine the frozen positions.

List decoding using probabilistic SC decoder. We can modify the above SC decoder to obtain a probabilistic decoder. To this end, when decoding non-frozen symbols in the codes on the leaves $U_{\mathbf{x}}$ where $\mathbf{x} \in \{0,1\}^m \setminus F$, then output the symbol s according to the distribution $\Pi_1^{(\mathbf{x})}$ instead of returning the one with the best probability.

To turn this probabilistic SC decoder into a list decoder, one only has to running it L times then choosing the codeword that minimizes the decoding distance. The complexity of a such algorithm is essentially L times the complexity of the SC decoder given by Lemma D.3. Note that, contrary to some more classical list decoders of polar codes, the L decoding procedures of our algorithm can be trivially parallelized.

The distortion properties of our polar code decoders. The *distortion* of a code is the typical distance that it is possible to achieve when decoding random words. We verified in experiments that the distortion of the polar codes over $\mathbb{Z}_q$, equipped with our list decoder, is close enough to the ideal distortion; namely the Gilbert-Varshamov analogue distance recalled by Equation (58).

To this end, we provide a C implementation of our polar codes and probabilistic SC decoder that can be found in this repository. We run some experiments

for some codes that are those we need in our dual attacks (see Table 4). The obtained results, summarized in Table D, are really satisfying given the relatively small parameters we had to consider. To fill Table D, let us recall that:

- we designed our polar codes over $\mathbb{Z}_q$ by choosing some coefficients $\alpha(\mathbf{x})$'s that are invertible in $\mathbb{Z}_q$;
- we run our experiments on several polar codes (changing each time the $\alpha(\mathbf{x})$'s) and then we choose the code that provides the best distortion;
- we punctured the polar codes for achieving lengths that are not powers of two;
- we measured the distortion on average on 100 decodings;
- we run both unique deterministic SC decoder ($L = 1$) and list decoder using probabilistic SC decoder (with list size $L = 8$ and 16).

Table 5. Distortion of the polar codes involved in our dual lattice attacks

	Scheme	q	n	k	d_{GV}	distortion with $L = 1$	distortion with $L = 8$	distortion with $L = 16$
C0:	Kyber 512	64	42	14	25.1	27.2	26.8	26.7
	Kyber 768	128	76	19	80.3	86.6	85.2	84.2
	Kyber 1024	512	106	22	349.4	367.9	366.4	364.4
	LightSaber	128	34	12	32.6	36.7	35.4	34.9
	Saber	64	58	23	22.7	24.6	24.3	24.1
	FireSaber	64	101	33	40.0	42.5	42.4	42.1
	Scheme	q	n	k	d_{GV}	distortion with $L = 1$	distortion with $L = 8$	distortion with $L = 16$
CC:	Kyber 512	128	45	14	45.9	49.6	48.8	48.7
	Kyber 768	256	83	19	158.6	168.5	167.6	167.2
	Kyber 1024	128	124	31	102.5	107.7	107.3	106.8
	LightSaber	256	39	12	70.2	77.1	75.9	75.8
	Saber	128	67	22	51.5	54.8	54.3	54.1
	FireSaber	512	109	24	327.5	343.1	341.4	341.3
	Scheme	q	n	k	d_{GV}	distortion with $L = 1$	distortion with $L = 8$	distortion with $L = 16$
CN:	Kyber 512	1024	43	9	380.8	406.7	404.5	405.5
	Kyber 768	1024	89	15	726.8	772.1	767.9	761.9
	Kyber 1024	2048	126	19	1761.8	1840.9	1828.4	1830.1
	LightSaber	1024	35	9	246.6	273.1	272.5	271.0
	Saber	512	62	16	195.0	206.0	205.9	205.4
	FireSaber	64	102	36	36.0	38.2	38.0	37.9

E Source code

Our code relies on the modified LWE Estimator from [APS15] available at `https://github.com/malb/lattice-estimator/`. We also attached our code as an attachment to this PDF. Not all PDF viewers support this feature. If the reader's PDF reader does not then *e.g.* `pdfdetach` can be used to extract the source code without having to copy and paste it by hand. To run our code, run `git clone https://github.com/malb/lattice-estimator/` in the directory where `estimates.py` is located.

```
# -*- coding: utf-8 -*-
"""
Run like this::

    sage: attach("estimates_code.py")
    sage: %time results = runall() # our nest attack
    sage: %time results = runall(attack="matzov") # use original Matzov with modulo switching
    sage: %time results = runall(attack="matzov", use_optimizer=True) # code with custom optimizer
    sage: %time results = runall(attack="matzov_code_prange", bet="senum_all_zero") # our new attack
    sage: print(results_table(results)) # for raw table
    sage: print(results_table(results, "latex")) # for latex code
    sage: print(parameter_tables(results)) # for raw table
    sage: for (nn,tbl) in parameter_tables(results, "latex").items(): print("{}:\n{}".format(nn,tbl)) # for latex

"""
from sage.all import sqrt, log, exp, e, pi, RR, ZZ

from estimator.estimator.cost import Cost
from estimator.estimator.lwe_parameters import LWEParameters
from estimator.estimator.lwe import Estimate
from estimator.estimator.reduction import delta as deltaf
from estimator.estimator.reduction import RC, ReductionCost
from estimator.estimator.conf import red_cost_model as red_cost_model_default
from estimator.estimator.util import local_minimum, early_abort_range
from estimator.estimator.io import Logging
from estimator.estimator.schemes import (
    Kyber512,
    Kyber768,
    Kyber1024,
    LightSaber,
    Saber,
    FireSaber,
)
from estimator.estimator.schemes import TFHE630, TFHE1024
from estimator.estimator.nd import NoiseDistribution
from estimator.estimator.lwe_primal import primal_bdd

def minimizer_convex(f, bounds, x0=None, verbose=False, names=None):
    """
    This function tries to minimize f over the integers, restricted to a box.
    More precisely, it tries to find a local minimum of the function.
    It can optinally take a starting point as a hint, which can useful to avoid getting
    stuck in bad local minimima, or to speed up the search.

    :param f: the function to minimize
    :param bounds: an array of tuples (min,max) for the parameters, see blow for details
    :param x0: optional starting point of the search (tuple or array of value)
    :param verbose: set to True for debug output
    :param names: pretty print name in debug output, should be an array of string, one per dimension

    The min/max bounds can either be integers, or any callable objects. When it is a callable,
    it will be called with the values of the dimension that appear *before* in the list.

    Example:
    f=lambda x,y: -x*exp(-x)-2*y*exp(-y)
```

```
# or
f=lambda x,y: -x*(10-x)*y*(20-y)
(x_min, y_min), f_min = minimizer_convex(f, [(0, 10), (0, 10)])
# or
(x_min, y_min), f_min = minimizer_convex(f, [(0, 10), (0, 10)], x0=(5, 5))
"""

if verbose and names is None:
    names = ["x[{}]".format(i) for i in range(len(bounds))]

# we use a recursive algorithm
# :param x: current best coordinate
# :param i: coordinate to optimize
n = len(bounds)
def opt(x, i):
    # if verbose: print("{}opt x={}".format(" "*i, x))
    # base case: we have reached the last coordinate
    if i == n:
        return tuple(x), f(*x)
    # compute lower/upper bounds
    lb = bounds[i][0]
    if callable(lb):
        lb = lb(*x[0:i])
    ub = bounds[i][1]
    if callable(ub):
        ub = ub(*x[0:i])
    # make sure x[i] is within bounds (could be outside if the ub/lb changes with x)
    xx = list(x)
    xx[i] = max(lb, min(ub, xx[i]))
    x = xx
    # optimize recursively
    # if verbose: print("{}{}={} [{},{}]".format(" "*i, names[i], x[i], lb, ub))
    orig_x, best = opt(x, i+1)
    # if verbose: print("{}got x={}, f={}".format(" "*i, orig_x, best))
    # see if we can do better by changing x[i]
    best_x = orig_x
    # try to go up
    x_copy = list(orig_x)
    while x_copy[i] < ub:
        x_copy[i] += 1
        # if verbose: print("{}{}={}".format(" "*i, names[i], x_copy[i]))
        new_x, newf = opt(x_copy, i+1)
        x_copy = list(new_x)
        # if verbose: print("{}got x={}, f={}".format(" "*i, new_x, newf))
        if newf > best:
            if verbose: print("{}which is worse, stop going up".format(" "*i))
            break
        best_x = tuple(new_x)
        best = newf
    # try to go down (note that we restart from original position to avoid duplicating computations)
    # if we made progress in the loop, assume that we can't make progress by going
    # in the other direction
    if best_x[i] == orig_x[i]:
        x_copy = list(best_x)
        while x_copy[i] > lb:
            x_copy[i] -= 1
            # if verbose: print("{}{}={}".format(" "*i, names[i], x_copy[i]))
            new_x, newf = opt(x_copy, i+1)
            x_copy = list(new_x)
            # if verbose: print("{}got x={}, f={}".format(" "*i, new_x, newf))
            if newf > best:
                # if verbose: print("{}which is worse, stop going down".format(" "*i))
                break
            best_x = tuple(new_x)
            best = newf
    else:
        # if verbose: print("{}skip going down".format(" "*i))
        pass
```

```
            return best_x, best

        for x in bounds:
            assert type(x) is tuple, "bounds must contains tuples"
            assert len(x) == 2, "bounds must contains tuples of size 2"
        if x0 is None:
            # use lower bound as starting point
            x0 = [x[0] for x in bounds]
            if verbose:
                print(f"no starting point specified, starting at {x0}")
        else:
            # check consistenct
            assert len(bounds) == len(x0), "x0 must have the same dimension as bounds"
        return opt(tuple(x0), 0)

    def example1(verbose=False):
        # minimize over [0,10]x[0,20]
        f=lambda x,y: numerical_approx(-x*exp(-x/10)-2*y*exp(-y/10))
        (x, y), fmin = minimizer_convex(f, [(0, 10), (0, 20)], verbose=verbose)
        print("best: f({},{})={}".format(x, y, fmin))
        g=lambda p: f(p[0], p[1])
        (x, y) = minimize_constrained(g, [(0, 10), (0, 20)], [1, 1])
        print("best over reals: f({},{})={}".format(x, y, f(x, y)))

    def example2(verbose=False):
        # minimize over {(x,y):x in [0,10], y in [0,10-x]}
        f=lambda x,y: -x*(10-x)*y*(20-y)
        (x, y), fmin = minimizer_convex(f, [(0, 10), (0, lambda x: 10-x)], x0=(5, 5), verbose=verbose)
        print("best: f({},{})={}".format(x, y, fmin))
        g=lambda p: f(p[0], p[1])
        (x, y) = minimize_constrained(g, [lambda p:p[0], lambda p: 10-p[0], lambda p: p[1], lambda p:10-p[0]-p[1]], [1
        print("best over reals: f({},{})={}".format(x, y, f(x, y)))

    class MATZOV_Orig:
        """ """

        C_prog = 1.0 / (1 - 2.0 ** (-0.292))  # p.37
        C_mul = 32**2  # p.37
        C_add = 5 * 32  # guessing based on C_mul

        @classmethod
        def T_fftf(cls, k, p):
            """
            The time complexity of the FFT in dimension ‘k‘ with modulus ‘p‘.

            :param k: Dimension
            :param p: Modulus ≥ 2

            """
            return cls.C_mul * k * p ** (k + 1)  # Theorem 7.6, p.38

        @classmethod
        def T_tablef(cls, D):
            """
            Time complexity of updating the table in each iteration.

            :param D: Number of nonzero entries

            """
            return 4 * cls.C_add * D  # Theorem 7.6, p.39

        @classmethod
        def Nf(cls, params, m, beta_bkz, beta_sieve, k_enum, k_fft, p):
            """
            Required number of samples to distinguish with advantage.

            :param params: LWE parameters
            :param m:
```

```
        :param beta_bkz: Block size used for BKZ reduction
        :param beta_sieve: Block size used for sampling
        :param k_enum: Guessing dimension
        :param k_fft: FFT dimension
        :param p: FFT modulus

        """
        mu = 0.5
        k_lat = params.n - k_fft - k_enum  # p.15

        # p.39
        lsigma_s = (
            params.Xe.stddev ** (m / (m + k_lat))
            * (params.Xs.stddev * params.q) ** (k_lat / (m + k_lat))
            * sqrt(4 / 3.0)
            * sqrt(beta_sieve / 2 / pi / e)
            * deltaf(beta_bkz) ** (m + k_lat - beta_sieve)
        )

        # p.29, we're ignoring O()
        N = (
            exp(4 * (lsigma_s * pi / params.q) ** 2)
            * exp(k_fft / 3.0 * (params.Xs.stddev * pi / p) ** 2)
            * (k_enum * cls.Hf(params.Xs) + k_fft * log(p) + log(1 / mu))
        )

        return RR(N)

    @staticmethod
    def Hf(Xs):
        return RR((1 / 2 + log(sqrt(2 * pi) * Xs.stddev)) / log(2.0)) # old bad formula

    @classmethod
    def cost(
        cls,
        beta,
        params,
        m=None,
        p=2,
        k_enum=0,
        k_fft=0,
        beta_sieve=None,
        red_cost_model=red_cost_model_default
    ):
        """
        Theorem 7.6

        """

        if m is None:
            m = params.n

        k_lat = params.n - k_fft - k_enum  # p.15

        # We assume here that β_sieve ≈ β
        N = cls.Nf(
            params,
            m,
            beta,
            beta_sieve if beta_sieve else beta,
            k_enum,
            k_fft,
            p,
        )
        rho, T_sample, _, beta_sieve = red_cost_model.short_vectors(
            beta, N=N, d=k_lat + m, sieve_dim=beta_sieve
        )
```

```
        H = cls.Hf(params.Xs)
        T_guess = RR(
            (2 ** (k_enum * H)) # old bad formula
            * (cls.T_fftf(k_fft, p) + cls.T_tablef(N))
        )

        cost = Cost(rop=T_sample + T_guess, problem=params)
        cost["red"] = T_sample
        cost["guess"] = T_guess
        cost["beta"] = beta
        cost["p"] = p
        cost["k_enum"] = k_enum
        cost["k_fft"] = k_fft
        cost["beta_"] = beta_sieve
        cost["N"] = N
        cost["m"] = m

        return cost

    def __call__(
        self,
        params: LWEParameters,
        red_cost_model=red_cost_model_default,
        log_level=1,
        use_optimizer=True,
        **kwargs
    ):
        """
        Optimizes cost of dual attack as presented in [Matzov22]_.

        :param params: LWE parameters
        :param red_cost_model: How to cost lattice reduction

        The returned cost dictionary has the following entries:

        - ``rop``: Total number of word operations (≈ CPU cycles).
        - ``red``: Number of word operations in lattice reduction and
                   short vector sampling.
        - ``guess``: Number of word operations in guessing and FFT.
        - ``β``: BKZ block size.
        - ``ζ``: Number of guessed coordinates.
        - ``t``: Number of coordinates in FFT part mod ``p``.
        - ``d``: Lattice dimension.

        """
        params = params.normalize()

        if use_optimizer:
            # note: parameter ordre is important!
            # we put beta last because the range is large so it's better to change it last to
            # quickly converge to a solution, same fo k_fft
            # those go after p because a change in p will reset all the values after
            # k_enum is small so it's better to put it first, for the smallest number of iterations
            def eval_params(k_enum, p, k_fft, beta):
                cost = self.cost(beta, params, k_enum=k_enum, p=p, k_fft=k_fft,
                                 red_cost_model=red_cost_model)
                return log(cost["rop"], 2)
            # a good hint can speed up the search massively
            hint = [10, 4, params.n//10, params.n - 200]
            opt, _ = minimizer_convex(eval_params,
                                      [(0, 50),(2, params.q),(1, params.n//5),(40,params.n)],
                                      x0=hint,
                                      verbose=False,
                                      names=["k_enum", "p", "k_fft", "beta"])
            (k_enum, p, k_fft, beta) = opt
            best_cost = self.cost(beta, params, k_enum=k_enum, p=p, k_fft=k_fft,
                                  red_cost_model=red_cost_model)
            return best_cost
```

```
        else:
            for p in early_abort_range(2, params.q):
                for k_enum in early_abort_range(0, params.n, 5):
                    for k_fft in early_abort_range(0, params.n - k_enum[0], 5):
                        with local_minimum(
                            40, params.n, log_level=log_level + 4
                        ) as it:
                            for beta in it:
                                cost = self.cost(
                                    beta,
                                    params,
                                    p=p[0],
                                    k_enum=k_enum[0],
                                    k_fft=k_fft[0],
                                    red_cost_model=red_cost_model,
                                )
                                it.update(cost)
                            Logging.log(
                                "dual",
                                log_level + 3,
                                f"t: {k_fft[0]}, {repr(it.y)}",
                            )
                            k_fft[1].update(it.y)
                    Logging.log(
                        "dual", log_level + 2, f"ζ: {k_enum[0]}, {repr(k_fft[1].y)}"
                    )
                    k_enum[1].update(k_fft[1].y)
                Logging.log("dual", log_level + 1, f"p:{p[0]}, {repr(k_enum[1].y)}")
                p[1].update(k_enum[1].y)
            Logging.log("dual", log_level, f"{repr(p[1].y)}")
            return p[1].y

    class Bet_Base:
        def set_params(self, params, p, k_enum, k_code):
            self.p_tau_tilde = 0 # compute this
            self.bet_set_size = 1 # compute that
            self.is_valid = False # set to False if the parameters don't allow for bets
            raise Exception("you must implement this function")

        @classmethod
        def min_k_code(cls, params, k_enum):
            return 0 # minimum valid value of k_code given k_enum

        @classmethod
        def max_k_code(cls, params, k_enum):
            return params.n - k_enum - 1 # maximum valid value of k_code given k_enum

        def log_to_cost(self, cost):
            # use this function add debugging stuff to cost if you want
            pass

    class Bet_senum_all_zero(Bet_Base):
        def set_lwe_params(self, lwe_params):
            assert lwe_params.Xs.tag == "CenteredBinomial", "this code only works with CenteredBinomial"
            self.lwe_params = lwe_params
            self.k = self.lwe_params.Xs.bounds[1] # ugly hack to get parameter
            self.p0 = binomial(2 * self.k, self.k) / 2**(2*self.k)
            # for each length n (which will be k_enum+k_code), we precompute the threshold
            # v0(n) such that
            # Pr[|s|_0 >= v0(n)] >= 3/4
            self.v0_tbl = {}
            for n in range(1, 300):
                # start with v0=0, then Pr[]=1
                # and increase v0, decreasing Pr[] by Pr[|s|_0=v0]=binomial(n,v0)*2^(-2n)
                prob = RR(1)
                v0 = 0
                while prob >= 3/4:
                    prob -= binomial(n, v0) * self.p0**v0 * (1-self.p0)**(n-v0)
```

```
                v0 += 1
            # go back to previous value
            v0 -= 1
            self.v0_tbl[n] = v0
        #print("k={} -> v0_tbl={}".format(self.k, self.v0_tbl))

        # precompute minimum value of k_code (stupid brute force)
        self.min_k_code_tbl = {}
        for k_enum in range(0, 50):
            k_code = k_enum
            self.is_valid = False
            while not self.is_valid:
                k_code += 1
                self.set_params(None, k_enum, k_code)
            self.min_k_code_tbl[k_enum] = k_code
        #print("k={} -> min_k_code_tbl={}".format(self.k, self.min_k_code_tbl))

    def set_params(self, p, k_enum, k_code):
        self.v0 = self.v0_tbl[k_enum + k_code]
        self.p_tau_tilde = RR(binomial(self.v0, k_enum) / binomial(k_enum + k_code, k_enum))
        self.bet_set_size = 1
        self.is_valid = (self.v0 >= k_enum)

    def log_to_cost(self, cost):
        cost["Xs_p0"] = self.p0
        cost["Xs_k"] = self.k
        cost["bet_v0"] = self.v0

    def min_k_code(self, k_enum):
        return self.min_k_code_tbl[k_enum]

class MATZOV_Mod_Code_Prange:
    """ """

    C_prog = 1.0 / (1 - 2.0 ** (-0.292))  # p.37
    C_mul = 32**2  # p.37
    C_add = 5 * 32  # guessing based on C_mul

    @classmethod
    def T_fftf(cls, k, p):
        """
        The time complexity of the FFT in dimension k with modulus p.
        """
        if (p & (p-1)) == 0: # power of two
            return cls.C_mul * k * p ** (k)
        else:
            return cls.C_mul * k * p ** (k + 1)

    @classmethod
    def T_tablef(cls, D):
        """
        Time complexity of updating the table in each iteration.
        """
        return 4 * cls.C_add * D

    @classmethod
    def T_decode(cls, N, p, k_fft):
        """
        Time complexity of decoding a polar code
        """
        return 2 * N * p * log(p, 2) * k_fft * log(k_fft, 2)

    @classmethod
    def Nf(cls, params, m, beta_bkz, beta_sieve, k_enum, k_fft, k_code, p, mu, bet_set_size):
        k_lat = params.n - k_code - k_enum  # p.15

        lsigma_s = (
            params.Xe.stddev ** (m / (m + k_lat))
```

```
            * (params.Xs.stddev * params.q) ** (k_lat / (m + k_lat))
            * sqrt(4 / 3.0)
            * sqrt(beta_sieve / 2 / pi / e)
            * deltaf(beta_bkz) ** (m + k_lat - beta_sieve)
        )

        N = RR(
            exp(4 * (lsigma_s * pi / params.q) ** 2)
            * exp(k_code / 3.0 * (params.Xs.stddev * pi / p) ** 2)
            * exp(2*pi/e * params.Xs.stddev**2 * p**(-2*k_fft/k_code) * k_code)
            * (log(bet_set_size) + k_fft * log(p) + log(1 / mu))
        )

        return N

    @staticmethod
    def Hf(Xs):
        return RR((1 / 2 + log(sqrt(2 * pi) * Xs.stddev) + log(coth(pi**2 * Xs.stddev**2))) / log(2.0))

    @classmethod
    def cost(
        cls,
        beta,
        params,
        m=None,
        p=2,
        k_enum=0,
        k_fft=0,
        k_code=0,
        Bet=None, # if Bet is none, use guessing complexity
        beta_sieve=None,
        red_cost_model=red_cost_model_default,
        nu=0.5 # success proba
    ):
        """
        Theorem 7.6

        """
        assert 2 <= p and p <= params.q, "make sure that 2 <= p <= q"
        assert k_fft is not None, "you need to provide k_fft"
        assert k_enum is not None, "you need to provide k_fft"
        assert k_fft <= k_code, "k_fft needs to be smaller than k_code"
        assert k_code + k_enum < params.n, "k_code + k_enum needs to be < n"

        k_lat = params.n - k_code - k_enum

        # this is the optimal dimension of the dual lattice to find short short vectors
        # note that our lattice has dimension m+k_lat so we need to subtract k_lat
        # also be careful that these is a scaling factor alpha that changes the determinant
        # and hence the formule for the optimal m
        alpha = params.Xe.stddev / params.Xs.stddev
        m = int(ceil(sqrt(k_lat * log(params.q/alpha) / log(deltaf(beta))))) - k_lat

        if Bet is None:
            # guessing complexity
            mu = nu
            H = cls.Hf(params.Xs)
            T_enum = RR(2 ** (k_enum * H))
            Nenum = T_enum
        else:
            # compute \tilde{p}_tau
            p_tau_tilde = RR(Bet.p_tau_tilde)
            # compute max mu
            mu = RR(nu * p_tau_tilde / (nu * p_tau_tilde + 2 - nu))
            # compute min R
            R = RR(log(nu/2) / (log(1-p_tau_tilde) + log(1-mu)))
            T_enum = R * Bet.bet_set_size
            Nenum = Bet.bet_set_size
```

```
        # We assume here that β_sieve ≈ β
        N = cls.Nf(
            params,
            m,
            beta,
            beta_sieve if beta_sieve else beta,
            k_enum,
            k_fft,
            k_code,
            p,
            mu,
            Nenum
        )
        rho, T_sample, _, beta_sieve = red_cost_model.short_vectors(
            beta, N=N, d=k_lat + m, sieve_dim=beta_sieve
        )

        T_guess = RR(
            T_enum * (cls.T_fftf(k_fft, p) + cls.T_tablef(N) + cls.T_decode(N, p, k_fft))
        )

        rop = T_sample + T_guess

        cost = Cost(rop=T_sample + T_guess, problem=params)
        cost["red"] = T_sample
        cost["guess"] = T_guess
        cost["beta"] = beta
        cost["p"] = p
        cost["k_enum"] = k_enum
        cost["k_fft"] = k_fft
        cost["k_code"] = k_code
        cost["beta_"] = beta_sieve
        cost["N"] = N
        cost["m"] = m
        cost["mu"] = mu
        if Bet is not None:
            cost["p_tau_tilde"] = p_tau_tilde
            cost["bet_set_size"] = Bet.bet_set_size
            cost["R"] = R
            Bet.log_to_cost(cost)

        return cost

    def __call__(
        self,
        params: LWEParameters,
        red_cost_model=red_cost_model_default,
        log_level=1,
        use_optimizer=True,
        use_2p2 = True, # p is a power of two
        Bet = None,
    ):
        """
        Optimizes cost of dual attack
        """
        params = params.normalize()
        assert use_optimizer, "old optimizer is not supported"

        if Bet is not None:
            Bet.set_lwe_params(params)

        # note: parameter ordre is important!
        # we put beta last because the range is large so it's better to change it last to
        # quickly converge to a solution, same for k_fft
        # those go after p because a change in p will reset all the values after
        # k_enum is small so it's better to put it first, for the smallest number of iterations
        def eval_params_(k_enum, p, k_code, k_fft, beta):
```

```
            if use_2p2:
                p = 2**p
            if Bet is not None:
                Bet.set_params(p, k_enum, k_code)
                # if p_tau_tilde is zero, bad value, return infinity
                if not Bet.is_valid:
                    return oo
            cost = self.cost(beta, params, k_enum=k_enum, p=p, k_fft=k_fft, k_code=k_code,
                             red_cost_model=red_cost_model, Bet=Bet)
            return cost
        def eval_params(k_enum, p, k_code, k_fft, beta):
            cost = eval_params_(k_enum, p, k_code, k_fft, beta)
            return log(cost["rop"], 2)
        # a good hint can speed up the search massively
        hint = [10, 3 if use_2p2 else 256, params.n//10, params.n//20, params.n - 200]
        if Bet is None:
            min_k_code_fn = lambda k_enum,p: 1
        else:
            min_k_code_fn = lambda k_enum,p: Bet.min_k_code(k_enum)
        opt, v = minimizer_convex(eval_params,
                             [(1, 50), # k_enum
                             (2, int(floor(math.log2(params.q))) if use_2p2 else params.q), # p
                             (min_k_code_fn, params.n//2), # k_code
                             (1, lambda k_enum,p,k_code:k_code), # k_fft
                             (40,params.n-10)], # beta
                             x0=hint,
                             verbose=False,
                             names=["k_enum", "p", "k_code", "k_fft", "beta"])
        return eval_params_(*opt)

def get_reduction_cost_model(nn):
    matzov_nns={
        "CN": "list_decoding-naive_classical",
        "CC": "list_decoding-classical",
    }
    if nn in matzov_nns:
        return RC.MATZOV.__class__(nn=matzov_nns[nn])
    elif nn == "C0":
        return RC.ADPS16
    else:
        raise Error("unknown cost model '{}'".format(nn))

def runall(
    schemes=(
        Kyber512,
        Kyber768,
        Kyber1024,
        LightSaber,
        Saber,
        FireSaber,
        # TFHE630,
        # TFHE1024,
    ),
    nns=(
        "CC",
        "CN",
        "C0"
    ),
    attack = "matzov_code_prange",
    use_optimizer = True,
    bet = "senum_all_zero",
):
    results = {}

    obj = None
    Bet = None
    if attack == "matzov":
        obj = MATZOV_Orig()
```

```
    elif attack == "matzov_code":
        obj = MATZOV_Code()
    elif attack == "matzov_code_prange":
        obj = MATZOV_Mod_Code_Prange()
        if bet == "senum_all_zero":
            Bet = Bet_senum_all_zero()
        elif bet == "guess_senum":
            Bet = None
        else:
            raise RuntimeError("unknown bet '{}'".format(bet))
    else:
        raise RuntimeError("unknown attack '{}'".format(attack))

    try:
        for scheme in schemes:
            results[scheme] = {}
            print(f"{repr(scheme)}")
            for nn in nns:
                red_mod = get_reduction_cost_model(nn)
                cost = obj(scheme, red_cost_model=red_mod, use_optimizer=use_optimizer, Bet=Bet)
                results[scheme][nn] = cost
                print(f" nn: {nn},  cost: {repr(cost)}")
    except KeyboardInterrupt:
        print("computation interrupted, partial results will be returned")
        pass
    except Exception as e:
        import traceback
        traceback.print_exception(e)
        pass

    return results

def results_table(results, fmt=None):
    import tabulate

    rows = []

    def pp(cost):
        return round(log(cost["rop"], 2), 1)

    # collect models
    nns = set()
    for costs in results.values():
        nns |= set(costs.keys())
    nns = list(sorted(nns))

    for scheme, costs in results.items():
        row = [ scheme.tag ] + [pp(costs[nn]) if nn in costs else "n/a" for nn in nns]
        rows.append(row)
    if fmt is None:
        return rows
    else:
        import tabulate
        return tabulate.tabulate(
            rows,
            headers=["Scheme"] + nns,
            tablefmt="latex_booktabs",
            floatfmt=".1f",
        )

def parameter_tables(results, fmt=None):
    # our new code uses variable "k_code", but the old Matzov code uses "p", this code supports both
    opt_vars = []
    # look at the first entry
    first_code = next(iter(next(iter(results.values())).values()))
    for x in ["k_code", "p"]:
        if getattr(first_code, x, None) is not None:
            opt_vars.append(x)
```

```
# collect models
nns = set()
for costs in results.values():
    nns |= set(costs.keys())
nns = list(sorted(nns))

def pp(cost):
    return round(log(cost["rop"], 2), 1)
res = {}
for nn in nns:
    rows = []
    for scheme, costs in results.items():
        if nn in costs:
            c = costs[nn]
            row = [
                scheme.tag,
                pp(c),
                c["m"],
                c["beta"],
                c["beta_"],
                c["k_enum"],
                c["k_fft"],
            ]
            for x in opt_vars:
                row.append(c[x])
            rows.append(row)
    if fmt is None:
        res[nn] = rows
    else:
        import tabulate
        res[nn] = tabulate.tabulate(
            rows,
            headers=[
                "Scheme",
                "Attack",
                "m",
                "beta_1",
                "beta_2",
                "k_enum",
                "k_fft",
            ] + opt_vars,
            tablefmt="latex_booktabs",
            # floatfmt=".1f",
        )
return res
```