Cold Boot Attacks on Ring and Module LWE Keys Under the NTT

Martin R. Albrecht, Amit Deo and Kenneth G. Paterson*

Information Security Group, Royal Holloway, University of London,
martin.albrecht@royalholloway.ac.uk,
amit.deo.2015@rhul.ac.uk,kenny.paterson@rhul.ac.uk

Abstract. In this work, we consider the ring- and module- variants of the LWE problem and investigate cold boot attacks on cryptographic schemes based on these problems, wherein an attacker is faced with the problem of recovering a scheme's secret key from a noisy version of that key. The leakage resilience of cryptography based on the learning with errors (LWE) problem has been studied before, but there are only limited results considering the parameters observed in cold boot attack scenarios. There are two main encodings for storing ring- and module-LWE keys, and, as we show, the performance of cold boot attacks can be highly sensitive to the exact encoding used. The first encoding stores polynomial coefficients directly in memory. The second encoding performs a number theoretic transform (NTT) before storing the key, a commonly used method leading to more efficient implementations. We first give estimates for a cold boot attack complexity on the first encoding method based on standard algorithms; this analysis confirms that this encoding method is vulnerable to cold boot attacks only at very low bit-flip rates. We then show that, for the second encoding method, the structure introduced by using an NTT is exploitable in the cold boot setting: we develop a bespoke attack strategy that is much cheaper than our estimates for the first encoding when considering module-LWE keys. For example, at a 1% bit-flip rate (which corresponds roughly to what can be achieved in practice for cold boot attacks when applying cooling), a cold boot attack on Kyber KEM parameters has a cost of 2^{43} operations when the second, NTT-based encoding is used for key storage, compared to 2^{70} operations with the first encoding. On the other hand, in the case of the ring-LWE-based KEM, New Hope, the cold boot attack complexities are similar for both encoding methods.

Keywords: Cold boot attack · Lattice reduction · Number theoretic transform · Post-quantum cryptography · Ring learning with errors · Module learning with errors

1 Introduction

One of the attractive features of the Learning with Errors problem (LWE) [Reg05] is its "leakage resilience" [DGK+10, BG10, BL14] which roughly states that the difficulty of the

*The research of Albrecht was supported by EPSRC grant EP/P009417/1 (Bit Security of Learning with Errors for Post-Quantum Cryptography and Fully Homomorphic Encryption) and by the European Union Horizon 2020 Research and Innovation Program Grant 780701 (PROMETHEUS). The research of Deo was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1). The research of Paterson was supported by EPSRC Grant EP/M013472/1 (UK Quantum Technology Hub for Quantum Communications Technologies), by EPSRC Grants EP/K035584/1 and EP/P009301/1 (Centre for Doctoral Training in Cyber Security at Royal Holloway), and by the European Union Horizon 2020 Research and Innovation Program Grant 780701 (PROMETHEUS).

problem deteriorates only gradually as information about the secret is leaked. Indeed, the LWE problem has been shown to remain hard even when an attacker knows many bits of the secret [AGV09] or when random inner products with the secret vector are known [Pie12]. With efficiency in mind, many systems proposed for practical use are based on the related ring-LWE problem (RLWE) [LPR13a] and module-LWE problem (MLWE) [LS15]. For this setting, fewer results are known. For example, the ring-LWE-based public key encryption scheme from [LPR13b] was recently shown to remain IND-CPA secure when certain information on the private key is leaked [DSGKS17].

A running motivating example in the leakage resilience literature is "cold boot attacks", cf. [AGV09, NS09, DSGKS17]. Cold boot attacks were introduced and studied in the seminal work of Halderman et al. [HSH+09]. Briefly, cold boot attacks rely on the fact that bits in RAM retain their value for some time after power is cut. In order to preserve the value for longer, memory can be cooled to extreme temperatures ($-50°$C) in order to retain a $\rho_0 = 1\%$ bit-flip rate even after a time period of ten minutes. Halderman et al. also noted that bit-flip rates as low as $\rho_0 = 0.17\%$ are possible when liquid nitrogen is used for cooling. Another key observation was that memory has a ground state that the bits will decay to over time, i.e. the noise introduced is very biased. However, it was also noticed that there is a very small but non-zero probability of retrograde bit-flips away from the ground state. It was estimated that these retrograde bit-flips occur at a rate of $\rho_1 \in [0.05 - 0.1\%]$. In a cold boot attack, then, the attacker is assumed to have physical access to a machine shortly after a power down cycle. The attacker proceeds by extracting from memory a noisy version of a scheme's secret key, where a small number of bits have been flipped. The attacker then recovers the key by applying bespoke error correction algorithms.

To date, cold boot attacks have received a significant amount of attention across a range of cryptographic primitives including a variety of symmetric ciphers [HSH+09, Tso09, KY10, AC11], RSA [HS09, HMM10, PPS12], discrete log systems [PS15] and, most recently, NTRU [PV17]. However, the literature so far contains no dedicated analysis of cold boot attacks against LWE-based cryptographic primitives.

Establishing the resilience to side-channel attacks of LWE-based schemes is gaining significance in light of these schemes being on the brink of widespread adoption. Firstly, LWE/RLWE/MLWE assumptions are popular candidates for post-quantum cryptography. For example, many proposals submitted to NIST's post-quantum standardisation process are based on these assumptions [SSZ17, Ham17, DKRV17, GMZB+17, BAA+17, PAA+17, PHAM17, SAL+17, LLJ+17, ZJGS17, Saa17, NAB+17, SPL+17, DTGW17, SAB+17, LDK+17, LLKN17]. We note that NIST considers resistance to side-channel attacks as a worthwhile, albeit secondary security feature: "schemes that can be made resistant to side-channel attack at minimal cost are more desirable than those whose performance is severely hampered by any attempt to resist side-channel attacks. We further note that optimised implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not" [Nat16]. Secondly, efforts to standardise homomorphic encryption schemes have gained traction, with the first white papers being issued [CCD+17, ACC+17, BDH+17]. The homomorphic encryption schemes being considered for standardisation are all based on the RLWE assumption.

Contributions and road map. We consider the resistance of RLWE- and MLWE-based schemes to cold boot attacks. In light of the leakage resilience of LWE mentioned above, we investigate how *cold boot leakage* of secrets stored as polynomial coefficients affects the hardness of the LWE problem. We show that for moderate cold boot error rates the resulting problem is considerably easier to solve than the side-channel-free RLWE/MLWE instances from which it is derived; for this analysis, we simply apply standard security estimates. However, we note that this analysis does not apply to many schemes as specified

Table 1: Cold boot attacks on Kyber KEM keys stored in the NTT domain with ρ_0, ρ_1 the cold boot bit-flip rates. The column "cost" gives the cost of recovering 256 components of the secret in terms of the number of lattice points visited during enumeration (≈ 100 CPU cycles each). The attack can be repeated to recover all 768 components. The column "rate" shows the overall success rate $1-(1-p_0)^2$ for recovering 256 components of the secret, cf. Section 7. We also give the costs of a cold boot attack when the secret key is stored in the time domain in the column "non-NTT", cf. Section 4. In that case, the success rate is always expected to be close to 100%.

bit-flip rates		NTT		non-NTT
ρ_0	ρ_1	cost	rate	cost
0.2%	0.1%	$3 \cdot 2^{21.1}$	95%	$2^{38.7}$
0.5%	0.1%	$3 \cdot 2^{33.1}$	87%	$2^{51.6}$
1.0%	0.1%	$3 \cdot 2^{43.3}$	91%	$2^{70.3}$
1.4%	0.1%	$3 \cdot 2^{53.6}$	91%	$2^{89.2}$
1.7%	0.1%	$3 \cdot 2^{62.8}$	89%	$2^{100.1}$

and implemented in practice. In particular, many schemes, e.g. [PAA+17, SAL+17, LLJ+17, ZJGS17, Saa17, SPL+17, DTGW17, SAB+17, LDK+17, CLP17], make use of a power-of-two cyclotomic ring $\mathbb{Z}[x]/(x^n+1)$. This ring is amenable to performing multiplications using a (negacyclic) number theoretic transform (NTT) with complexity $\mathcal{O}(n \log_2 n)$. Adopting the language of the Fourier transform from which the NTT is derived, the expensive step of an NTT computation is to transform the inputs from the time domain into the frequency domain and back; the actual multiplication takes only $\mathcal{O}(n)$ elementary operations. Thus, it is beneficial to keep intermediate values in the frequency domain. For example, the Kyber specification [SAB+17] directly specifies the secret key in the frequency domain. This implementation detail dramatically alters the landscape for cold boot attacks on RLWE/MLWE-based schemes that specify the use of an NTT: now, a cold boot attacker is confronted with the problem of "decoding a noisy NTT", i.e. recovering the input to an NTT given a noisy output. This problem is well-defined in our setting since the sought after input is small compared to the modulus q for which the NTT is specified.

While our attack in principle applies to all RLWE/MLWE schemes using the NTT and storing secret keys in the frequency domain, we use a running example of the default Kyber parameters [SAB+17] for concreteness. To start off, in Section 3, we establish the decoding cost for cold boot attacks when the NTT is not used for secret key storage, and obtain a solving cost of 2^{70} operations for $\rho_0 = 1\%, \rho_1 = 0.1\%$ bit-flip rates. We then introduce the "cold boot NTT problem" in Section 4. This accurately captures the information available to the adversary in a cold boot attack on Kyber and related schemes that store the private key using an NTT. We then develop a practical attack with a cost of roughly 2^{43} operations for the aforementioned bit-flip rates by exploiting properties of the NTT. We summarise our findings in Table 1. In addition to the running example of Kyber, we also analyse New Hope KEM [PAA+17] to give an idea of the attack performance on a RLWE-based scheme. The results for New Hope are slightly different to the Kyber results and are summarised in Table 2. In particular, for the bit-flip rates considered, the attack complexities on New Hope when using the NTT for key storage are comparable to the case where the NTT is not used.

Our attack proceeds as follows. In Section 5 we show how to reduce the dimension of our NTT cold boot problem by using a divide and conquer approach that is inspired by the standard recursive formula for the NTT. We then show that the resulting low dimensional instance is efficiently solvable using a careful application of lattice reduction. What prevents our attack from having trivial complexity is that we encounter LWE-like instances where the secret distribution has a peculiar form. Specifically, each component of the secret can be

Table 2: Cold boot attacks on New Hope KEM. The column "cost" gives the cost of recovering all 1024 components of the secret in terms of the number of lattice points visited during enumeration ($\approx$ 100 CPU cycles each). The column "rate" shows the overall success rate $1-(1-p_0)^2$ for recovering 1024 components of the secret, cf. Section 7. For the columns labelled "non-NTT", see caption of Table 1.

bit-flip rates		NTT		non-NTT
ρ_0	ρ_1	cost	rate	cost
0.17%	0.1%	$2^{48.7}$	84%	$2^{53.7}$
0.25%	0.1%	$2^{60.6}$	81%	$2^{60.0}$
0.32%	0.1%	$2^{70.2}$	81%	$2^{66.1}$

written as the sum of a *small number* of positive/negative powers of two and the secret itself is guaranteed to be sparse.[1] Therefore, in Section 6, we introduce a special attack on LWE with this type of secret. This attack combines guessing of higher-order bits with running an enumeration for a closest vector [LP11, LN13]. Similar combinations of combinatorial and lattice-reduction techniques have been previously considered, e.g. in [HG07]. In particular, our approach is similar to that used in [BCGN17, dBDJdW18] for solving the Mersenne Low Hamming Ratio Search Problem [AJPS17]. However, in contrast to [BCGN17, dBDJdW18], our attack is aided by the fact that we are considering a lattice derived from an NTT matrix. These lattices are highly structured and display a behaviour very far from that observed for random lattices. Thus, on the one hand, we cannot apply standard estimates for various quantities involved in lattice reduction. On the other hand, performing lattice reduction on the lattice bases that we encounter turns out to be easier than expected. Thus, the cost of our lattice-reduction is not derived from (standard) estimates but based on experimental evidence obtained using [FPL17, FPY18]. We describe the relevant properties of these lattices and the lattice reduction/enumeration step of our attack in Section 7. Finally, we report on the overall cost of our attack in Section 8.

For completeness, we include an overview of other possible cold boot attack techniques (meet-in-the-middle and Gröbner bases) in the appendix. We also consider there an alternative approach to solving the cold boot problem based on Blahut's Theorem and the Berlekamp-Massey algorithm [Mas69]. This approach succeeds when the bit-flip rate is low and where the secret key is guaranteed to have low Hamming weight when compared to the ring dimension. In particular, if the secret has Hamming weight w and an attacker has access to $2w$ consecutive clean components of the secret, then the full secret can be derived at a trivial cost.

Discussion. While our attacks are a far cry from the impressive bit-flip rates that can be handled for other primitives such as RSA and AES, they highlight that cold boot attacks apply to RLWE/MLWE-based schemes. Our results show that use of the NTT makes cold boot attacks easier for the MLWE-based Kyber KEM. However, for the RLWE-based scheme New Hope KEM, the complexity of cold boot attacks in the non-NTT and NTT cases is roughly the same for the bit-flip rates we considered. One reason for this is that our NTT-based attack allows us to consider each ring element of an MLWE secret individually which reduces the dimension of the cold boot problem. This is not possible in the case of RLWE where the secret key is a single ring element with a large dimension.[2] This fact also explains why the bit-flip rates that our attack effectively handles are lower for New Hope.

[1] The positive powers of two correspond to a $0 \to 1$ bit-flip in a cold boot scenario whereas the negative powers correspond to bit-flips in the other direction.

[2] We note that while LWE can be viewed as MLWE with $n = 1$, this behaviour does not translate. For this divide-and-conquer approach to be successful, we require unique decoding for each ring element. This condition is not satisfied in the case of LWE.

For Kyber KEM, our results suggest that vulnerability to cold boot attacks can be mitigated by storing the secret in the time domain instead of the frequency domain. This counter measure would increase decryption time in a typical IND-CCA setting by a factor of at most two as such a conversion from the time to frequency domain must take place already due to the re-encryption step.[3] However, such a counter measure would not completely rule out cold boot attacks: for bit-flip rates of $\rho_0 = 0.2\%$ the resulting MLWE instance is still relatively easy to solve using the methods of Section 3. This countermeasure does not appear to be relevant in the case of New Hope according to Table 2 where the complexity of attacking a New Hope key remains comparable whether the NTT is used for key storage or not. However, future work may propose better algorithms for solving the cold boot NTT decoding problem.

2 Preliminaries

For positive real y, we write $\lfloor y \rfloor$ to denote the integer part of y, $\lceil y \rceil$ to denote the smallest positive integer larger than y and $\lfloor y \rceil$ to denote the rounding of y to the nearest integer (where we round down in the case of a tie). We denote the integers modulo q as $\mathbb{Z}_q$. We use subscripts to reference individual entries of vectors e.g. a_i. We start counting at zero. In the case where we have a polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$, we often identify it with its vector of coefficients $(a_0, \ldots, a_{n-1})$. Our treatment of a as either a polynomial or a vector should be clear from the context. We use the notation $s \leftarrow \mathcal{D}$ to mean that s is an element sampled from the distribution $\mathcal{D}$. If s is a k-dimensional vector, then $s \leftarrow (\mathcal{D})^k$ denotes that each entry in s is drawn independently from the distribution $\mathcal{D}$. If $\mathcal{S}$ is a finite set, we use the notation $s \leftarrow \mathcal{S}$ to denote that s is an element sampled from the uniform distribution over $\mathcal{S}$.

Let q be a prime such that an $2n^{th}$ primitive root of unity γ exists, and set $\omega = \gamma^2$. The negacyclic number theoretic transform (NTT) in dimension n will be defined as the linear function $\mathsf{NTT} : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$ given by

$$\mathsf{NTT}_n(a)_i := \sum_{j=0}^{n-1} \gamma^j \omega^{ij} a_j \bmod q.$$

This transform allows for fast polynomial multiplication in rings of the form $\mathbb{Z}_q[x]/(x^n + 1)$ where n is a power of two [SS71, Win96]. In general, $\hat{a}$ will be used as shorthand for the NTT of a and we often drop the subscript n when its value is clear from the context. The inverse negacyclic NTT is given by

$$\mathsf{NTT}_n^{-1}(\hat{a})_i := n^{-1}\gamma^{-i} \sum_{j=0}^{n-1} \omega^{-ij} \hat{a}_j.$$

We need the following result, whose proof is an easy exercise:

Proposition 1. *Let $X \leftarrow \{\pm 2^0, \pm 2^1, \pm 2^2, \ldots, \pm 2^{\ell-1}\}$. We have*

$$\mathbb{E}[X] = 0 \quad \text{and} \quad \mathbb{E}[X^2] = \frac{4^\ell - 1}{3\,\ell}.$$

Furthermore, let $Y = (X_0, \ldots, X_{n-1})$ where each $X_i \leftarrow \{\pm 2^0, \pm 2^1, \pm 2^2, \ldots, \pm 2^{\ell-1}\}$. Then the expected squared norm of Y is

$$\mathbb{E}[\|Y\|^2] = n\left(\frac{4^\ell - 1}{3\,\ell}\right).$$

[3] Note that the NTT is typically not the most expensive operation in RLWE/MLWE-based schemes, thus the factor of two is conservative. There is also a conversion from the frequency to time domain during decryption. However, in an MLWE setting this operates on one ring element as opposed to k ring elements, which is the dimension of the secret s.

2.1 LWE definitions

We will be using the definition of ring-LWE (RLWE) discussed in [LPR13b] since it best represents practical use. The reason for this is that the original RLWE definition [LPR10] (and the definition of module-LWE (MLWE) from [LS15]) uses a continuous error distribution which is inconvenient in practice. We restrict to rings of the form $R = \mathbb{Z}[x]/(x^n + 1)$ where n is a power of two. We also define $R_q := R/(qR)$.

Definition 1 (Ring-LWE distribution)**.** For a "secret" $s \in R_q$ and an error distribution χ over R, a sample from the ring-LWE distribution $A_{s,\chi}$ over $R_q \times R_q$ is generated by choosing $a \leftarrow R_q$ uniformly, $e \leftarrow \chi$ and outputting $(a, a \cdot s + e \bmod qR)$.

Definition 2 (Search ring-LWE problem)**.** The search ring-LWE problem with secret distribution $\mathcal{D}$ over R entails recovering s from arbitrarily many samples of $A_{s,\chi}$ where $s \leftarrow \mathcal{D}$.

We note that in practice, we usually have a restriction on the number of samples. In the module-LWE definitions below, k will be a positive integer representing the module "rank". It is understood that if $a := (a^{(0)}, \ldots, a^{(k-1)}) \in (R_q)^k$ and $s := (s^{(0)}, \ldots, s^{(k-1)}) \in (R_q)^k$, then $a \cdot s = \sum_{i=0}^{k-1} a^{(i)} s^{(i)} \in R_q$.

Definition 3 (Module-LWE distribution)**.** For a "secret" $s \in (R_q)^k$ and error distribution χ over R, a sample from the module-LWE distribution $A_{k,s,\chi}$ over $(R_q)^k \times R_q$ is generated by choosing $a \leftarrow (R_q)^k$ uniformly, $e \leftarrow (\chi)^k$ and outputting $(a, a \cdot s + e \bmod qR)$.

Definition 4 (Search module-LWE problem)**.** The search module-LWE problem with secret distribution $\mathcal{D}$ over R entails recovering s from arbitrarily many samples of $A_{k,s,\chi}$ where $s \leftarrow (\mathcal{D})^k$.

The decision variant of RLWE challenges an adversary to *distinguish* between samples from $A_{s,\chi}$ and the uniform distribution over $R_q \times R_q$ given that $s \leftarrow \mathcal{D}$. Similarly, decision MLWE is the problem of distinguishing between $A_{k,s,\chi}$ and the uniform distribution over $(R_q)^k \times R_q$ given $s \leftarrow (\mathcal{D})^k$.

2.2 Minimal binary signed digit representation

We will often consider integers in *binary signed digit representation* (BSDR). This representation is reminiscent of binary representation for positive integers, apart from the fact that each individual bit in BSDR has its own sign. For example, $(1, 0, -1)$ is a BSDR of -3 because $-3 = 1 \cdot 2^0 + 0 \cdot 2^1 - 1 \cdot 2^2$. We also have that -3 can be written as $(-1, -1)$ in BSDR. It is clear that integers can have many BSDRs. In order to reduce the number of possibilities, we often consider the *minimal* BSDRs corresponding to the BSDRs with the minimum possible Hamming weight. For example, the minimal BSDR of 31 is $(-1, 0, 0, 0, 0, 1)$. Note that this has a lower Hamming weight than the binary representation of 31 i.e. $(1, 1, 1, 1, 1)$. Even when considering minimal BSDRs, the issue of non-uniqueness can arise. The integer -3 is a simple example of this. One can also consider integers in q-ary signed digit representation (q-SDR). For example, if $q = 3$, a possible q-SDR of the integer 8 would be $(-1, 0, 1)$. Once again these representations are not unique. We extend these definitions to vectors in the obvious way, i.e. by considering vectors component-wise.

2.3 Lattices

We here only briefly recall the definitions relevant to this work. For an introduction to lattices and lattice-based cryptography see [MR09, Pei15]. An n-dimensional lattice

is a discrete subgroup of $\mathbb{R}^n$. A rank m lattice Λ can be written in terms of a basis $\{\vec{b}_0, \ldots, \vec{b}_{m-1}\}$ as

$$\Lambda := \{\vec{x} \in \mathbb{R}^n : \vec{x} = \sum_{i=0}^{m-1} z_i \vec{b}_i, z_i \in \mathbb{Z}\}.$$

We only consider full-rank lattices in this work where $m = n$. We can represent the basis as a matrix $B \in \mathbb{R}^{m \times n}$ where each *row* is considered to be a basis vector. The main computational lattice problem that will arise in this work is the *bounded distance decoding* problem (BDD), cf. [LN13]. To define BDD in it simplest form, we will denote the shortest nonzero vector in a lattice Λ as $\lambda_1(\Lambda)$. The BDD problem then asks to find the closest lattice vector to some target point $\vec{t}$ under the guarantee that there exists a lattice vector within distance $\lambda_1(\Lambda)$ from $\vec{t}$.

A common strategy for solving BDD is to first obtain a "high quality" basis for the lattice and then to run Babai's nearest plane algorithm to obtain a solution. At a high level, the most desirable bases for running Babai are short and orthogonal. Obviously, because of the geometry of most lattices, these bases simply do not exist. Due to this fact, definitions of "reduced" bases aim to mimic the notion of a short and orthogonal basis. The well-known BKZ algorithm [Sch87, CN11] outputs a so-called BKZ-reduced basis. It is parametrised by a block size β, indicating at which dimension calls are made to an exact SVP oracle as a subroutine in the algorithm. After performing BKZ-β reduction, the first vector in the transformed lattice basis will have norm $\delta_0^m \cdot \det(\Lambda)^{1/m}$ where $\det(\Lambda)$ is the determinant of the lattice under consideration and the root-Hermite factor δ_0 is a constant based on the block size parameter β. More generally, the quality of a reduced basis B can be expressed by the slope of the logs of the lengths of the vectors $\vec{b}_i^\star$ in the Gram-Schmidt orthogonalisation of B. For random bases, the Geometric Series Assumption (GSA) is commonly assumed to hold:

Definition 5 (Geometric Series Assumption [Sch03])**.** The norms of the Gram-Schmidt vectors after lattice reduction satisfy

$$\|\vec{b}_i^*\| = \alpha^{i-1} \cdot \|\vec{b}_1\| \text{ for some } 0 < \alpha < 1.$$

Combining the GSA with the root-Hermite factor and the fact that $\det(\Lambda) = \prod_{i=1}^{n} \|\vec{b}_i^\star\|$, we get $\alpha = \delta_0^{-2n/(n-1)} \approx \delta_0^{-2}$. Increasing the block-size parameter β of BKZ-β leads to a smaller δ_0 but also leads to an increase in run-time. In this work, we consider the "enumeration regime" where lattice point enumeration is used to realise the exact SVP oracle in dimension β. In this case the running time grows as $\beta^{\Theta(\beta)}$ [Kan83, MW15].

Babai's nearest plane algorithm has been generalised to consider multiple planes [LP11]. This, in turn, can be considered as a form of pruned BDD enumeration [LN13]. In this work, we follow the BDD enumeration approach to solving BDD, i.e. we first compute a high quality basis and then run pruned enumeration to recover the (hopefully) closest vector to our target vector. As is usual, we run enumeration in some sub-dimension and then extend the solution in the projected sub-lattice to a full solution by running Babai's nearest plane algorithm. This is equivalent to picking very small pruning coefficients for the smallest indices. We make use of BKZ and enumeration as implemented in [FPL17, FPY18]. This implementation also features a `Pruning` module, which computes parameters for pruned enumeration.

3 Leakage resilience for Kyber's parameters

As mentioned above, we use the default parameter set of the Kyber KEM [SAB+17], henceforth referred to simply as "Kyber", as the running example. However, we stress that our analysis applies generally to RLWE/MLWE keys as we will see later when the New

Hope KEM [PAA+17] is considered. Kyber relies on the MLWE problem in dimension $k = 3$ over the ring $R_q = \mathbb{Z}_{7681}[x]/(x^{256} + 1)$. It uses a centred binomial error distribution $\mathcal{B}_\eta$ with parameter $\eta = 4$. This distribution has standard deviation $\sqrt{\eta/2} = \sqrt{2}$. In Kyber, the components of the secret also follow $\mathcal{B}_\eta$.

Now, consider the Kyber public key $a, b \coloneqq a \cdot s + e$ with $s_i, e_i \leftarrow \mathcal{B}_\eta$ and assume that, due to some leakage, we are given a noisy version of s, denoted by $\tilde{s} \coloneqq s + \Delta$. Here, the addition is over R_q and Δ is an element of R_q representing bit-flips. This means that each component of Δ should have low Hamming weight when written in minimal BSDR. For illustrative purposes, we will focus on cold boot bit-flip rates of $\rho_0 = 1.0\%$ towards the ground state and a retrograde bit-flip rate of $\rho_1 = 0.1\%$, cf. [HSH+09]. More values are given in Table 1 and estimates for the New Hope KEM are given in Table 2. We consider

$$a \cdot \tilde{s} - b = a \cdot \tilde{s} - a \cdot s - e = a \cdot (\tilde{s} - s) + e = a \cdot \Delta + e \tag{1}$$

which is an MLWE instance for the secret Δ. We note that the conversion works both ways, i.e. an attacker who can find Δ can then solve the above MLWE instance, and thus the two problems are equivalent.

By definition of $\mathcal{B}_\eta$ we have $-\eta \leq s_i \leq \eta$. Thus, s_i fits into four bits (including one sign bit) and we may assume that the secret Δ is both relatively sparse (at least when considered in minimal BSDR) and has components that are bounded by $\eta = 4$ in absolute value. This means that we only need to consider $768 \cdot 4$ bits altogether. We assume that half of these bits are in the ground state and the other half are not. That is, for $\rho_0 = 1.0\%, \rho_1 = 0.1\%$, we obtain a Δ with an expected number of $17 = \lceil (1.0 + 0.1)/100 \cdot 768 \cdot 4/2 \rceil$ non-zero components, each bounded by four in absolute value. According to the LWE estimator from [APS15] the MLWE instance (1) for these parameter sets take $\approx 2^{70.3}$ operations to solve assuming enumeration is used to realise the SVP oracle [CN11].[4] This attack might be improved somewhat by taking into account the *a priori* distribution of the s_i.

4 Cold boot NTT decoding problem

The discussion in the previous section assumes that s is stored in RAM as a vector with small components, allowing a cold boot attacker to obtain a noisy image of s. Yet, as discussed in the introduction, Kyber stores $\hat{s} \coloneqq \mathsf{NTT}_n(s)$ instead of s. Thus, a cold boot attacker *does not* encounter a noisy version of s but a noisy version of $\hat{s}$. In other words, the costs derived in Section 3 are immaterial for a real-world attack on Kyber. In particular, the decoding problem encountered during a cold boot attack on M/RLWE-based schemes utilising an NTT, is as follows:

Definition 6 (Cold boot NTT decoding problem)**.** Let NTT be a (negacyclic) NTT of dimension n modulo q, let ξ be some known constant mod q, let s be a vector with some known distribution χ and let Δ be some vector with known distribution ψ. Then the *Cold Boot NTT Decoding Problem* is to recover s given

$$\tilde{\hat{s}} \coloneqq \xi\, \mathsf{NTT}(s) + \Delta.$$

In the definition above, we slightly generalise the cold boot problem encountered by permitting a scaling factor ξ, cf. Section 5. As before, in our setting Δ corresponds to bit-flips which means that each component of Δ should have low Hamming weight when written in minimal BSDR. However, contrary to the discussion in Section 3, the norm of

[4] The following call to the code available at https://bitbucket.org/malb/lwe-estimator was used to establish this cost:

```
sage: f = partial(drop_and_solve, primal_usvp, n=3*256, q=7681, alpha=sqrt(2)*sqrt(2*pi)/7681,
reduction_cost_model=BKZ.CheNgu12, decision=False, postprocess=False)
sage: f(secret_distribution=((-4, 4), ceil((1.0 + 0.1)/100 * 4 * 768/2)))
```

the "noise term" Δ is not necessarily small. By analogy with LWE, it will be convenient to consider the problem with the roles of s and Δ reversed, i.e. to consider the inverse NTT of the above instance. In particular, we will be considering the problem of recovering s or Δ given

$$\tilde{s} := W\Delta + s \tag{2}$$

where W is the inverse (of a possibly scaled by some constant) negacyclic NTT matrix for dimension n, $\tilde{s}$ is known, s is small and Δ is sparse in minimal BSDR. We sometimes write W_n to explicitly indicate the dimension of the NTT.

In a standard LWE setting, the matrix A is uniformly randomly sampled mod q. Indeed, to prevent precomputation attacks, Kyber specifies that a fresh A is computed for s. In contrast, in our decoding problem each instance has the same W which is the matrix representation of an inverse negacyclic NTT. Thus, precomputation attacks become feasible. More importantly, though, this matrix is highly structured and, indeed, the q-ary lattices derived from this matrix do not behave like random lattices. We consider this in Sections 5 and 7.

We note that while we are only given n samples in our decoding problem, the problem is still well defined, despite Δ not being small. This is because Δ is sparse when its components are written in BSDR form. On the other hand, the distribution of Δ implies that standard techniques for solving LWE-like problems need to be adapted. We consider this in Section 6.

We parametrise the cold boot NTT decoding problem by a parameter κ representing the number of expected bit-flips; explicitly:

$$\kappa \coloneqq \lceil (\rho_0 + \rho_1) \cdot n \cdot \lceil \log_2 q \rceil / 2 \rceil.$$

Finally, we note that, for Kyber, the dimension of the problem is immediately reduced from $n \cdot k = 768$ to $n = 256$ since a single Kyber key gives rise to k independent cold boot problems. It should be noted that this reduction in dimension does not occur when considering RLWE keys since RLWE is effectively MLWE with $k = 1$. For bit-flip rates of 0.17% and 1% in the ground state direction (and 0.1% in the retrograde direction), we expect a total of less than $\lceil (0.17 + 0.1) \cdot 256 \cdot 13/200 \rceil = 5$ and $\lceil (1 + 0.1) \cdot 256 \cdot 13/200 \rceil = 19$ bits to be flipped respectively. Therefore, under these cold boot assumptions, we expect either 5 or 19 unknown bit-flips. Note that in both cases, the number of retrograde bit-flips is approximately 2. The case $\rho_0 = 0.17\%$ can therefore be solved by exhaustive search in $\binom{13\cdot 256/2}{3} \cdot \binom{13\cdot 256/2}{2} \approx 2^{50}$ operations. For the case, $\rho_0 = 1.0\%$, the naive strategy of simply guessing the positions of bit-flips implies an attack of complexity roughly $\binom{13\cdot 256/2}{17} \cdot \binom{13\cdot 256/2}{2} \approx 2^{154}$. This is the case that we will use as our running example.

5 Divide and conquer

It is well known that a 2^n-dimensional Fourier transform can be written in terms of two 2^{n-1}-dimensional Fourier transforms. The same holds for a negacyclic NTT. To aid the presentation of the appropriate formulae, define $\mathbf{g}^{(e)} := (g_0, g_2, \ldots, g_{n-2})$ and $\mathbf{g}^{(o)} := (g_1, g_3, \ldots, g_{n-1})$ for any $\mathbf{g} \in \mathbb{Z}_q^n$. The negacyclic NTT can be shown to satisfy the following relations:

$$2\mathsf{NTT}_{n/2}(\mathbf{g}^{(e)})_i = \mathsf{NTT}_n(\mathbf{g})_i + \mathsf{NTT}_n(\mathbf{g})_{i+n/2} \tag{3}$$

$$2\gamma\omega^i\mathsf{NTT}_{n/2}(\mathbf{g}^{(o)})_i = \mathsf{NTT}_n(\mathbf{g})_i - \mathsf{NTT}_n(\mathbf{g})_{i+n/2} \tag{4}$$

for $i \in \{0, 1, \ldots, n/2 - 1\}$.

Example 1. Consider $n = 8$, given a $2n$-th root of unity γ, we can write the forward negacyclic NTT in matrix form as

$$W_n = \begin{pmatrix} 1 & \gamma & \gamma^2 & \gamma^3 & \gamma^4 & \gamma^5 & \gamma^6 & \gamma^7 \\ 1 & \gamma^3 & \gamma^6 & -\gamma & -\gamma^4 & -\gamma^7 & \gamma^2 & \gamma^5 \\ 1 & \gamma^5 & -\gamma^2 & -\gamma^7 & \gamma^4 & -\gamma & -\gamma^6 & \gamma^3 \\ 1 & \gamma^7 & -\gamma^6 & \gamma^5 & -\gamma^4 & \gamma^3 & -\gamma^2 & \gamma \\ 1 & -\gamma & \gamma^2 & -\gamma^3 & \gamma^4 & -\gamma^5 & \gamma^6 & -\gamma^7 \\ 1 & -\gamma^3 & \gamma^6 & \gamma & -\gamma^4 & \gamma^7 & \gamma^2 & -\gamma^5 \\ 1 & -\gamma^5 & -\gamma^2 & \gamma^7 & \gamma^4 & \gamma & -\gamma^6 & -\gamma^3 \\ 1 & -\gamma^7 & -\gamma^6 & -\gamma^5 & -\gamma^4 & -\gamma^3 & -\gamma^2 & -\gamma \end{pmatrix}$$

Adding the rows i and $i + 4$ for $i \in \{0, 1, 2, 3\}$, we obtain $W_n^{(+)}$ as shown below which corresponds to the NTT matrix for $n = 4$ scaled by $\xi = 2$:

$$W_n^{(+)} = \begin{pmatrix} 2 & 0 & 2\gamma^2 & 0 & 2\gamma^4 & 0 & 2\gamma^6 & 0 \\ 2 & 0 & 2\gamma^6 & 0 & -2\gamma^4 & 0 & 2\gamma^2 & 0 \\ 2 & 0 & -2\gamma^2 & 0 & 2\gamma^4 & 0 & -2\gamma^6 & 0 \\ 2 & 0 & -2\gamma^6 & 0 & -2\gamma^4 & 0 & -2\gamma^2 & 0 \end{pmatrix}, \quad 2\,W_{n/2} = \begin{pmatrix} 2 & 2\gamma^2 & 2\gamma^4 & 2\gamma^6 \\ 2 & 2\gamma^6 & -2\gamma^4 & 2\gamma^2 \\ 2 & -2\gamma^2 & 2\gamma^4 & -2\gamma^6 \\ 2 & -2\gamma^6 & -2\gamma^4 & -2\gamma^2 \end{pmatrix}.$$

Using this halving property, we can split our cold boot NTT decoding problem into two smaller cold boot NTT decoding problems. Recall that our cold boot instance is described by the equation $\tilde{s} = \mathsf{NTT}_n^{-1}(\Delta) + s$ (see Equation (2)). To show how we utilise Equations (3) and (4), we perform the following steps:

1. Take a forward NTT to obtain the instance $\mathsf{NTT}_n(\tilde{s}) = \mathsf{NTT}_n(s) + \Delta$.
2. Perform the two folding steps:
   (a) (Positive Fold) Compute the vector described by
   $$\mathsf{NTT}_n(\tilde{s})_i + \mathsf{NTT}_n(\tilde{s})_{i+n/2} = 2\mathsf{NTT}_{n/2}(s^{(e)})_i + (\Delta_i + \Delta_{i+n/2}).$$
   (b) (Negative Fold) Compute the vector described by
   $$\frac{1}{2\gamma\omega^i}\left(\mathsf{NTT}_n(\tilde{s})_i - \mathsf{NTT}_n(\tilde{s})_{i+n/2}\right) = \mathsf{NTT}_{n/2}(s^{(o)})_i + \frac{1}{2\gamma\omega^i}(\Delta_i - \Delta_{i+n/2}).$$
3. Define $\Delta_{(l)} := (\Delta_0, \ldots, \Delta_{n/2-1})$, $\Delta_{(r)} := (\Delta_{n/2}, \ldots, \Delta_{n-1})$ and do the following:
   (a) (Positive Fold): Multiply by $2^{-1} \bmod q$ and take an inverse NTT. The resulting instance is
   $$\tilde{s}^{(e)} = 2^{-1}\,\mathsf{NTT}_{n/2}^{-1}(\Delta_{(l)} + \Delta_{(r)}) + s^{(e)}.$$
   (b) (Negative Fold) Define the matrix Ω such that $\Omega_{i,j} = (\gamma\omega^i)^{-1}\delta_{i,j}$ where $\delta_{i,j}$ is the Kronecker delta function. Take an inverse NTT to obtain the instance
   $$\tilde{s}^{(o)} = 2^{-1}\,\mathsf{NTT}_{n/2}^{-1}\left(\Omega \cdot (\Delta_{(l)} - \Delta_{(r)})\right) + s^{(o)}.$$

To summarise, in matrix notation, we can halve the dimension of the instance $\tilde{s} = W_n\Delta + s$ by performing the folding step and deriving the following two instances of half the dimension:

$$\tilde{s}^{(e)} = 2^{-1}W_{n/2}(\Delta_{(l)} + \Delta_{(r)}) + s^{(e)}, \tag{5}$$
$$\tilde{s}^{(o)} = 2^{-1}\left(W_{n/2}\ \Omega\right)(\Delta_{(l)} - \Delta_{(r)}) + s^{(o)}. \tag{6}$$

Looking at the form of the sub-instance given by the "positive fold" (Equation (5)), it is clear that we can run a further divide and conquer step to reduce the dimension further.

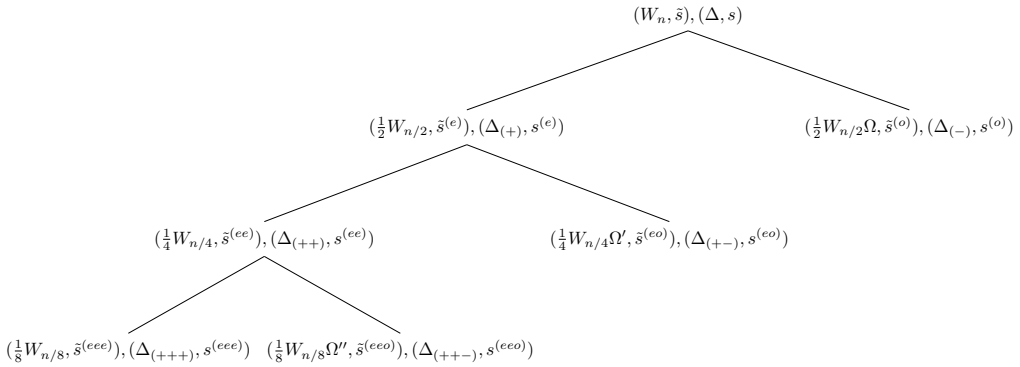

Figure 1: Recursive folding/dimension reduction.

In fact, we can repeatedly divide and conquer the positive fold to reach any dimension we wish as illustrated in Figure 1.

Considering, the "negative fold", the additional scaling factor Ω prevents us from folding down further without rescaling the rows (and thus s). However, we note that on the lowest level, the attacker may still solve the negative branch.

Remark 1. Note that we can also attempt to divide and conquer on the inverse NTT directly in the hope of obtaining sub-instances with error terms of the form $s_{(l)} \pm s_{(r)}$ and secrets $\Delta^{(e)}$ or $\Delta^{(o)}$. Yet, when attempting to do this for the *negacyclic* NTT, we actually obtain sub-instances with errors of the form $s_{(l)} + \omega^{\pm n/4} s_{(r)}$ which are not guaranteed to be small. However, these instances are still susceptible to lattice attacks for limited folding levels.

A drawback of reducing to an extremely small dimension is that the secret becomes less sparse at each level, eventually to the point that its distribution approaches the uniform distribution. Nonetheless, performing only a limited number of folding steps can preserve sparsity. This is because if $\Delta := (\Delta_{(l)}, \Delta_{(r)})$ is *very* sparse, then $\Delta_{(l)} \pm \Delta_{(r)}$ is still expected to be sparse (albeit not as sparse as Δ) and of the same Hamming weight as Δ when written in minimal BSDR. We will see later that a sparse minimal BSDR is the key to our lattice-based attack, so reducing to trivial dimension would be detrimental to our cold boot attack.

5.1 Extending a solution

We now show how to derive a solution to an n-dimensional instance given an oracle that solves just one of the child instances in dimension $n/2$. We instantiate such an oracle in Sections 6 and 7.

For this we note that given the solution to one of the sub-instances, we can derive a solution to the other. First assume that the minimal BSDR (or possibly elements of a minimal BSDR list) of $\Delta_{(l)} + \Delta_{(r)}$ has Hamming weight equal to that of Δ. In other words, there was no decrease in minimal BSDR Hamming weight when performing the positive fold. Then each bit set in the minimal BSDR of $\Delta_{(l)} + \Delta_{(r)}$ (or the single correct element of the BSDR list) originate from either $\Delta_{(l)}$ or $\Delta_{(r)}$. Therefore, in order to guess $\Delta_{(l)} - \Delta_{(r)}$, we simply flip some bits in the minimal BSDR of $\Delta_{(l)} + \Delta_{(r)}$. We then check the correctness of the guess by substituting the value of $\Delta_{(l)} - \Delta_{(r)}$ back into the instance. Note that the list of minimal BSDRs is expected to be relatively short. For example, of the integers $\{1, \ldots, 7680\}$, less than 4.92% have a BSDR list length of 4 or more when considering 13-bit representations. The maximum BSDR list length observed for these

integers is 21 and occurs just 4 times. Since we will typically be encountering integers with low Hamming weight minimal BSDR, the length of the minimal BSDR lists ought to be shorter than suggested by these figures over $\{1, \ldots, 7680\}$.

If the Hamming weight of the minimal BSDR of $\Delta_{(l)} + \Delta_{(r)}$ is different to that of Δ, it has decreased with very high probability.[5] For example, assume that each component of $\Delta_{(l)}$ is the result of at most a single bit-flip. Assume the same for $\Delta_{(r)}$. Performing a fold in such a case would mean that each component of $\Delta_{(l)} + \Delta_{(r)}$ is the result of at most two bit-flips. Therefore the minimal BSDR of *each component* should have Hamming weight *at most* 2. In what follows, a sum $a + b$ is intended to represent folding where a is from $\Delta_{(l)}$ and b is from $\Delta_{(r)}$. Under this assumption, there are two cases where the Hamming weight decreases by 1:

(a) Two bits with the same sign and position collide after folding e.g., $(1 + 1) = 2, (-1 - 1) = -2$

(b) Two bits with opposite signs appear in consecutive positions after folding e.g., $(-1 + 2) = 1, (2 - 1) = 1, (1 - 2) = -1, (-2 + 1) = -1$.

The Hamming weight can also decrease by 2 if two bits with opposite signs collide e.g., $(1 - 1) = 0, (-1 + 1) = 0$.

In light of these observations, we can still use a combinatorial approach to derive $\Delta_{(l)} - \Delta_{(r)}$ from $\Delta_{(l)} + \Delta_{(r)}$ even when folding caused the Hamming weight to decrease. For now, assume that the Hamming weight κ of Δ is known[6] and let κ' denote the Hamming weight of $\Delta_{(r)} + \Delta_{(l)}$ and ignore the small factors arising from the non-uniqueness of the minimal BSDR. We perform one of the following three guessing strategies depending on $\kappa - \kappa'$:

0: Flip signs of $\Delta_{(l)} + \Delta_{(r)}$ to guess $\Delta_{(l)} - \Delta_{(r)}$; $2^{\kappa'}$ guesses required.

1: Assume the Hamming weight decreased by 1 when folding due to either case (a) or (b) above; $3\kappa' \cdot 2^{\kappa'-1}$ guesses required.

2: Assume the Hamming weight decreased by 2 due to a single collision in bits with opposing signs; at most $(n/2 \cdot \lceil \log(q) \rceil - \kappa) \cdot 2 \cdot 2^{\kappa}$ guesses required

Note that the $3\kappa'$ factor arises because we must choose one out of the κ' bits that directly resulted from the Hamming weight decrease, and then there are at most three ways that this spurious bit occurred. For example, suppose the spurious bit represented the integer 2. Then it could be that this value arose from the $(1 + 1), (4 - 2)$ or $(-2 + 4)$. The $(n/2 \cdot \lceil \log(q) \rceil - \kappa)$ factor arises in the third case because we must choose a 0 bit that arose from a collision and there are at most $(n/2 \cdot \lceil \log(q) \rceil - \kappa)$ zeros that are set to 0. There is a chance that this guessing approach fails. In order to increase the probability of success, we would have to perform additional guessing phases where we try to correct multiple spurious bits assuming various configurations. However, our experimental results below show that performing the three guessing phases above already yields a good probability of success. We also note that in a cold boot attack the exact value of κ is not known. In this case, the attacker starts by assuming $\kappa = \kappa'$, followed by $\kappa = \kappa' + 1$ and $\kappa = \kappa' + 2$. We note that this is sufficient to achieve a high rate of success.

Furthermore, an attacker may also directly solve the problem of the neighbour branch $\Delta_{(l)} - \Delta_{(r)}$. Indeed, given $\Delta_{(l)} + \Delta_{(r)}$, we can eliminate either $\Delta_{(l)}$ or $\Delta_{(r)}$ from the neighbour instance to obtain a problem in either $\Delta_{(l)}$ or $\Delta_{(r)}$. This new problem will have associated Hamming weight roughly $\kappa/2$. Furthermore, since $\kappa < n$ there is a very

[5] Modular reduction by q may increase the Hamming weight, but this case occurs so infrequently that we ignore it here.

[6] While this does not hold in a real cold boot attack, we will discuss below how to handle this.

high probability that a known value $(\Delta_{(l)})_i + (\Delta_{(r)})_i = 0$ is indeed the result of adding $(\Delta_{(l)})_i = 0$ and $(\Delta_{(r)})_i = 0$. Thus, the dimension of the neighbour instance can be further reduced by eliminating those components, producing a rather easy instance.

Combining the solutions from the two neighbour instances yields a solution for the parent instance. Thus, a solution in dimension n, implies a solution in dimension $2n$ which can then be extended to solutions in $4n, 8n, \ldots$ using the simple guessing approach above. The overall divide and conquer strategy can be summarised as follows:

1. Repeatedly divide and conquer the positive fold until a desired target dimension n' has been reached.
2. Solve the bottom (positive fold) instance, cf. Section 7.
3. (a) Given a solution to the positive fold, guess the solution to the negative fold and work the solution upwards. This costs in the order of[7]

$$\max\left(2^{\kappa}, \quad 3(\kappa-1)2^{\kappa-2}, \quad (n'/2 \cdot \lceil \log(q) \rceil - (\kappa-2)) \cdot 2 \cdot 2^{\kappa-2}\right)$$

   operations multiplied by the number of folds.

   (b) If guessing fails, solve the negative instance directly, using partial information about $\Delta_{(l)}$ or $\Delta_{(r)}$.
4. Repeat the previous step until the full solution is recovered.

Table 3 uses Kyber parameters with $\kappa = 19$ bits flipped to give an overview of how the Hamming weight of Δ evolves as we fold multiple times. Assuming two folds, this shows a rough success rate of 74% when only considering the trivial phase of guessing to work a 64-dimensional solution upwards. However, when all three phases of guessing are used, we empirically estimate that the success probability is around 97% when working a solution up from dimension 64. The corresponding success probability with $\kappa = 25$ is 94%. These values were obtained by sampling 1,000 random vectors Δ with minimal BSDR of Hamming weight $\kappa = 19$ and 25 and then analysing the cause of a decrease in Hamming weight whenever this occurred. A breakdown of the statistics of 1000 trials at the 128 to 64-dimensional fold are shown in Tables 4 and 5. In particular, we include how many times the Hamming weight decreases by 0,1 and 2 as well as how many of these are solvable in the three simple guessing phases described above. We also report success rates of 98% and 96% for solving this particular fold for $\kappa = 19$ and $\kappa = 25$ respectively. We reiterate that even when the simple guess-and-verify algorithm presented here fails, we expect to be able to solve the neighbour branch by making use of partial information about Δ_l or Δ_r. Thus, from now on, we will assume that the aspect of our attack introduced in this section always succeeds.

Table 3: The preservation rate of the Hamming weight of Δ on folding multiple times for $\kappa = 19$ cold boot flips on Kyber parameters.

Folds	Bottom level dimension	Hamming weight preserved
1	128	90.4%
2	64	73.5%
3	32	48.3%
4	16	19.1%

What remains to be established is how to solve one or both of the bottom level instances; this is the subject of the following sections.

[7] Once again, we ignore the small factor arising from the non-uniqueness of the minimal BSDR.

Table 4: A breakdown of the statistics on the 128 to 64 dimensional fold on 1000 Kyber cold boot instances ($\kappa = 19$) when carrying out the three guessing phases. The "Solvable" row indicates how many of the instances in each category are solvable by the three guessing phases.

	No decrease	Decrease by 1	Decrease by 2
Frequency	824	119	45
Solvable	824	119	39
Success rate	100%	100%	87%

Overall success rate for fold: 98.2%

Table 5: The analogous statistics to those in Table 4 for $\kappa = 25$. For details on the table entries, see the caption for Table 4.

	No decrease	Decrease by 1	Decrease by 2
Frequency	714	174	94
Solvable	714	173	70
Success rate	100%	99%	74%

Overall success rate for fold: 95.7%

6 Lattice formulation

Our algorithm for solving the bottom level instance after applying repeated folding is inspired by the normal form of the primal attack on LWE. At a high level, the aim of this attack is to construct a lattice Λ which contains a vector v closest to $(0, \tilde{s})$, such that the offset between Λ and $(0, \tilde{s})$ is (Δ, s). Then, finding this unique closest vector v to $(0, \tilde{s})$ allows to recover (Δ, s). The success of this attack depends on v being the unique closest vector. Heuristically, we can expect the attack to work if (Δ, s) is shorter than the shortest vector in Λ.[8] Looking at our instance in Equation (2), our "secret term" (interpreting the instance as LWE) is the vector Δ, which is not guaranteed to have small norm, but is guaranteed to be sparse. Note that we abuse notation slightly here and let Equation (2) refer to the bottom level instance after folding, i.e. $\xi > 1$ and Δ is a vector obtained by repeated folding. As mentioned in the introduction, this setting is similar to that considered in [BCGN17, dBDJdW18]. Now, since we know that the component-wise minimal BSDR of Δ will be small in norm, the idea is to construct a lattice resembling the primal attack lattice with an offset vector containing the minimal BSDR of Δ in its components.

In fact, we will generalise this idea to construct a lattice with the 2^ℓ-ary signed digit representation of Δ as an offset. Let $b = \lceil \log_{2^\ell} q \rceil$ and $\Delta^{(\ell)} \in \mathbb{Z}^{nb}$ be the vector where all components of Δ are expanded in the 2^ℓ-ary signed digit representation of minimal *norm*, i.e. we consider 2^ℓ-SDR. Concretely, for Kyber the reader may assume $\ell = 7$ and thus $b = 2$. Now, let $W^{(\ell)} = W \otimes (1, 2^\ell, \ldots, 2^{(b-1)\ell}) \in \mathbb{Z}^{n \times nb}$ and $\theta \in \mathbb{Q}$ be some rational scaling factor. We take as our lattice

$$\Lambda := \{x \in \mathbb{Z}^{nb} \times \mathbb{Q}^n : \left[W^{(\ell)} | (1/\theta) I_n\right] \cdot x \equiv 0 \bmod q\}. \tag{7}$$

Concretely, a basis for this $(nb + n)$-dimensional lattice can be constructed from the rows of

$$B = \begin{pmatrix} I_{bn \times bn} & \theta \, (W^{(\ell)^T}) \\ 0 & q \, \theta \, I_{n \times n} \end{pmatrix}$$

[8] The attack might still succeed even if Λ contains shorter vectors if these vectors are fairly orthogonal to the offset vector (Δ, s).

where $(\cdot)^T$ denotes a transpose. Our aim is that $v := (0, \theta\tilde{s}) - (\Delta^{(\ell)}, \theta s) \in \Lambda$ is the closest lattice vector to $(0, \theta\tilde{s})$. To estimate whether this is the case, we need to estimate the norm of the offset vector $\|(\Delta^{(\ell)}, \theta s)\|$ and the length of the shortest vector in Λ denoted by $\lambda_1(\Lambda)$.

For LWE, $\lambda_1(\Lambda)$ is estimated using the Gaussian heuristic. This is well justified for the LWE case where A is a uniformly random matrix mod q. However, the tensor product in $W^{(\ell)}$ means that there are two classes of unusually short vectors in Λ. The first class contains vectors of the form

$$(0, \ldots, 0, 2^\ell, -1, 0, \ldots, 0)$$

where the last n components are 0 and the 2^ℓ and 1 belong to the same chunk of b entries. This vector essentially "undoes" the tensor product, producing zero in the part corresponding to $W^{(\ell)}$. This vector has norm $\approx 2^\ell$, e.g. 128 in our Kyber-based running example.

The second class of fairly short vectors is given in terms of the 2^ℓ-ary signed digit representation of q that has minimum norm, which we denote as $\vec{q}^{(\ell)} \in \mathbb{Z}^b$. Explicitly, the second class of vectors are of the form

$$(0, \ldots, 0, \vec{q}^{(\ell)}, 0, \ldots, 0)$$

where b divides the number of leading zeros and the last n components are 0. For example, for $\ell = 7$, we can write $q = 7681$ as $60 \cdot 2^{128} + 1$ implying our lattice contains vectors $(0, \ldots, 0, 60, 1, 0, \ldots, 0)$ of norm ≈ 60.

In addition to these short vectors, we must consider the expected length of the shortest vector in Λ ignoring such unusually short vectors. We will denote this length as $\lambda_1'(\Lambda)$. As mentioned above, if W were uniformly random, we could follow the usual strategy and consider the Gaussian heuristic to estimate this norm as:

$$\lambda_1'(\Lambda) \coloneqq \sqrt{\frac{n+nb}{2\pi e}} (\theta q)^{n/(n+nb)}.$$

However, as we will discuss in Section 7 the Gaussian Heuristic does not hold in our case. Thus, we will establish $\lambda_1'(\Lambda)$ empirically using strong lattice reduction.

Now, we expect that the unique vector $v \in \Lambda$ closest to $(0, \theta\,\tilde{s})$ satisfies $v + (\Delta^{(\ell)}, \theta s) = (0, \theta\,\tilde{s})$ when the following three conditions are all met:

$$\|(\Delta^{(\ell)}, \theta s)\| < \begin{cases} \sqrt{2^{2\ell}+1} \approx 2^\ell \\ \|\vec{q}^{(\ell)}\| \\ \lambda_1'(\Lambda). \end{cases}$$

We note that the above conditions imply that we expect that a unique closest vector to our target exists. It does not, by itself, imply that it is efficient to recover it.

Furthermore, we need to estimate the expected length of the vector $(\Delta^{(\ell)}, \theta s)$. Assuming $\kappa \ll n$ bit-flips and $(\rho_0 + \rho_1) \cdot \log_2 q \ll 1$ (so that each non-zero component of Δ is with high probability the result of a single bit-flip), we have that $\|\Delta^{(\ell)}\|^2 \approx \kappa \frac{4^\ell - 1}{3\,\ell}$, cf. Proposition 1. We then expect that

$$\mathbb{E}\left[\|(\Delta^{(\ell)}, \theta s)\|^2\right] = \kappa \frac{4^\ell - 1}{3\,\ell} + n\,\theta^2\,\sigma^2 \tag{8}$$

where σ is the standard deviation of the secret distribution.

Example 2. To carry out the analysis for Kyber, we pick $\ell = 7$ which means $\|\vec{q}^{(\ell)}\|^2 = 3601$ and $\lceil \log_{2^\ell}(q) \rceil = 2$. Thus, we heuristically require our offset vector to have squared norm $< \min(16385, 3601)$. Even picking a very small θ, i.e. ignoring the third condition above, this implies that we can only satisfy our constraints for $\kappa \leq 15$.

6.1 A guessing strategy

To shorten the distance between the lattice and our target vector we simply guess the bits of Δ that contribute most significantly to the norm of $\Delta^{(\ell)}$.[9] To formalise the former approach, we define a "band size" β that describes which bits we consider as contributing significantly to $\Delta^{(\ell)}$. For example, suppose we choose some $\ell \geq 2$ and a band size of $\beta < \ell$. Then we consider the top β bits of each entry in $\Delta^{(\ell)}$ (written in minimum Hamming weight BSDR) as being significant.

We can decompose $\Delta^{(\ell)}$ into two parts: $\Delta^{(\ell,\uparrow)}$ (the vector arising from the bits in the significant band) and $\Delta^{(\ell,\downarrow)}$ (the vector arising from the non-significant band). In doing so, we can write $\Delta^{(\ell)} = \Delta^{(\ell,\uparrow)} + \Delta^{(\ell,\downarrow)}$.

Our "guessing approach" is simply to guess $\Delta^{(\ell,\uparrow)}$ and use the basic primal attack to find the short vector $\Delta^{(\ell,\downarrow)}$. Note that assuming sparsity, the norm of $\Delta^{(\ell,\downarrow)}$ is smaller than that of $\Delta^{(\ell)}$ so it is more likely that the primal attack will succeed. More concretely, once we have guessed $\Delta^{(\ell,\uparrow)}$, we define $\tilde{s}^{(\downarrow)} := \tilde{s} - W^{(\ell)}\Delta^{(\ell,\uparrow)}$ and target offset vector $(\Delta^{(\ell,\downarrow)}, \theta s)$.

Now to investigate when $(\Delta^{(\ell,\downarrow)}, \theta s)$ is likely to be the offset to the unique closest vector in Λ, we begin by assuming some fixed ℓ and $\beta < \ell$ and calculating the expected length of $\Delta^{(\ell,\downarrow)}$. For every individual entry of $\Delta^{(\ell)}$, there are $\ell - \beta$ bits in the non-significant band and β bits in the significant band. Therefore, assuming κ bit-flips in total, we would expect roughly $\frac{\ell-\beta}{\ell}\kappa$ bit-flips[10] in $\Delta^{(\ell,\downarrow)}$. Assuming $\kappa \ll n$ (i.e. sparsity of bit-flips), we expect that

$$\mathbb{E}\left[\|(\Delta^{(\ell,\downarrow)}, \theta s)\|^2\right] = \frac{\ell-\beta}{\ell}\,\kappa\,\frac{4^{\ell-\beta-1}-1}{3\,(\ell-\beta-1)} + n\,\theta^2\,\sigma^2. \tag{9}$$

At this point, we can reuse the three success conditions detailed above, as the characteristic properties of Λ remain unchanged. We refer to the process of removing the top-most bits of a vector as "shaving". This process is parametrised by a band size β and a maximum number of bits to correct, α. Setting α to be less than the expected number of bits set in the top band has the advantage of yielding a shorter number of potential guesses available, but there is also the disadvantage that there may still be a few bits set in the top band. If there are some bits still set in the top band, then the candidate vector $\Delta^{(\ell,\downarrow)}$ may still be too long. The number of possible guesses for the top band with at most α bits flipped is

$$\sum_{i=0}^{\alpha} 2^i \cdot \binom{\#\text{bits in significant band}}{i} \tag{10}$$

where the factor 2^i takes care of the fact that each set bit-flip takes values in $\{-1, 1\}$ when multiple folding steps have been performed. If we have not folded, the factor of 2^i may be omitted since the sign of the bit-flips are known.

Example 3. Returning to our running example, we analyse the case $\ell = 7$ again. Firstly, there are $256 \cdot 2\,\beta$ bits in the significant band. Note the factor of 2 due to the fact that each element of $\mathbb{Z}_{7681}$ requires two integers when written in base 2^7. However, since $7681 < 2^{13}$, the top most bit of each element of $\mathbb{Z}_{7681}$ must be 0. This leaves $256 \cdot (2\beta - 1)$ unknown bit positions where we must correct bit-flips. There is an average of $\frac{2\beta-1}{13} \cdot \kappa$ bit-flips in the unknown part of the significant band. The maximum κ such that (9) < 3601 with θ arbitrarily small, i.e. we are ignoring the second summand in (9), is given in Table 6. We use Equation (10) with α set to the expected number of bit-flips to estimate the number of guesses required for $\kappa = 19$ bit-flips in total.

[9] Of course, other guessing strategies are possible. For example, for sufficiently small κ we may have Δ sparse even mod q. An attacker might thus attempt to guess which columns of Δ can be ignored in an attack.

[10] The number of expected bit-flips is actually less than this for some parameters (see Example 3 below).

Table 6: The maximum possible κ handled by each guessing band size β for Kyber parameters and the cost of guessing the significant band.

		guessing cost for $\kappa = 19$	
β	max κ	$n = 16$	$n = 32$
0	15		
1	52	$2^{9.0}$	$2^{11.0}$
2	169	$2^{25.8}$	$2^{30.9}$

Even strategy. As illustrated in Example 3, the existence of vectors $\vec{q}^{(\ell)}$ is a main limiting factor for ensuring that our offset vector is unusually short. To remove this class of vectors from our lattice, we focus on resolving bit-flips in the least significant bits of the components of Δ. Assume for the moment that this has been achieved, and $\Delta_i \bmod 2 \equiv 0$ for all $0 \leq i < n$. Then, instead of considering $W \otimes (1, 2^\ell, \ldots, 2^{(b-1)\ell}) \in \mathbb{Z}^{n \times nb}$ we may consider

$$W \otimes (2, 2^\ell, \ldots, 2^{(b-1)\ell}) \in \mathbb{Z}^{n \times nb},$$

i.e. scale the rows $0, b, 2b, \ldots, nb$ of B by a factor of two. Since $q \bmod 2 \equiv 1$ we cannot write q as a linear combination of $2, 2^\ell, \ldots, 2^{(b-1)\ell}$. This removes the annoying vectors $\vec{q}^{(\ell)}$ from our lattice. To ensure $\Delta_i \bmod 2 \equiv 0$, as assumed here, we may apply a similar guessing strategy as discussed above. However, we note that this comes with some additional cost for guessing and correcting the least significant bits of the components of Δ.

Finally, we stress that our analysis so far uses expected values throughout. In Figure 2, we plot an example histogram of the $\left\|(\Delta^{(\ell)}, \theta s)\right\|^2$ against our expectation for $\kappa = 19$ and $\theta = 3$. As illustrated in Figure 2, the actually observed distribution has a large variance. Thus, to estimate the cost of our attack, we will derive parameters from empirical evidence.

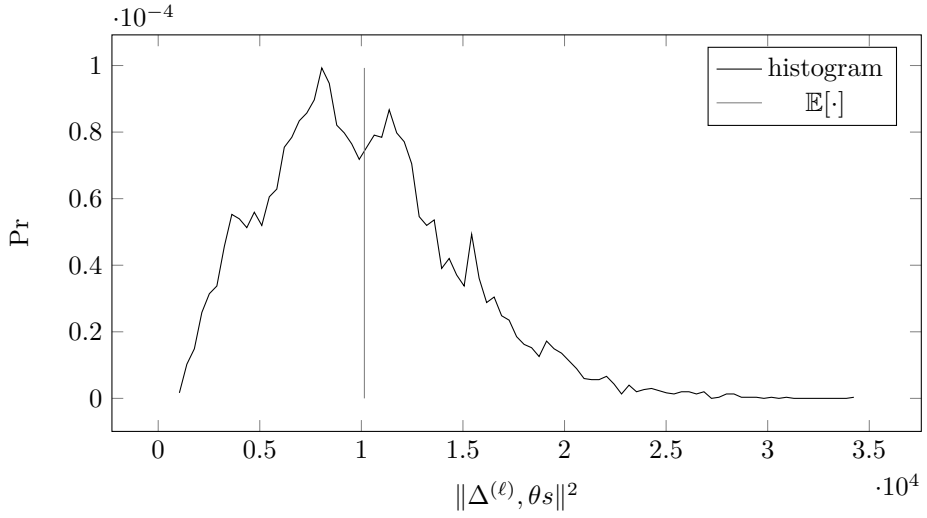

Figure 2: Histogram of observed squared norms of vectors of length $n = 256$ mod $q = 7681$, folded three times to dimension 32, written in base 2^7, for $\theta = 3$ and $\kappa = 19$. Note that in this example $\mathbb{E}[\left\|(\Delta^{(\ell)}, \theta s)\right\|^2] < \kappa \frac{4^\ell - 1}{3^\ell} + n\,(\theta\sigma)^2$ because $\log_2 q < 14$. Thus, half of our entries are bounded by 2^6 instead of 2^7 This is taken into account when we compute the expectation in this figure.

7 BDD on NTT lattices

So far, we have only analysed the existence of a unique closest vector to our target. The last ingredient of our attack is to *find* this vector, i.e. a vector in

$$\Lambda := \{x \in \mathbb{Z}^{n\lceil \log_{2^\ell} q\rceil} \times \mathbb{Q}^n : \left[W^{(\ell)}(1/\theta)I_n\right] \cdot x \equiv 0 \bmod q\}$$

that is close to $(0, \theta\tilde{s})$. Concretely, for Kyber we set $\ell = 7$ and $n = 32$, where $n \geq 16$ is chosen to preserve sparsity for $\rho_0 = 1.0\%, \rho_1 = 0.1\%$, where we expect $\kappa = 19$ bit-flips. To consider the geometry of the lattice spanned by our instances, consider the smaller case $n = 4$, $\theta = 1$ (since it fits on this page). We obtain the q-ary lattice basis

$$B = \begin{pmatrix}
1 & & & & & & & & 1 & -\omega^3 & -\omega^2 & -\omega \\
& 1 & & & & & & & 2^7 & -2^7\omega^3 & -2^7\omega^2 & -2^7\omega \\
& & 1 & & & & & & 1 & -\omega & \omega^2 & -\omega^3 \\
& & & 1 & & & & & 2^7 & -2^7\omega & 2^7\omega^2 & -2^7\omega^3 \\
& & & & 1 & & & & 1 & \omega^3 & -\omega^2 & \omega \\
& & & & & 1 & & & 2^7 & 2^7\omega^3 & -2^7\omega^2 & 2^7\omega \\
& & & & & & 1 & & 1 & \omega & \omega^2 & \omega^3 \\
& & & & & & & 1 & 2^7 & 2^7\omega & 2^7\omega^2 & 2^7\omega^3 \\
& & & & & & & & 7681 & & & \\
& & & & & & & & & 7681 & & \\
& & & & & & & & & & 7681 & \\
& & & & & & & & & & & 7681
\end{pmatrix}.$$

where all of the omitted entries are zero. Note that the lattice spanned by B contains the unusually short vector

$$(1, 0, 1, 0, 1, 0, 1, 0, 4, 0, 0, 0).$$

This vector is not an artefact of the tensor product but an artefact of B being derived from an NTT matrix: it corresponds to folding all the way down to dimension $n = 1$.

More generally, the geometry of the q-ary lattices Λ considered in this work is far from what we would expect from a random q-ary lattice. In Figure 3, we plot the lengths of the Gram-Schmidt vectors of a BKZ-90 reduced basis for a lattice Λ corresponding to folding our 256-dimensional instance down to dimension $n = 32$. This lattice has dimension $96 = \lceil \log_{2^7} q\rceil \cdot n + n$. For comparison, we also plot the expected lengths of the Gram-Schmidt vectors according to the Geometric Series Assumption which approximates the behaviour of random q-ary lattices reasonably well.

Due to this unusual geometry, we cannot readily apply standard estimates for lattice reduction. As a case in point, computing a BKZ-90 reduced basis of the 96-dimensional lattice in Figure 3 took less than an hour with FPLLL [FPL17], i.e. reducing this basis is considerably faster than expected for random q-ary lattices.

Thus, to find the vector $v \in \Lambda$ closest to $(0, \theta\tilde{s})$, we proceed as follows. First, we remove the unusually short vector that corresponds to folding all the way down to $n = 1$. This is accomplished by guessing the value of Δ_0 and considering the sublattice spanned by the rows of Λ except for the first $\lceil \log_{2^\ell} q\rceil$ rows. Pessimistically, we expect that this increases our guessing cost by a factor of $\lceil \log_2 q\rceil$. We refer to this smaller basis as B' and call d the dimension of the lattice spanned by B'. Then, we compute a high-quality basis for the lattice spanned by B'. In particular, for $n = 32$ we compute a BKZ-90 reduced basis. Then, for each guess as in Section 6.1, we perform one pruned BDD enumeration in dimension $\mathsf{bs} = \min(60, d)$, i.e. the bs-dimensional sub-lattice orthogonal to the first $d - \mathsf{bs}$ vectors in B'. We heuristically expect that BDD enumeration in block size bs will find the closest vector iff the projection of the offset vector orthogonal to the first $d - \mathsf{bs}$ vectors in B' is shorter than $\vec{b}^\star_{d-\mathsf{bs}}$, the Gram-Schmidt vector at index $d - \mathsf{bs}$ in B' [ADPS16, AGVW17].

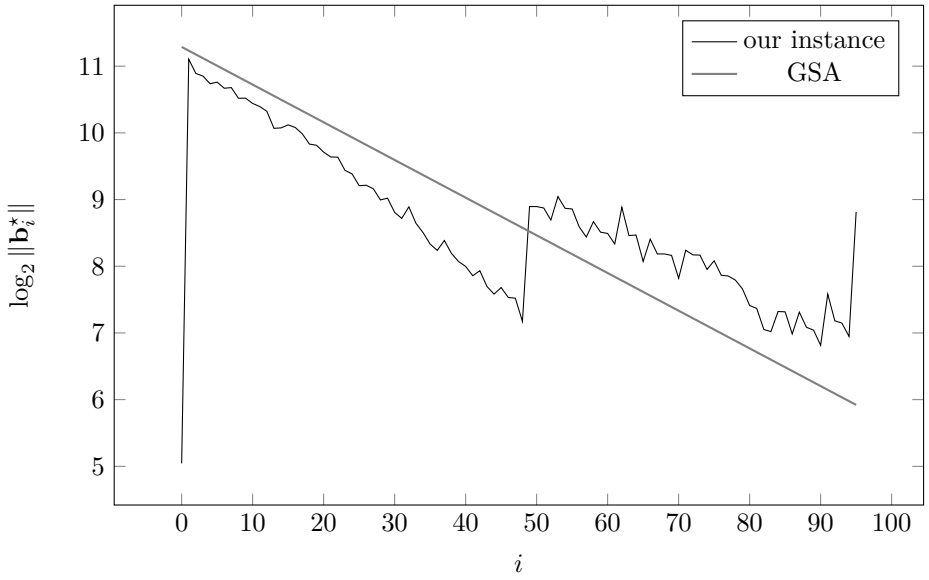

Figure 3: Length of Gram-Schmidt vectors in q-ary lattice derived from negacyclic inverse NTT at dimension 32 using parameters $\ell = 7, \theta = 1$.

As in [ADPS16, AGVW17], we assume that the length of this projection is

$$\sqrt{\frac{\mathsf{bs}}{d}}\,\mathbb{E}\left[\|\Delta^{(\ell)}, \theta s\|\right]$$

where $\mathbb{E}[\|(\Delta^{(\ell)}, \theta s)\|]$ is experimentally established by sampling 1024 vectors.[11] As enumeration radius we pick

$$\min\left(\sqrt{\frac{\mathsf{bs}}{d}} \cdot \mathbb{E}[\|(\Delta^{(\ell)}, \theta s)\|],\ \|\vec{b}^\star_{d-\mathsf{bs}}\|\right). \tag{11}$$

The right-hand argument in (11) takes care of the fact that there is little point in enumerating beyond the length of the shortest vector in the projected sub-lattice if we are targeting a unique closest vector. We illustrate the expected behaviour in Figure 4, where we plot the projected norms for 256 samples of $(\Delta^{(\ell)}, \theta s)$ against the norms of the Gram-Schmidt vectors for our reduced basis B' for $\theta = 3$. Note that in contrast to Figure 3, the basis in Figure 4 is B' and not B. We expect enumeration to succeed for every grey line that stays below the Gram-Schmidt vectors for all indices $< d - \mathsf{bs}$. Figure 4 illustrates that we can improve our probability of success by increasing the enumeration dimension at the cost of increasing the running time. Note that the algorithm may still succeed when the heuristic success condition discussed above is not satisfied due to the orientation of the vectors involved. Therefore, we use the empirical evidence (cf. Tables 7-11) to establish the success rate.

The experiments we performed are as follows. We sampled random sparse binary vectors Δ in dimension n for various κ and construct a corresponding cold boot NTT decoding problem. We then folded this instance down to dimension $n = 32$ and ran the guessing part of the algorithm for some parameters α, β. Since the cost of the guessing part of the attack is easy to predict, we simulated it by always picking the best "shaving" under the constraints imposed by α, β. This is implemented as the `shave` function in

[11] In our experiments, the approximation $\sqrt{\frac{\mathsf{bs}}{d}}\,\mathbb{E}\left[\|\Delta^{(\ell)}, \theta s\|\right]$ indeed appears reasonably accurate.

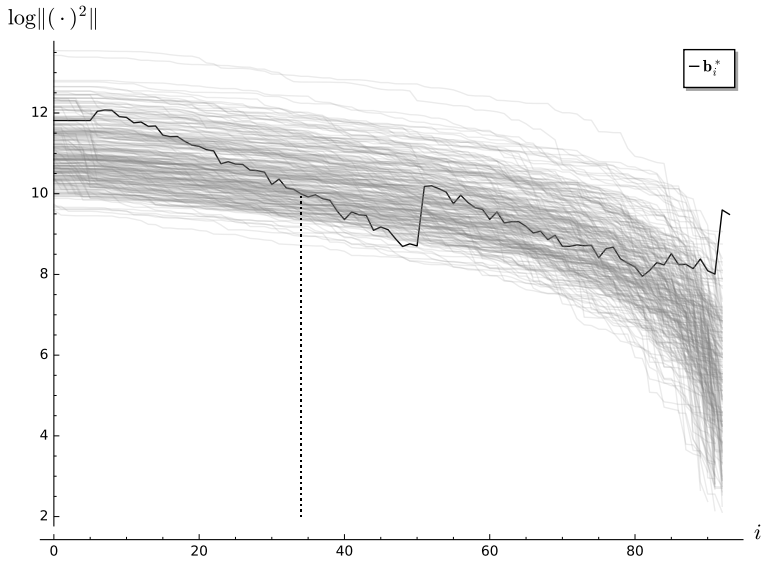

Figure 4: Projected lengths for 256 samples of $(\Delta^{(\ell)}, \theta s)$ and the norms of the Gram-Schmidt vectors for our reduced basis B' for $\theta = 3$ with $\kappa = 19$, folded down three times to $n = 32$ and shaved with parameters $\alpha, \beta = 4, 2$. The dotted line indicates $d - \mathtt{bs}$, i.e. where we start enumerating.

Appendix C. We then ran lattice point enumeration to recover the offset vector, this is implemented in the function `offset_vector`. We report success when the returned vector matches the norm of our target exactly and failure otherwise. In summary, we implemented the full attack on the $n = 32$ sub-problem except for the guessing part. We note that we also implemented and verified extending the solution upwards as described in Section 5.1.

We summarise the observed behaviour of our algorithm for solving the bottom-level $n = 32$ instance in Tables 7-11. These tables illustrate the trade off between the two pruned exhaustive search steps in our algorithm, the first searching for set higher-order bits, the second searching for lattice points. Increasing one reduces the other. Furthermore, according to our empirical evidence, the "even" strategy may provide a small gain in some cases, but it is not overall more efficient than the "not even", i.e. "odd" strategy. All numbers in these tables were obtained using the proof of concept implementation in Sage [S+17] in Appendix C. To establish the cost of the enumeration, we use the number of nodes in the pruned enumeration tree as reported by the `Pruner` class from FPLLL/FPYLLL [FPL17, FPY18]. Processing each node is generally assumed to take about 100 CPU cycles [FPL17].

8 Putting it all together

8.1 Kyber KEM

We now draw together Sections 5-7 to give a concise account of our attack and its performance on the Kyber KEM. Recall that we have 3 instances of the form $\tilde{s} = W_n \Delta + s$ for a single private key in Kyber, with $n = 256$. We first establish some notation. Below, "label, (n, m)" indicates that the instance with "label" in Figure 1 has n variables s_i and that each error term is the sum of m original error terms Δ_j. Note that in Figure 1, the label of a node is given by the subscript in Δ.

Table 7: Experimental results for Kyber parameters and number of bit-flips $\kappa = 5$ ($\rho_0 = 0.2\%$, $\rho_1 = 0.1\%$); θ is the scaling factor of our lattice, α the number of bits we guess in a band of size β. In the "even" case we target the least significant bits of the components of Δ first. The column "guess" holds the number of guesses before lattice enumeration which includes the cost of guessing Δ_0, the column "enum" holds the number of nodes in the pruned lattice-point enumeration tree. The column "total" is the product of the two. All costs are give as $\log_2(\cdot)$. The column "rate" is the success rate over 200 experiments. Only parameters with success rate $\geq 60\%$ are shown. The minimal total cost is highlighted in bold and used in Table 1.

			cost							cost			
θ	α	β	guess	enum	total	rate	θ	α	β	guess	enum	total	rate
$\kappa = 5$, odd							$\kappa = 5$, even						
2	1	1	9.7	16.2	25.9	77.5%	2	1	1	10.7	11.1	21.8	74.5%
2	1	2	11.3	14.8	26.1	85.0%	2	1	2	11.7	9.4	**21.1**	78.5%
2	2	1	14.7	16.2	30.9	83.5%	2	2	1	16.7	10.7	27.4	87.5%
2	2	2	17.9	13.7	31.5	96.5%	2	2	2	18.7	6.0	24.7	94.0%
2	3	1	19.0	16.2	35.3	84.0%	2	3	1	22.1	10.5	32.6	90.5%
2	3	2	23.8	13.5	37.3	99.5%	2	3	2	25.1	5.7	30.8	98.5%
3	1	1	9.7	14.0	23.8	81.0%	3	1	1	10.7	11.6	22.4	79.5%
3	1	2	11.3	13.6	24.9	87.0%	3	1	2	11.7	10.3	22.0	81.0%
3	2	1	14.7	14.0	28.7	87.5%	3	2	1	16.7	11.3	28.0	92.0%
3	2	2	17.9	12.8	30.7	98.5%	3	2	2	18.7	7.8	26.5	96.0%
3	3	1	19.0	14.0	33.1	88.0%	3	3	1	22.1	11.2	33.3	94.5%
3	3	2	23.8	12.7	36.5	100.0%	3	3	2	25.1	7.6	32.7	99.5%

root (256,1) This instance is in the secrets s_i for $i \in \{0, 1, 2, , \ldots, n-1\}$ and has error Δ_j for the j-th equation. It corresponds to the root node in Figure 1.

`+` (128, 2) This instance is the result of folding once on the plus branch. It is in the secrets s_i for $i \in \{0, 2, 4, \ldots, n-2\}$. The j-th equation has error term $\Delta_j + \Delta_{j+128}$.

`++` (64, 4) This instance is the result of folding twice on the plus branch. It is in the secrets s_i for $i \in \{0, 4, 8 \ldots, n-4\}$. The j-th equation has error term $\Delta_j + \Delta_{j+64} + \Delta_{j+128} + \Delta_{j+192}$.

`+++` (32, 8) This instance is the result of folding three times on the plus branch. It is in the secrets s_i for $i \in \{0, 8, 16, \ldots, n-8\}$. The j-th equation has error term $\Delta_j + \Delta_{j+32} + \Delta_{j+64} + \Delta_{j+96} + \Delta_{j+128} + \Delta_{j+160} + \Delta_{j+192} + \Delta_{j+224}$.

`++-` (32, 8) This instance the result of folding twice on the plus branch and once on the negative. It is in the secrets s_i for $i \in \{4, 12, 20, \ldots, n-4\}$. The j-th equation has error term $\Delta_j - \Delta_{j+32} + \Delta_{j+64} - \Delta_{j+96} + \Delta_{j+128} - \Delta_{j+160} + \Delta_{j+192} - \Delta_{j+224}$.

For each of our three sub-problems we perform the following steps:

1. Divide and conquer 3 times to obtain two bottom level instances `+++` and `++-` as in Section 5.
2. Solve at least one bottom level instance using combinatorial and lattice-reduction techniques as in Sections 6 and 7. The cost and expected success rate for solving one such instance are given in Section 7. If solving one instance succeeds with probability p_0, we assume that this step succeeds with probability $1 - (1 - p_0)^2$, i.e. we assume the two bottom level instances are sufficiently different.
3. Substitute the solution obtained into the instance `++`. This reduces it from $(64, 4)$ to $(32, 4)$, Solve this instance as in Sections 6 and 7. Note that solving this instance

Table 8: Experimental results for Kyber parameters and $\kappa = 10$ ($\rho_0 = 0.5\%$, $\rho_1 = 0.1\%$). For details see Table 7.

θ	α	β	cost guess	cost enum	cost total	rate
$\kappa = 10$, odd						
2	3	2	23.8	16.2	40.0	93.0%
2	4	2	29.4	16.2	45.6	97.5%
2	5	2	34.6	15.9	50.5	99.5%
3	3	1	19.0	14.0	**33.1**	63.5%
3	3	2	23.8	14.0	37.9	95.5%
3	4	1	22.9	14.0	37.0	64.0%
3	4	2	29.4	14.0	43.4	98.0%
3	5	1	26.5	14.0	40.5	64.0%
3	5	2	34.6	14.0	48.6	98.5%
4	3	1	19.0	20.9	40.0	64.0%
4	3	2	23.8	19.5	43.4	87.5%
4	4	1	22.9	20.9	43.9	64.5%
4	4	2	29.4	19.1	48.5	91.5%
4	5	1	26.5	20.9	47.4	64.5%
4	5	2	34.6	18.8	53.4	92.0%

θ	α	β	cost guess	cost enum	cost total	rate
$\kappa = 10$, even						
2	3	1	22.1	11.1	33.2	66.5%
2	3	2	25.1	9.8	34.9	89.5%
2	4	1	27.0	11.1	38.1	69.0%
2	4	2	31.1	8.2	39.3	95.0%
2	5	1	31.6	11.1	42.7	70.0%
2	5	2	36.7	7.7	44.4	99.5%
3	3	1	22.1	14.2	36.3	76.5%
3	3	2	25.1	10.6	35.7	90.5%
3	4	1	27.0	14.2	41.3	79.0%
3	4	2	31.1	9.4	40.5	95.0%
3	5	1	31.6	14.2	45.9	79.0%
3	5	2	36.7	9.0	45.8	99.5%
4	3	1	22.1	12.2	34.3	86.0%
4	3	2	25.1	12.1	37.2	95.5%
4	4	1	27.0	12.2	39.3	87.0%
4	4	2	31.1	11.2	42.2	98.0%
4	5	1	31.6	12.2	43.9	87.5%
4	5	2	36.7	10.9	47.6	99.5%

Table 9: Experimental results for Kyber parameters and $\kappa = 19$ ($\rho_0 = 1.0\%$, $\rho_1 = 0.1\%$). For details see Table 7.

θ	α	β	cost guess	cost enum	cost total	rate
$\kappa = 19$, odd						
2	4	2	29.4	16.2	45.6	62.5%
2	5	2	34.6	16.2	50.8	80.0%
2	6	2	39.5	16.2	55.7	86.5%
2	7	2	44.2	16.2	60.4	91.0%
2	8	2	48.7	16.2	64.9	94.5%
3	4	2	29.4	14.0	43.4	71.0%
3	5	2	34.6	14.0	48.6	82.0%
3	6	2	39.5	14.0	53.6	87.0%
3	7	2	44.2	14.0	58.3	91.5%
3	8	2	48.7	14.0	62.8	92.5%
4	4	2	29.4	20.9	50.3	66.5%
4	5	2	34.6	20.9	55.5	70.5%
4	6	2	39.5	20.9	60.5	79.0%
4	7	2	44.2	20.9	65.2	83.5%
4	8	2	48.7	20.9	69.7	84.0%

θ	α	β	cost guess	cost enum	cost total	rate
$\kappa = 19$, even						
2	5	2	36.7	11.1	47.8	70.5%
2	6	2	42.1	11.1	53.1	84.0%
2	7	2	47.2	11.1	58.3	90.5%
2	8	2	52.1	10.6	62.8	96.0%
3	5	2	36.7	14.2	50.9	74.0%
3	6	2	42.1	13.3	55.4	86.0%
3	7	2	47.2	12.2	59.4	93.0%
3	8	2	52.1	11.8	63.9	98.0%
4	4	2	31.1	12.2	**43.3**	70.5%
4	5	2	36.7	12.2	48.9	83.0%
4	6	2	42.1	12.2	54.3	89.0%
4	7	1	40.0	12.2	52.2	60.0%
4	7	2	47.2	12.2	59.4	96.5%
4	8	1	43.8	12.2	56.1	60.5%
4	8	2	52.1	12.2	64.4	97.5%

Table 10: Experimental results for Kyber parameters and $\kappa = 25$ ($\rho_0 = 1.4\%, \rho_1 = 0.1\%$). For details see Table 7.

			cost							cost			
θ	α	β	guess	enum	total	rate	θ	α	β	guess	enum	total	rate
$\kappa = 25$, odd							$\kappa = 25$, even						
2	6	2	39.5	16.2	55.7	60.5%							
2	6	3	44.0	16.2	60.2	68.0%							
2	7	2	44.2	16.2	60.4	72.5%	2	7	2	47.2	11.1	58.3	65.0%
2	7	3	49.5	16.2	65.7	82.5%	2	7	3	51.4	11.1	62.4	67.0%
2	8	2	48.7	16.2	64.9	79.0%	2	8	2	52.1	11.1	63.2	77.5%
2	8	3	54.8	16.2	71.0	90.5%	2	8	3	56.9	11.1	68.0	80.5%
3	5	3	38.3	14.0	**52.4**	60.0%							
3	6	2	39.5	14.0	53.6	70.0%							
3	6	3	44.0	14.0	58.1	74.0%							
3	7	2	44.2	14.0	58.3	76.0%	3	7	2	47.2	14.2	61.4	71.0%
3	7	3	49.5	14.0	63.5	81.5%	3	7	3	51.4	14.2	65.6	71.5%
3	8	2	48.7	14.0	62.8	80.0%	3	8	2	52.1	14.2	66.4	82.5%
3	8	3	54.8	14.0	68.8	87.0%	3	8	3	56.9	13.3	70.2	84.5%
4	6	2	39.5	20.9	60.5	60.5%	4	6	2	42.1	12.2	54.3	66.5%
4	6	3	44.0	20.9	65.0	68.0%	4	6	3	45.6	12.2	57.9	66.0%
4	7	2	44.2	20.9	65.2	69.0%	4	7	2	47.2	12.2	59.4	80.0%
4	7	3	49.5	20.9	70.4	80.5%	4	7	3	51.4	12.2	63.6	79.0%
4	8	2	48.7	20.9	69.7	71.0%	4	8	2	52.1	12.2	64.4	89.5%
4	8	3	54.8	19.5	74.3	84.0%	4	8	3	56.9	12.2	69.1	89.0%

Table 11: Experimental results for Kyber parameters and $\kappa = 30$ ($\rho_0 = 1.7\%, \rho_1 = 0.1\%$). For details see Table 7.

			cost							cost			
θ	α	β	guess	enum	total	rate	θ	α	β	guess	enum	total	rate
$\kappa = 30$, odd							$\kappa = 30$, even						
2	8	3	54.8	16.2	71.0	66.5%							
2	9	2	53.0	16.2	69.2	63.0%	2	9	2	56.9	11.1	67.9	65.5%
2	9	3	59.8	16.2	76.0	82.5%	2	9	3	62.3	11.1	73.3	66.5%
3	7	2	44.2	14.0	58.3	60.0%							
3	7	3	49.5	14.0	63.5	66.0%							
3	8	2	48.7	14.0	**62.8**	67.5%	3	8	2	52.1	14.2	66.4	65.0%
3	8	3	54.8	14.0	68.8	73.0%	3	8	3	56.9	14.2	71.1	65.0%
3	9	2	53.0	14.0	67.1	74.5%	3	9	2	56.9	14.2	71.1	73.5%
3	9	3	59.8	14.0	73.9	84.0%	3	9	3	62.3	14.2	76.5	75.0%
							4	7	2	47.2	12.2	59.4	64.5%
4	7	3	49.5	20.9	70.4	62.5%	4	7	3	51.4	12.2	63.6	64.5%
4	8	2	48.7	20.9	69.7	63.5%	4	8	2	52.1	12.2	64.4	72.0%
4	8	3	54.8	20.9	75.7	73.0%	4	8	3	56.9	12.2	69.1	72.0%
4	9	2	53.0	20.9	74.0	66.0%	4	9	2	56.9	12.2	69.1	79.5%
4	9	3	59.8	20.7	80.5	80.0%	4	9	3	62.3	12.2	74.5	79.0%

is much easier than in the previous step since the Hamming weight of the noise is reduced to $\approx \kappa/2$. We assume this step always succeeds.

4. Work the solution of ++ upward to + by solving +- using the information from ++ as in Section 5.1. This step succeeds with probability p_1 and we assume that it is cheaper than the previous steps.

5. Work the solution of + upward to "root" by solving - using the information from + as in Section 5.1. We assume this step always succeeds and we assume that it is cheaper than the previous steps.

Thus, the overall complexity of recovering 256 components of the Kyber secret is to run the lattice attack from Section 7 three times (steps 2 and 3) and succeeds with probability $\approx p_1 \cdot (1-(1-p_0)^2)$. In particular, for our choice of parameters we have[12] $p_1 \approx 1$ and $p_0 > 0.6$ and thus expect success with probability > 0.84. For example, with $\kappa = 19$, Table 9 shows that we can solve the hardest BDD problem with a cost of $2^{43.3}$ and success probability $p_0 = 0.705$. Since this is by far the most expensive stage of the attack, we report an attack cost of enumerating $\approx 2^{43.3}$ nodes in an enumeration tree where each node requires about 100 CPU cycles to process and a $p_1 \cdot (1-(1-p_0)^2) \approx 0.91$ success probability. We can attack each of the $k = 3$ module elements separately and combine the final solution. We note that the attacker can detect with high probability when a sub-solution is incorrect and thus invest more computational resources to increase the chance of success. We summarise our results in Table 1.

The attack needs to be run $k = 3$ times to recover a full Kyber secret. If a solution cannot be obtained for one of the three secret ring elements, then the solutions of the other two sub-problems can be substituted back into the original MLWE problem for Kyber's public key. This reduces the effective dimension of the public key to $n = 256$. An attacker could then target this smaller RLWE instance. Solving such an instance costs roughly 2^{77} according to the LWE estimator from [APS15], again assuming that enumeration is used to realise the SVP oracle inside BKZ. As suggested above, an attacker could alternatively attempt to re-run our cold boot attack on the remaining unknown secret element with different parameter choices from Tables 7-11. This would boost the probability of success at the expense of a greater computational cost.

8.2 New Hope KEM

We now move away from our MLWE-based example of Kyber KEM and give a concise account of the performance of our attack on the RLWE-based New Hope KEM [PAA+17]. The parameters used are $n = 1024, q = 12289$ and the secret polynomials have coefficients lying in the set $\{0, \pm 1, \ldots, \pm 8\}$. Similarly to Kyber KEM, New Hope uses an NTT to store its secret keys, meaning that we can launch the same cold boot attack. An important distinction between the Kyber and New Hope cases is that, for Kyber, we obtain multiple independent cold boot instances, each one corresponding to an individual polynomial in the secret key; this leads to multiple instances of relatively low dimension for Kyber. However, in the case of New Hope, we have just one cold boot instance in a large dimension. This distinction between MLWE- and RLWE-based schemes holds true in general for our cold boot attack in the NTT domain.

We focus our attention on the lattice aspect of the attack, assuming that we have folded the New Hope 1024-dimensional cold boot instance repeatedly to reach a 32-dimensional instance using the methods in Section 5. We can then experimentally estimate the success rate of solving this bottom level instance for various choices of θ, α, β using the methods

[12] Note that it is easy to amplify p_1 by performing additional guessing phases in Section 5.1.

Table 12: Experimental results for New Hope parameters and number of bit-flips $\kappa = 10$; θ is the scaling factor of our lattice, α the number of bits we guess in a band of size β. In the "even" case we target the least significant bits of the components of Δ first. The column "guess" holds the number of guesses before lattice enumeration which includes the cost of guessing Δ_0, the column "enum" holds the number of nodes in the pruned lattice-point enumeration tree. The column "total" is the product of the two. All costs are give as $\log_2(\cdot)$. The column "rate" is the success rate over 100 experiments. Only parameters with success rate $\geq 50\%$ are shown. The minimal total cost is highlighted in bold and used in Table 2.

			cost			
θ	α	β	guess	enum	total	rate
$\kappa = 10$, odd						
2	3	2	25.1	15.6	40.7	82.0%
2	4	2	31.1	15.6	46.6	96.1%
2	5	2	36.7	15.6	52.3	100.0%
3	3	2	25.1	12.8	37.9	68.0%
3	4	2	31.1	12.8	43.9	76.6%
3	5	2	36.7	12.8	49.5	81.2%

			cost			
θ	α	β	guess	enum	total	rate
$\kappa = 10$, even						
2	3	1	23.8	10.3	**34.2**	60.9%
2	3	2	26.1	10.3	36.4	73.4%
2	4	1	29.4	10.3	39.7	73.4%
2	4	2	32.4	10.3	42.7	93.0%
2	5	1	34.6	10.3	44.9	77.3%
2	5	2	38.3	10.3	48.7	97.7%
3	3	1	23.8	11.1	35.0	63.3%
3	3	2	26.1	11.1	37.2	71.9%
3	4	1	29.4	11.1	40.5	76.6%
3	4	2	32.4	11.1	43.5	91.4%
3	5	1	34.6	11.1	45.7	78.1%
3	5	2	38.3	11.1	49.4	93.0%

in Section 7 with $b = 2$ and $\ell = 7$. The results for $\kappa = 10, 19, 25, 30$ are given in Tables 12–15. Note that the value $\kappa = 19$ roughly corresponds to the limiting cold boot case of $\rho_0 = 0.17\%, \rho_1 = 0.1\%$ where liquid nitrogen is used to cool the RAM chip.

We now reuse the analysis and notation from Section 8.1 to estimate the running time and success probability of the full attack on New Hope. The success probability of the attack is $\approx p_1 \cdot (1 - (1 - p_0)^2)$ where p_1 is the success probability of working a bottom level solution up and p_0 is the probability of successfully solving a bottom level instance. Once again, we assume that this aspect of the attack can be performed successfully with probability $p_1 \approx 1$ without dominating the complexity of the overall attack. To determine p_0, we use the results form Tables 12-15. A summary of our results for $\kappa = 19, 25, 30$ is given in Table 2.

References

[AC11] Martin Albrecht and Carlos Cid. Cold boot key recovery by solving polynomial systems with noise. In Javier Lopez and Gene Tsudik, editors, *ACNS 11: 9th International Conference on Applied Cryptography and Network Security*, volume 6715 of *Lecture Notes in Computer Science*, pages 57–72. Springer, Heidelberg, June 2011.

[ACC+17] David Archer, Lily Chen, Jung Hee Cheon, Ran Gilad-Bachrach, Roger A. Hallman, Zhicong Huang, Xiaoqian Jiang, Ranjit Kumaresan, Bradley A. Malin, Heidi Sofia, Yongsoo Song, and Shuang Wang. Applications of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, July 2017.

[ACFP14] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, and Ludovic Perret. Algebraic algorithms for LWE. Cryptology ePrint Archive, Report 2014/1018, 2014. http://eprint.iacr.org/2014/1018.

Table 13: Experimental results for New Hope parameters and number of bit-flips $\kappa = 19$; for details see Table 12.

			cost			
θ	α	β	guess	enum	total	rate
$\kappa = 19$, odd						
2	5	2	36.7	15.6	52.3	57.8%
2	6	2	42.1	15.6	57.6	76.6%
2	7	2	47.2	15.6	62.8	82.8%
2	8	2	52.1	15.6	67.7	90.6%
3	6	2	42.1	12.8	54.9	51.6%
3	7	2	47.2	12.8	60.0	56.2%
3	8	2	52.1	12.8	64.9	62.5%

			cost			
θ	α	β	guess	enum	total	rate
$\kappa = 19$, even						
2	5	2	38.3	10.3	**48.7**	59.4%
2	6	2	44.0	10.3	54.4	69.5%
2	7	2	49.5	10.3	59.8	84.4%
2	8	2	54.8	10.3	65.1	89.8%
3	5	2	38.3	11.1	49.4	58.6%
3	6	2	44.0	11.1	55.1	75.0%
3	7	2	49.5	11.1	60.6	81.2%
3	8	2	54.8	11.1	65.9	85.9%
4	6	2	44.0	11.7	55.7	60.9%
4	7	2	49.5	11.7	61.2	69.5%
4	8	2	54.8	11.7	66.4	75.0%
5	6	2	44.0	12.7	56.7	50.8%
5	7	2	49.5	12.7	62.2	59.4%
5	8	2	54.8	12.7	67.5	60.2%

Table 14: Experimental results for New Hope parameters and $\kappa = 25$. For details see Table 12.

			cost			
θ	α	β	guess	enum	total	rate
$\kappa = 25$, odd						
2	7	2	47.2	15.6	62.8	52.3%
2	7	3	51.4	15.6	66.9	59.4%
2	8	2	52.1	15.6	67.7	63.3%
2	8	3	56.9	15.6	72.5	75.0%
3	8	2	52.1	12.8	64.9	50.0%
3	8	3	56.9	12.8	69.7	57.8%

			cost			
θ	α	β	guess	enum	total	rate
$\kappa = 25$, even						
2	7	2	49.5	10.3	59.8	53.9%
2	7	3	52.9	10.3	63.3	54.7%
2	8	2	54.8	10.3	65.1	64.1%
2	8	3	58.7	10.3	69.0	64.8%
3	7	2	49.5	11.1	**60.6**	56.2%
3	7	3	52.9	11.1	64.1	56.2%
3	8	2	54.8	11.1	65.9	66.4%
3	8	3	58.7	11.1	69.8	67.2%

Table 15: Experimental results for New Hope parameters and $\kappa = 30$. For details see Table 12.

			cost			
θ	α	β	guess	enum	total	rate
$\kappa = 30$, odd						
2	8	3	56.9	15.6	72.5	52.3%
2	9	2	56.9	15.6	72.4	56.2%
2	9	3	62.3	15.6	77.8	66.4%
2	10	2	61.5	15.6	77.0	64.1%
2	10	3	67.5	15.6	83.0	76.6%
3	9	3	62.3	12.8	75.1	50.0%
3	10	3	67.5	12.8	80.3	54.7%

			cost			
θ	α	β	guess	enum	total	rate
$\kappa = 30$, even						
2	9	2	59.8	10.3	**70.2**	56.2%
2	9	3	64.3	10.3	74.6	58.6%
2	10	2	64.8	10.3	75.1	68.0%
2	10	3	69.7	10.3	80.1	70.3%
3	9	2	59.8	11.1	71.0	55.5%
3	9	3	64.3	11.1	75.4	56.2%
3	10	2	64.8	11.1	75.9	67.2%
3	10	3	69.7	11.1	80.8	68.0%

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16*, pages 327–343. USENIX Association, 2016.

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011: 38th International Colloquium on Automata, Languages and Programming, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, Heidelberg, July 2011.

[AGV09] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 474–495. Springer, Heidelberg, March 2009.

[AGVW17] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the expected cost of solving uSVP and applications to LWE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 297–322. Springer, Heidelberg, December 2017.

[AI06] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 459–468. IEEE, 2006.

[AJPS17] Divesh Aggarwal, Antoine Joux, Anupam Prakash, and Mikos Santha. Mersenne-756839. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[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.

[BAA+17] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Kramer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. qTESLA. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[BCGN17] Marc Beunardeau, Aisling Connolly, Rémi Géraud, and David Naccache. On the hardness of the Mersenne low Hamming ratio assumption. Cryptology ePrint Archive, Report 2017/522, 2017. https://eprint.iacr.org/2017/522.

[BDH+17] Michael Brenner, Wei Dai, Shai Halevi, Kyoohyung Han, Amir Jalali, Miran Kim, Kim Laine, Alex Malozemoff, Pascal Paillier, Yuriy Polyakov, Kurt Rohloff, Erkay Savaş, and Berk Sunar. A standard API for RLWE-based homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, July 2017.

[BG10] Zvika Brakerski and Shafi Goldwasser. Circular and leakage resilient public-key encryption under subgroup indistinguishability - (or: Quadratic residuosity strikes back). In Rabin [Rab10], pages 1–20.

[BL14] Alexandra Berkoff and Feng-Hao Liu. Leakage resilient fully homomorphic encryption. In Yehuda Lindell, editor, *TCC 2014: 11th Theory of Cryptography Conference*, volume 8349 of *Lecture Notes in Computer Science*, pages 515–539. Springer, Heidelberg, February 2014.

[CCD+17] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, July 2017.

[CLP17] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - SEAL v2.1. Cryptology ePrint Archive, Report 2017/224, 2017. http://eprint.iacr.org/2017/224.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, Heidelberg, December 2011.

[dBDJdW18] Koen de Boer, Léo Ducas, Stacey Jeffery, and Ronald de Wolf. Attacks on the AJPS Mersenne-based cryptosystem. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 101–120, Cham, 2018. Springer.

[DGK+10] Yevgeniy Dodis, Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Public-key encryption schemes with auxiliary inputs. In Daniele Micciancio, editor, *TCC 2010: 7th Theory of Cryptography Conference*, volume 5978 of *Lecture Notes in Computer Science*, pages 361–381. Springer, Heidelberg, February 2010.

[DKRV17] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[DSGKS17] Dana Dachman-Soled, Huijing Gong, Mukul Kulkarni, and Aria Shahverdi. On the leakage resilience of ideal-lattice based public key encryption. Cryptology ePrint Archive, Report 2017/1127, 2017. https://eprint.iacr.org/2017/1127.

[DTGW17] Jintai Ding, Tsuyoshi Takagi, Xinwei Gao, and Yuntao Wang. Ding key exchange. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[FPL17] The FPLLL Development Team. FPLLL, a lattice reduction library. Available at https://github.com/fplll/fplll, 2017.

[FPY18] The FPYLLL Development Team. FPYLLL, a Python interface for FPLLL. Available at https://github.com/fplll/fpylll, 2018.

[GMZB+17] Oscar Garcia-Morchon, Zhenfei Zhang, Sauvik Bhattacharya, Ronald Rietman, Ludo Tolhuizen, and Jose-Luis Torre-Arce. Round2. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[Hal09] Shai Halevi, editor. *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*. Springer, Heidelberg, August 2009.

[Ham17] Mike Hamburg. Three Bears. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[HG07] Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 150–169. Springer, Heidelberg, August 2007.

[HMM10] Wilko Henecka, Alexander May, and Alexander Meurer. Correcting errors in RSA private keys. In Rabin [Rab10], pages 351–369.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Halevi [Hal09], pages 1–17.

[HSH+09] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[Kan83] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *15th Annual ACM Symposium on Theory of Computing*, pages 193–206. ACM Press, April 1983.

[KUI99] Takayasu Kaida, Satoshi Uehara, and Kyoki Imamura. An algorithm for the k-error linear complexity of sequences over $GF(p^m)$ with period p^n, p a prime. *Information and Computation*, 151(1-2):134–147, 1999.

[KY10] Abdel Alim Kamal and Amr M Youssef. Applications of SAT solvers to AES key recovery from decayed key schedule images. In *Emerging Security Information Systems and Technologies (SECURWARE), 2010 Fourth International Conference on*, pages 216–220. IEEE, 2010.

[Laa14] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. Cryptology ePrint Archive, Report 2014/744, 2014. http://eprint.iacr.org/2014/744.

[LDK+17] Vadim Lyubashevsky, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[LLJ+17] Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, Jingnan He, and Zhenfei Zhang. LAC. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[LLKN17] Dongxi Liu, Nan Li, Jongkil Kim, and Surya Nepal. Compact LWE. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[LN13] Mingjie Liu and Phong Q. Nguyen. Solving BDD by enumeration: An update. In Ed Dawson, editor, *Topics in Cryptology – CT-RSA 2013*, volume 7779 of *Lecture Notes in Computer Science*, pages 293–309. Springer, Heidelberg, February / March 2013.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, Heidelberg, February 2011.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, Heidelberg, May / June 2010.

[LPR13a] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM*, 60(6):43:1–43:35, November 2013.

[LPR13b] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 35–54. Springer, Heidelberg, May 2013.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, June 2015.

[Mas69] James Massey. Shift-register synthesis and BCH decoding. *IEEE transactions on Information Theory*, 15(1):122–127, 1969.

[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.

[MW15] Daniele Micciancio and Michael Walter. Fast lattice point enumeration with minimal overhead. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 276–294. ACM-SIAM, January 2015.

[NAB+17] Michael Naehrig, Erdem Alkim, Joppe Bos, Leo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[Nat16] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the Post-Quantum Cryptography standardization process. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf, December 2016.

[NS09] Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. In Halevi [Hal09], pages 18–35.

[PAA+17] Thomas Poppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Leo Ducas, Antonio de la Piedra, Peter Schwabe, and Douglas Stebila. NewHope. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[Pei15] Chris Peikert. A decade of lattice cryptography. Cryptology ePrint Archive, Report 2015/939, 2015. http://eprint.iacr.org/2015/939.

[PHAM17] Le Trieu Phong, Takuya Hayashi, Yoshinori Aono, and Shiho Moriai. LOTUS. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[Pie12] Krzysztof Pietrzak. Subspace LWE. In Ronald Cramer, editor, *TCC 2012: 9th Theory of Cryptography Conference*, volume 7194 of *Lecture Notes in Computer Science*, pages 548–563. Springer, Heidelberg, March 2012.

[PPS12] Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 386–403. Springer, Heidelberg, December 2012.

[PS15] Bertram Poettering and Dale L. Sibborn. Cold boot attacks in the discrete logarithm setting. In Kaisa Nyberg, editor, *Topics in Cryptology – CT-RSA 2015*, volume 9048 of *Lecture Notes in Computer Science*, pages 449–465. Springer, Heidelberg, April 2015.

[PV17] Kenneth G. Paterson and Ricardo Villanueva-Polanco. Cold boot attacks on NTRU. In Arpita Patra and Nigel P. Smart, editors, *Progress in Cryptology - INDOCRYPT 2017: 18th International Conference in Cryptology in India*, volume 10698 of *Lecture Notes in Computer Science*, pages 107–125. Springer, Heidelberg, December 2017.

[Rab10] Tal Rabin, editor. *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*. Springer, Heidelberg, August 2010.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93. ACM Press, May 2005.

[S+17] William Stein et al. *Sage Mathematics Software Version 8.1*. The Sage Development Team, 2017. http://www.sagemath.org.

[Saa17] Markku-Juhani O. Saarinen. HILA5. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[SAB+17] Peter Schwabe, Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehle. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[SAL+17] Nigel P. Smart, Martin R. Albrecht, Yehuda Lindell, Emmanuela Orsini, Valery Osheter, Kenny Paterson, and Guy Peer. LIMA. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[Sch87] Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53:201–224, 1987.

[Sch03] Claus-Peter Schnorr. Lattice reduction by random sampling and birthday methods. In Helmut Alt and Michel Habib, editors, *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2607 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2003.

[SM93] Mark Stamp and Clyde F Martin. An algorithm for the k-error linear complexity of binary sequences with period 2^n. *IEEE Transactions on Information Theory*, 39(4):1398–1401, 1993.

[SPL+17] Minhye Seo, Jong Hwan Park, Dong Hoon Lee, Suhri Kim, and Seung-Joon Lee. EMBLEM and R.EMBLEM. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[SS71] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3):281–292, Sep 1971.

[SSZ17] Ron Steinfeld, Amin Sakzad, and Raymond K. Zhao. Titanium. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[Tso09] Alex Tsow. An improved recovery algorithm for decayed AES key schedule images. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *SAC 2009: 16th Annual International Workshop on Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 215–230. Springer, Heidelberg, August 2009.

[Win96] Franz Winkler. *Polynomial Algorithms in Computer Algebra*. Texts & Monographs in Symbolic Computation. Springer, 1996.

[ZJGS17] Yunlei Zhao, Zhengzhong Jin, Boru Gong, and Guangye Sui. KCL (pka OKCN/AKCN/CNKE). Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

A The Blahut, Berlekamp-Massey attack

A.1 Linear complexity

In this section, we will be considering sequences of elements in a field $\mathbb{Z}_q$ where q is prime. Linear feedback shift registers (LFSR) for binary sequences are well known as a concept.

We will be considering LFSRs over a field $\mathbb{Z}_q$ i.e. shift registers where the input (or feedback function) is a linear combination (over $\mathbb{Z}_q$) of the current register values.

Definition 7 (Linear Complexity)**.** The linear complexity of a sequence is the length of the shortest LFSR generating the sequence.

Definition 8 (Connection Polynomial)**.** Suppose an LFSR produces a sequence via the relation $s_n + c_1 \cdot s_{n-1} + \ldots c_L \cdot s_{n-L} = 0$. Then the connection polynomial of this LFSR is defined to be $C(D) := 1 + c_1 D + \ldots c_L D^L$.

Remark 2. The linear complexity need not be equal to the degree of the minimal connection polynomial for *finite* sequences (see the example below). However, these two quantities are equal when considering infinite periodic sequences with a finite period.

Example 4. Suppose that we are working in the field $\mathbb{Z}_7$ and consider the sequence $(3, 2, 3, 1, 3, 2, 4)$. It can be shown that a LFSR with 4 registers with a connection polynomial of $C(D) = 1 + 4D + 6D^2 + 5D^3$ can be used to generate this sequence. To check this, note that the initial loading of the 4 registers would be $(3, 2, 3, 1)$ and we have that $3 + 4 + 6 \cdot 3 + 5 \cdot 2 = 0$, $2 + 4 \cdot 3 + 6 + 5 \cdot 3 = 0$ etc. A pictorial representation of this LFSR is given in Figure 5.

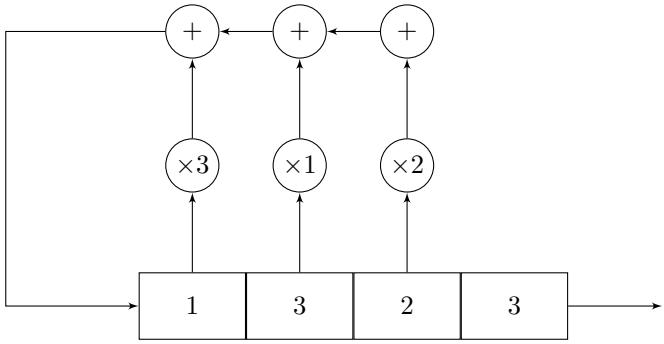

Figure 5: A minimal length LFSR generating the finite sequence $(3, 2, 3, 1, 3, 2, 4)$ over $\mathbb{Z}_7$. Note that the coefficients of the connection polynomial are the negation of the multiplicands in this diagram. This LFSR has length 4, yet the minimal polynomial is of degree 3.

The linear complexity and minimal connection polynomial of any finite (or infinite periodic) sequence can be calculated in polynomial time using the Berlekamp-Massey algorithm [Mas69]. This algorithm is extremely generic as it accounts for sequences over *any* field and does not restrict to periodic sequences.

We now briefly overview the structure of the Berlekamp-Massey algorithm. Suppose we wish to find the linear complexity and connection polynomial of the finite sequence $(a_0, \ldots, a_{n-1})$. Then the Berlekamp-Massey algorithm iteratively calculates the linear complexity and connection polynomial of each subsequence $a_0, \ldots, a_i$ for $i = 0, \ldots, n-1$. Suppose we have just completed the $(k-1)^{th}$ loop and have arrived at a linear complexity of l_{k-1} and connection polynomial $C^{(k-1)}(D) := 1 + c_1^{(k-1)} D + \cdots + c_{l_{k-1}}^{(k-1)} D^{l_{k-1}}$ (recall that some of these coefficients may be 0) for the subsequence $(a_0, \ldots, a_{k-1})$. To start the k^{th} iteration, we calculate the *discrepancy* defined to be $d := a_k + \sum_{i=1}^{l_{k-1}} c_1^{k-i} a_{k-i}$. This tells us how far $C^{k-1}(D)$ is from being the connection polynomial of the subsequence $(a_0, \ldots a_k)$. There are three cases to consider when updating the linear complexity and connection polynomial:

1. If $d = 0$, then the linear complexity and connection polynomial remain the same.
2. If $d \neq 0$, there are two sub-cases:
   (a) If $2 \cdot l_{k-1} > k + 1$, then the connection polynomial must change but the linear complexity stays the same.
   (b) If $2 \cdot l_{k-1} \leq k + 1$ then the connection polynomial changes and the linear complexity increases.

The Berlekamp-Massey algorithm gives explicit formulae for updating linear complexities and connection polynomials depending on which of the three cases is relevant. For a rigorous proof of correctness, see [Mas69]. The pseudocode for the Berlekamp-Massey algorithm is given as Algorithm 1.

Algorithm 1 The Berlekamp-Massey algorithm

Input: $s = (s_0, \ldots, s_{n-1})$
Output: Linear complexity of s and connection polynomial $(L, C(D))$
% Initialisation
$C(D) \leftarrow 1$; $B(D) \leftarrow 1$; $x \leftarrow 1$;
$L \leftarrow 0$; $b \leftarrow 1$; $N \leftarrow 0$
% Main Loop
while $N < n$ **do**
  $(c_0, c_1, \ldots, c_{n-1}) \leftarrow C(D)$.Coefficients();
  $d \leftarrow s_N + \sum_{i=1}^{L} c_i s_{N-i}$; % the discrepancy
  % Case 1: no updates required
  if $d = 0$ **then**
    $x \leftarrow x + 1$;
    continue;
  % Case 2: update only the connection polynomial
  else if $d \neq 0$ and $2L > N$ **then**
    $C(D) \leftarrow C(D) - db^{-1} D^x B(D)$;
    $x \leftarrow x + 1$;
    continue;
  % Case 3: update both linear complexity and connection polynomial
  else
    $T(D) \leftarrow C(D)$;
    $C(D) \leftarrow C(D) - db^{-1} D^x B(D)$;
    $L \leftarrow N + 1 - L$;
    $B(D) \leftarrow T(D)$;
    $b \leftarrow d$;
    $x \leftarrow 1$;
  end if
  $N \leftarrow N + 1$;
end while
return $(L, C(D))$

The second component of our attack is the following theorem:

Theorem 1 (Blahut)**.** *Let q be a prime such that there exists an n^{th} primitive root of unity in $\mathbb{Z}_q$ and let $\mathsf{NTT}(\cdot)$ denote a traditional NTT[13] of dimension n over $\mathbb{Z}_q$. For any $s \in \mathbb{Z}_q$, define $(\hat{s}) := (\mathsf{NTT}(s), \mathsf{NTT}(s), \ldots)$ to be the sequence comprising of infinitely many copies of $\mathsf{NTT}(s)$. Then $\mathsf{LC}((\hat{s})) = \mathsf{HW}(s)$.*

[13]Note that the matrix associated to the traditional NTT has $(i, j)^{th}$ component given by ω^{ij}

Blahut's Theorem has been proven for the traditional NTT. However, in this work we are considering the *negacyclic* NTT. It turns out that the correctness of Blahut's Theorem for the negacyclic NTT follows straight-forwardly from the traditional case:

Lemma 1 (Negacyclic Blahut)**.** *Let q be a prime such that there exists an $2n^{th}$ primitive root of unity in $\mathbb{Z}_q$ and let $\mathsf{NTT}(\cdot)$ denote a negacyclic NTT of dimension n over $\mathbb{Z}_q$. For $s \in \mathbb{Z}_q^n$, define $(\hat{s}) := (\mathsf{NTT}(s), \mathsf{NTT}(s), \dots)$ to be the sequence comprising of infinitely many copies of $\mathsf{NTT}(s)$. Then, from Blahut's theorem for the traditional NTT, $\mathsf{LC}((\hat{s})) = \mathsf{HW}(s)$.*

Proof. In this proof, we denote whether an NTT is negacyclic or traditional using *neg* or *trad* in the subscript. Let $\omega \in \mathbb{Z}_q$ be a primitive n^{th} root of unity and $\gamma \in \mathbb{Z}_q$ be a square root of ω. Also let $\vec{g} = (1, \gamma, \gamma^2, \dots, \gamma^{n-1})$ and $\odot$ denote the component-wise multiplication of vectors. We then have

$$\mathsf{NTT}_{neg}(s)_i = \sum_{j=0}^{n-1} \omega^{ij} s_j = \sum_{j=0}^{n-1} \omega^{ij} (\gamma^j s_j) = \mathsf{NTT}_{trad} x(\vec{g} \odot s)_i \tag{12}$$

Defining the infinite sequence $(\widehat{\vec{g} \odot s}_{trad}) := (\mathsf{NTT}_{trad}(\vec{g} \odot s), \mathsf{NTT}_{trad}(\vec{g} \odot s), \dots)$, we have that

$$\mathsf{LC}((\hat{s})) = \mathsf{LC}((\widehat{\vec{g} \odot s}_{trad})) = \mathsf{HW}(\vec{g} \odot s) = \mathsf{HW}(s) \tag{13}$$

where the first equality is due to Equation (12), the second is due to Blahut's theorem for the traditional NTT, and the last is due to the fact that $\gamma^j s_j = 0$ if and only if $s_j = 0$. □

Blahut's theorem tells us that a secret with Hamming weight w corresponds precisely to an infinite sequence in the NTT domain with linear complexity w. In order to exploit this relation, we use the Berlekamp-Massey algorithm [Mas69] which provides a method for finding the linear complexity and connection polynomial of any finite sequence.

To investigate this further, we can produce a linear complexity *profile* for a sequence by plotting the maximal index present in the subsequence in each iteration against the linear complexity calculated for that subsequence. This is a trivial task when considering the previously mentioned structure of the Berlekamp-Massey algorithm. For the linear complexity profile of a random sequence, we typically end up observing that the points on the profile exhibit a step behaviour roughly lying on the line $y = x/2$. However, if our sequence is the result of a NTT transform of a low Hamming weight vector, Blahut's theorem tells us that we should get low linear complexity. In this case, the linear complexity profile shows the same step behaviour, but levels off when the low linear complexity of the sequence is reached. Examples of linear complexity profiles in both these cases in given in Figure 6.

A.2 Attack description

Suppose we are given a noisy version of a secret key with low Hamming weight w in the NTT domain. We will show that the Berlekamp Massey algorithm implicitly yields a strategy for finding such a key given $2w$ consecutive error-less symbols. The logic behind the attack is that the connection polynomial must be recovered fully once $2w$ symbols have been considered. A consequence of this is that the attack in Algorithm 2 works if there are $2w$ clean symbols in the noisy key. Note that if we were to disregard the NTT, leaking $2w$ symbols of the secret key does not lead to an immediate key recovery attack.

Lemma 2. *For a prime q, integer n and vector $s \in \mathbb{Z}_q$ with Hamming weight w, the minimal connection polynomial of $\hat{s} := \mathsf{NTT}(s)$ can be recovered given $2w$ consecutive symbols of $\hat{s}$.*

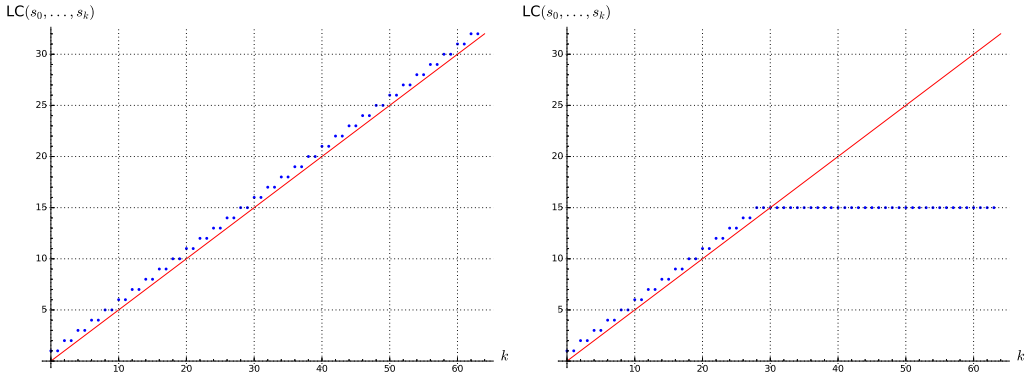

Figure 6: Linear complexity profiles for a random sequence and for the NTT of a low Hamming weight vector.

Proof. Suppose that our linear complexity has reached w (which is its maximum value for the error-less NTT sequence) after the consideration of the first $2w$ symbols. We analyse the loop in the Berlekamp-Massey algorithm that considers $2w+1$ symbols. Since we know that the linear complexity cannot increase, we must either be in case 1 or 2 from Algorithm 1. However, to be in case 2, we must have $2L > N$ which translates to $2w > 2w+1$ for the loop in consideration. This is clearly impossible, so we must be in the case where the connection polynomial does not change. The same argument holds for the remaining iterations.

To complete the argument, we need to show that the linear complexity after $2w$ iterations is in fact w. Suppose not, i.e. that we have a linear complexity of $w' < w$. Then at some point in the remaining iterations, we must increase the linear complexity to w. Suppose the first increase occurs for $N = 2w+k$ for some $k \geq 0$. Then we must be in case 3 from Algorithm 1, so we update the linear complexity to $2w+k+1-w' > w$ which is a contradiction. Therefore we must reach the linear complexity of w after $2w$ symbols have been considered. □

Note that we can change the starting point of the sequence $\hat{s}$ without changing the proof of the result above. Therefore in the attack, we do not require that the $2w$ error-less symbols occur in the first components of $\hat{s}$. If we do not know where the error-less symbols are, we can simply re-run the Berlekamp-Massey algorithm on all of the cyclic shifts of $\hat{s}$ in an attempt to get the error-less symbols at the beginning of the sequence.

A general framework for this attack is given as Algorithm 2. Note that this algorithm outputs a list of candidates given a noisy NTT secret. A simple way to find the solution within this list would be to check whether $b - a \cdot s$ is small (in the non-NTT domain) for each candidate.

A.3 Cold boot scenario

We now consider the Blahut-Berlekamp Massey attack within an NTT cold boot scenario. We will work with RLWE parameters $n, q, w := \mathsf{HW}(s)$. Recall that we need $2w$ consecutive clean symbols for the attack to go through which is equivalent to requiring $2w\lceil \log_2 q \rceil$ consecutive *bits* of the secret key. When considering these bits in a noisy version of the secret key, about half of the bits will be out of the ground state indicating that they have not flipped. Therefore, assuming a bit-flip rate of ρ_0 towards the ground state and a bit-flip of ρ_1 away from the ground state, we expect $(\rho_0 + \rho_1)w\lceil \log_2 q \rceil$ bit-flips within the entire block of $2w\lceil \log_2 q \rceil$ bits. The strategy is to exhaustively search for the bits that were flipped and run the Berlekamp-Massey algorithm to check each guess. Ignoring the

Algorithm 2 Generic attack based on Berlekamp-Massey algorithm

Input: $\tilde{\hat{s}} = (\tilde{s}_0, \ldots, \tilde{s}_{n-1})$ % noisy NTT of secret, Hamming weight $w < n/2$
Output: List of candidate secrets $\mathcal{L}$
$\mathcal{L} \leftarrow \emptyset$;
for $i = 0, \ldots, n-1$ **do**
  $(t_0, \ldots, t_{2w-1}) \leftarrow (s_i, \ldots, s_{2w+i})$;
  $(L, C(D)) \leftarrow$Berlekamp-Massey$(t_0, \ldots t_{2w-1})$;
  $(c_0, \ldots, c_w) \leftarrow C(D)$.coefficients();
  for $j = 2w, \ldots, n-1$ **do**
    $t_j \leftarrow -\sum_{k=1}^{w} c_i t_{j-i}$; % derive the remaining symbols
  end for
  for $j = 0, \ldots, n-1$ **do**
    $r_j = t_{j-i \bmod n}$;
  end for$\mathcal{L}$.Add$((r_0, \ldots, r_{n-1}))$;
end for
return $\mathcal{L}$

trivial cost of running Berlekamp-Massey, we have a rough average complexity of

$$\binom{w\lceil \log_2 q \rceil}{\lfloor \rho_0 w \lceil \log_2 q \rceil \rceil} \cdot \binom{w\lceil \log_2 q \rceil}{\lfloor \rho_1 w \lceil \log_2 q \rceil \rceil}. \tag{14}$$

For example parameters $w = 64, q = 12289, n = 1024$, we have an attack with complexity roughly 2^{80} for bit-flip rate $\rho_0 = 1\%, \rho_1 = 0.1\%$ remembering that ρ_1 is the retrograde flip rate (if $\rho = 0.17\%$, the attack complexity is roughly 2^{28}). In certain scenarios, this complexity could be much lower. For example, suppose there is a block of $2w\lceil \log_2 q \rceil$ consecutive bits where the majority flips could have only occurred away from the ground state. Then we expect only a small number of bit-flips in this block (since $\rho_1 < \rho_0$), which reduces the amount of guesses required before the attack is successful. Therefore, in a cold boot attack, we would be able to identify the optimal consecutive block of $2w$ symbols to launch our attack on very easily.

A.4 Future directions for linear complexity attacks

The k-error linear complexity of a sequence is the lowest minimal complexity attainable when changing at most k symbols. This notion corresponds closely to the case where we have a noisy version of the key that contains at most k erroneous symbols. If we had an algorithm that computed k-error linear complexity along with the symbol changes required to minimise the linear complexity, then we would be able to recover secret keys in many non-trivial cases.

However, efficient algorithms for calculating k-error linear complexities only exist for specific classes of sequences [SM93, KUI99]. There is currently no efficient algorithm that handles sequences with power-of-two period n over a field $GF(q)$ satisfying $2n|(q-1)$. It is an interesting open problem to discover such an algorithm.

B Alternative algorithms

B.1 Meet in the middle attack analysis

Meet in the middle attacks offer improvements in terms of time over exhaustive search at the expense of increased memory requirements. It seems feasible that combinatorial meet in the middle attacks are compatible with our cold boot scenarios from Section 4 since we

are recovering a low entropy secret Δ that is not small in Euclidean norm (see Equation 2). The idea is to split Δ into a left half $\Delta_{(l)} \in \mathbb{Z}_q^{n/2}$ and a right half $\Delta_{(r)} \in \mathbb{Z}_q^{n/2}$. In addition to this, define W_i to be the i^{th} *row* of the inverse NTT matrix and $W_{i,(l)}$ ($W_{i,(r)}$) to be the left (resp. right) half of this i^{th} row. The meet in the middle attack hinges on the relations

$$\tilde{s}_i - W_{i,(l)} \cdot \Delta_{(l)} \approx W_{i,(r)} \cdot \Delta_{(r)} \bmod q, \quad i = 0, \ldots, n-1, \tag{15}$$

where the i^{th} approximate equality is up to an error given by the i^{th} coefficient of the true secret s which is assumed to be small for practical constructions.

We assume that the bit errors are uniformly spread across our noisy key i.e. that there are $\kappa/2$ bit errors in both the left and right halves of our noisy key. We then pick a particular candidate $\Delta^*_{(l)}$ and calculate the components of the vector arising from evaluating the LHS of Equation (15). We denote the resulting vector by $\mathbf{t}_{\Delta^*_{(l)}}$ and store the pair $(\mathbf{t}_{\Delta^*_{(l)}}, \Delta^*_{(l)})$ in a table T. This process is repeated for all valid choices of $\Delta^*_{(l)}$.

Next we consider the value of the RHS for each candidate $\Delta_{(r)}$ and check the approximate equality given in Equation (15) with each entry of the table T. In particular, we are looking for pairs $\Delta_{(l)}$ and $\Delta_{(r)}$ such that the error in Equation (15) is a valid sample from the distribution that s was drawn from. Enumerating over all $\Delta_{(r)}$ produces a list of candidates for the full vector Δ.

B.1.1 Locality sensitive hashing

The description above should serve as an intuition rather than a guide to implementing such an attack in practice. Techniques such as locality sensitive hashing (LSH) can significantly decrease the computational cost of meet in the middle attacks in this setting [dBDJdW18]. LSH essentially offers a method of efficiently finding the most likely table entries for a given candidate $\Delta^*_{(r)}$ by organising the table entries into hash buckets according to some measure of closeness. More concretely, for some similarity measure D giving rise to the definition of a "ball" $B(p, r) := \{p' : D(p, p') \leq r\}$ around point p with radius r, we can define a locality sensitive hash family as follows:

Definition 9 (LSH Family [IM98])**.** A family of functions $\mathcal{H} := \{h : S \to U\}$ is (r_1, r_2, p_1, p_2)-sensitive with respect to D if $\forall\ p, p', p'' \in S$

- if $p' \in B(p, r_1)$, then $\Pr_{\mathcal{H}}[h(p') = h(p)] \geq p_1$,
- if $p'' \notin B(p, r_2)$ then $\Pr_{\mathcal{H}}[h(p'') = h(p)] \leq p_2$,

where $\Pr_{\mathcal{H}}$ denotes a probability when h is uniformly sampled from $\mathcal{H}$.

Once this family is chosen, the standard LSH strategy is to construct different hash functions, each consisting of μ uniform random functions from $\mathcal{H}$ i.e. pick

$$g_j(\cdot) = (h_{j,1}(\cdot), \ldots, h_{j,\mu}(\cdot)), \quad j = 1, \ldots \phi. \tag{16}$$

where each of the $h_{(\cdot,\cdot)}$ are chosen uniformly at random from $\mathcal{H}$. We then create one hash table per function g_i and store our data points in the appropriate hash buckets. Now suppose we want to query the database on point p to see if there is a similar point in the database. Then we simply calculate the values $g_1(p), \ldots, g_\phi(p)$ and compare with all of the data points in these ϕ buckets.

A concise summary of the runtime and space requirements of LSH is given in [Laa14] in the form of the following lemma.

Lemma 3. *Suppose there exists a (r_1, r_2, p_1, p_2)-sensitive family $\mathcal{H}$. For a list L of size N, let*

$$\rho = \frac{\log 1/p_1}{\log 1/p_2}, \quad \mu = \frac{\log N}{\log 1/p_2}, \quad \phi = cN^\rho$$

for some constant c. Then for any v in the appropriate space, we can either (a) find an element $w^* \in L$ such that $D(v, w^*) \leq r_2$, or (b) conclude that with high probability $(\geq 1 - e^{-c})$ that for all $w \in L$, $D(v, w) \geq r_1$. Let τ_h be the time taken to compute a hash, m_h be the storage size of a hash, and τ_D be the time to calculate $D(\cdot, \cdot)$. The algorithm requires:

- *Preprocessing time:* $N\mu\phi \cdot \tau_h$
- *Hash table storage:* $N\mu\phi \cdot m_h$
- *Single query time:*
  - *Hash evaluation:* $\mu\phi \cdot \tau_h$
  - *Expectation of comparison time:* $cN^\rho \cdot \tau_D$

Note that the lemma only considers finding a single vector in the ball of radius r_2. Our search problem asks to find all vectors in this ball. Therefore, we would have to search through all candidates output by LSH. Nonetheless, we will assume that our LSH family is good enough to ensure that our list of candidates ends up being very short.

Next, we must choose a similarity measure along with a family $\mathcal{H}$. We choose to work with the Euclidean norm using the hash family from [AI06]. A single hash is computed by a projection onto a random 24-dimensional plane followed by a translation and then a very cheap (< 519 real operations) Leech lattice decoding procedure. Letting ν denote the initial dimension of the data being hashed, we approximate $\tau_h \approx 24\nu$, $m_h \approx 24$ and $\tau_D = \nu$ in Lemma 3.

We now return to our running example of the Kyber scheme ($n = 256, q = 7681, \sigma = \sqrt{2}$) with κ bit-flips per secret ring element. As before, we consider attacking each of the secret ring elements individually. We have that the dimension of each vector $\Delta_{(l)}$ is $\nu = 128$ and make the simplifying assumption that there are no retrograde bit-flips[14]. This means that there are $N = \binom{13 \cdot 128/2}{\lfloor \kappa/2 \rfloor}$ possibilities for this vector taking into account that the ground state of memory makes it clear when a bit-flip has not occurred and that approximately half of the secret bits will be out of the ground state. Thus the number of unknown bits in $\Delta_{(l)}$ is $13 \cdot 138/2$ (rather than $13 \cdot 128$). Note that we have rounded $\kappa/2$ down to the nearest integer and are implicitly assuming $\binom{13 \cdot 128/2}{\lceil \kappa/2 \rceil}$ choices for $\Delta_{(r)}$. Since the secret in Kyber actually comes from a binomial distribution, we have that each coefficient lies in the set $\{-4, 3, \ldots, 3, 4\}$. Therefore, the maximum Euclidean length of the secret is $\sqrt{256 \cdot 4} = 32$. The values of r_1 and r_2 chosen should be compatible with this. One particular choice might be $r_1 = 22.85$ and $r_2 = 32$ which corresponds to values $p_1 = 0.0177896$ and $p_2 = 0.0013332$ according to the empirical analysis of [AI06]. In the statement of Lemma 3, we can take $\rho = 0.609$, $\mu = 6.62 \cdot \log\left(\binom{13 \cdot 128/2}{\kappa/2}\right)$ and $c = 2$. If we made these choices, a summary of the costs would be:

- Preprocessing time: $6.62 \cdot \binom{832}{\lfloor \kappa/2 \rfloor}^{1.609} \log(\binom{832}{\lfloor \kappa/2 \rfloor}) \cdot (24 \cdot 128)$
- Storage: $24 \cdot 6.62 \cdot \binom{832}{\lfloor \kappa/2 \rfloor}^{1.609} \log(\binom{832}{\lfloor \kappa/2 \rfloor})$
- Query time per guess:
  - Hash evaluations: $6.62 \cdot \binom{832}{\lfloor \kappa/2 \rfloor}^{0.609} \log(\binom{832}{\lfloor \kappa/2 \rfloor}) \cdot (24 \cdot 128)$
  - Comparison time: $2 \cdot \binom{832}{\lfloor \kappa/2 \rfloor}^{0.609} \cdot 128$

[14] We show that this attack is not competitive, even under making simplifying assumptions

We note that the time complexity of the preprocessing step and the guessing step (assuming N guesses) roughly balance each other out. Therefore, the running time of the attack is around twice the preprocessing time. Using this heuristic, we obtain a 2^{120} attack for the cold boot case of $\rho_0 = 1\%$ bit-flip rate to ground state, ignoring retrograde bit-flips ($\kappa = 17$). We can divide and conquer the instance from dimension 256 to 32 before running meet in the middle to attempt to bring down the complexity. However, it becomes harder to consider the ground state memory to rule out bit-flips at this dimension. Nonetheless, assume that we can rule out flips for half of the bits that we consider even at dimension 32. Rerunning the analysis with the same ratio r_2/r_1 does yield a significant improvement to time complexity of 2^{79}. However, despite making optimistic assumptions, this attack is still not competitive when comparing to our lattice attacks despite significantly larger memory requirements.

B.2 Arora-Ge attacks

Another algorithm for solving LWE is due to Arora and Ge [AG11]. The main idea is to set up a system of non-linear but noiseless equations whose solution is the LWE secret. We briefly describe the attack in our setting.

Recall that our LWE-like sample is $(W, \tilde{s} := W\Delta + s)$. Working under the assumption that s has small coefficients, the polynomial $P(X) := X\prod_{i=0}^{t}(X-i)(X+i)$ satisfies $P(s_i) = 0$ with high probability for some large enough t. Rewriting, we have that $P((b - W\Delta)_i) = 0$ with high probability. Note that this formulation of the problem makes no use of s being small but instead relies on s having low entropy. In a similar fashion, we can add equations encoding the solution set for each Δ_i as $Q(X) := X\prod_{i=0}^{\lceil \log_2 q \rceil}(X+2^i)(X-2^i)$. Finally, to encode the number of bit-flips κ, we may add equations $R(X) := \prod_{i\in\mathcal{S}_j} X_i$ where $\mathcal{S}$ is the set containing all subsets of $\{0, 1, \ldots, n-1\}$ with $\kappa+1$ elements and $\mathcal{S}_j \in \mathcal{S}$, i.e. a set of indices. Since there are only κ non-zero Δ_i, any product of $\kappa+1$ of them must evaluate to zero.

Arora and Ge use linearisation to solve the system of equations which requires a very large number of samples. Linearisation is a special case of a Gröbner basis computation and an improved analysis using Gröbner bases was carried out in [ACFP14] which, in principle, works for any number of samples such that the problem is well-defined. Thus, the attack then reduces to computing a Gröbner basis for a system in n unknowns, given n equations of degree $2t+1$, n equations of degree $2\lceil \log_2 q \rceil + 1$, and $\binom{n}{\kappa+1}$ equations of degree $\kappa+1$. For Kyber, we take $t = 4$ and $\lceil \log_2 q \rceil = 13$ and fold down to $n = 32$. Using the `gb_cost` function from the estimator of [APS15], we obtain a cost of $2^{117.3}$ operations which is not competitive with our lattice attack.

C Proof of concept implementation

C.1 BSDR

```
"""
An (inefficient) set of functions that compute minimal BSDR lists
"""

from sage.all import vector, cached_function, ZZ


def bsdr_from_vec(v, bits):
    """
    Returns a list of all minimal BSDR representations of a vector v

    :param v: input vector
    :param bits: number of bits in BSDR for each component of v

    EXAMPLE::

        sage: load("bsdr.py")
        sage: set_random_seed(0)
        sage: v = vector([randint(-15,15) for _ in range(4)])
        sage: v
```

```
        (-12, 0, -14, -5)
        sage: bsdr_from_vec(v,5)
        [(0, 0, 1, 0, -1, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1, -1, 0, -1, 0, 0),
        (0, 0, -1, -1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1, -1, 0, -1, 0, 0)]

    """
    components_list = []
    for vi in v:
        components_list.append(bsdr_from_int(vi, bits))
    return create_vecs_from_components(components_list)


def create_vecs_from_components(components_list):
    """
    The ith list in components_list should be the the list of possibilities of the ith component of
    the original vector.
    """
    n = len(components_list)
    current_list = components_list[0]
    for i in range(1, n):
        next_list = add_prefix_to_suffices(current_list, components_list[i])
        current_list = next_list
    return map(vector,  current_list)


@cached_function
def bsdr_from_int(s, bits):
    """
    Return a list of minimal BSDRs of an integer with some maximal number of bits

    :param s: an integer
    :param bits: the maximum number of bits in the output

    EXAMPLE::

        sage: load("bsdr.py")
        sage: bsdr_from_int(31,6)
        [[-1, 0, 0, 0, 0, 1]]

    """
    if bits == 0:
        return [[]]
    if s == 0 or s % 2**(bits) ==0:
        return [[0]*bits]

    for i in range(bits):
        if s % 2**(i+1) != 0:
            break

    # try two lists, one for -1 and one for +1
    prefix1 = [0]*i + [1]
    prefix2 = [0]*i + [-1]

    suffixes1 = bsdr_from_int((s-2**i)//(2**(i+1)), bits-(i+1))
    suffixes2 = bsdr_from_int((s+2**i)//(2**(i+1)), bits-(i+1))

    l1 = add_prefix_to_suffices(prefix1, suffixes1)
    l2 = add_prefix_to_suffices(prefix2, suffixes2)

    entire_list = l1+l2
    purge_list(s, entire_list)

    min_list = minimal_hw_list(entire_list)
    return min_list


def purge_list(target, ll):
    """
    Purge a list of candidates ll of all wrong BSDRs

    :param target: the integer we want to express in BSDR
    :param ll: list of candidate BSDRs
    """
    for l in ll:
        if ZZ(l, base=2) != target:
            ll.remove(l)


def add_prefix_to_suffices(prefixes, suffixes):
    """
    Combine a list of prefixes with a list of suffixes.

    :param prefixes: list of prefixes
    :param suffixes: list of suffixes

    EXAMPLE::

        sage: load("bsdr.py")
        sage: prefixes = [[1,2,3],[4,5,6]]; suffixes = [[10,20,30],[40,50,60]]
        sage: add_prefix_to_suffices(prefixes,suffixes)
        [[1, 2, 3, 10, 20, 30],
         [1, 2, 3, 40, 50, 60],
         [4, 5, 6, 10, 20, 30],
         [4, 5, 6, 40, 50, 60]]
    """
    # prefixes is a list of prefixes
    if type(prefixes[0]) == list:
        output = []
        for i in prefixes:
            for j in suffixes:
                output.append(i+j)
        return output
    else:  # then prefixes is a single prefix
        output = []
```

```
        for i in suffixes:
            output.append(prefixes+i)
        return output


def minimal_hw_list(inlist):
    """
    Return the minimal hamming weight entries of the input list

    :param inlist: a list of bit string lists of the same length

    """
    outlist = []
    min_hw = len(inlist[0])+1   # gets reset in the first loop
    for l in inlist:
        hw = vector(l).hamming_weight()
        if hw == min_hw:
            outlist.append(l)
        elif hw < min_hw:
            del(outlist)
            outlist = []
            outlist.append(l)
            min_hw = hw

    return outlist
```

C.2 Proof of concept attack

```
# -*- coding: utf-8 -*-
"""
Proof of concept implementation of cold boot attacks on R/MLWE under the NTT

We consider the following scenario. Let `s` be some vector of length `n` with small elements mod
`q`.  We are given `\hat{s} = NTT*s + Δ` (in the code below, we use ``d`` to denote `Δ` to highlight
it is a vector, not a matrix) and want to recover `s`.  We write: `c = NTT^{-1}\hat{s} = s + iNTT
· Δ`.

..  note :: Run tests by ``sage -t poc.py``

EXAMPLE::

    sage: load("poc.py")
    sage: ip = InstanceParams(n=256, q=7681, eta=4, kappa=19)
    sage: pp = PreprocParams(q=7681, fold=3, alpha=5, beta=2)

"""
import sys
from sage.all import ceil, log, vector, ZZ, shuffle, GF, RR, QQ, sqrt
from sage.all import matrix, set_random_seed, show, load, line
from sage.all import parent, binomial, cached_function, median
from fpylll import BKZ
from collections import namedtuple

# HACK
sys.path.append(".")
from bsdr import bsdr_from_int  # noqa



"Utility functions"


def basis_shape(A, showit=True):
    """
    A's geometry

    :param A: an integer matrix
    :param showit: show a plot of the lengths of the Gram-Schmidt vectors

    """
    from fpylll import IntegerMatrix, GSO
    A = IntegerMatrix.from_matrix(A)
    M = GSO.Mat(A)
    M.update_gso()
    r = M.r()
    mmin = min(r)
    mmax = max(r)
    print "min: (%3d, %6.1f)"%(r.index(mmin), mmin),
    print "max: (%3d, %6.1f)"%(r.index(mmax), mmax),
    print
    if showit:
        show(line(zip(range(M.d), map(lambda x: log(x, 2), M.r()))))


def balance(e, q=None):
    """
    Return a balanced representation between `-q/2` and `q/2` for `e`.

    :param e: a scalar, polynomial or vector
    :param q: optional modulus

    """
    try:
        p = parent(e).change_ring(ZZ)
        return p([balance(e_, q=q) for e_ in e])
    except (TypeError, AttributeError):
        if q is None:
            try:
                q = parent(e).order()
            except AttributeError:
                q = parent(e).base_ring().order()
```

```
        return ZZ(e)-q if ZZ(e)>q/2 else ZZ(e)


def snort(s, q, ell=None, b=None):
    """
    Return the vector `s` written in base `2^\ell` scaled by factor `\theta`

    :param s: a vector (over the integers)
    :param q: an integer modulus
    :param ell: the vector will be rewritten in base `2^\ell`
    :param b: if not ``None`` only consider the lower `2^{b\ell}` bits.
              This assume the values of `s` are bounded by `2^{b\ell}`

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: d = bitflipf(n=16, q=7681, kappa=10)
        sage: snort(d, 7681)
        (-1, 32, 0, 32, 0, 0, 4, 0, 0, -32, 0, 0, 0, 0, -32, -4, 0, 0, 0, 0, 0, 0, 0, 0, 32, 0, 0, 0, 0, 1, -4)

    """
    ell = PreprocParams.ellf(q, ell)
    v = []
    b = PreprocParams.bf(q, ell, b)
    for s_ in s:
        s_ = ZZ(s_)
        s_ = bsdr_from_int(s_, ell*b)[0]
        for i in range(0, ell*b, ell):
            v.append(ZZ(list(s_[i:i+ell]), base=2))
    return vector(v)


def sneeze(v, q, ell=None, theta=1, b=None):
    """Inverse of snort

    :param v: vector holding `2^\ell` decomposition of a vector `s`
    :param q: an integer modulus
    :param ell: the vector will be rewritten in base `2^\ell`
    :param theta: the rewritten vector will be scaled by `\theta` in QQ`
    :param b: if not ``None`` only consider the lower ``2^{b\ell}` bits

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: d = bitflipf(n=16, q=7681, kappa=10)
        sage: d_ = snort(d, 7681)
        sage: d == sneeze(d_, 7681)
        True

    """
    s = []
    ell = int(PreprocParams.ellf(q, ell))
    b = int(PreprocParams.bf(q, ell, b))
    base = 2**ell
    v = list(v)
    if theta == 1:
        s = vector([ZZ(v[i:i+b], base=base) for i in range(0, len(v), b)])
    else:
        s = vector([ZZ(v[i:i+b], base=base)//theta for i in range(0, len(v), b)])
    return s



"""Our instances are characterised by `n, q, \sigma, \kappa` where the first three are LWE-like parameters and
the last one denotes the hamming weight of `\Delta` considered as a bitstring.
"""


class InstanceParams(namedtuple("InstanceParams", ["n", "eta", "q", "kappa"])):
    @property
    def sd(self):
        return sqrt(self.eta)


@cached_function
def intt_matrix(n, q, omega=None, gamma=None):
    """
    Inverse NTT matrix.

    :param n: dimension
    :param q: modulus
    :param omega: enforce a choice for omega
    :param gamma: enforce a choice for gamma

    EXAMPLE::

        sage: load("poc.py")
        sage: W = intt_matrix(8, 7681)
        sage: W*vector(ZZ, 8*[1]) ## constant polynomial evaluated at anything is `1`
        (1, 0, 0, 0, 0, 0, 0, 0)

    """

    K = GF(q)
    if omega is None and gamma is None:
        om = K(1).nth_root(n)
        gam = om.nth_root(2)
    else:
        om = K(omega)
        gam = K(gamma)
    inv_n = K(n)**(-1)
    W = matrix(K, n, n)
```

```
    for i in range(n):
        for j in range(n):
            W[i, j] = inv_n * gam**(-i) * om**(-i*j)
    return W


@cached_function
def ntt_matrix(n, q, omega=None, gamma=None):
    K = GF(q)
    if omega is None and gamma is None:
        om = K(1).nth_root(n)
        gam = om.nth_root(2)
    else:
        om = K(omega)
        gam = K(gamma)
    W = matrix(K, n, n)
    for i in range(n):
        for j in range(n):
            W[i, j] = gam**(j) * om**(i*j)
    return W


def bitflipf(n, q, kappa, s_hat=None):
    """
    Sample a secret vector `Δ` of length `n` mod `q` with `κ` bits set in random positions.

    :param n: dimension
    :param q: an integer modulus
    :param k: number of non-zero bits
    :param s_hat: clean NTT(s) to derive the bit flip signs from

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: d = bitflipf(16, 7681, 10); d
        (4095, 4096, 0, 4, -4096, 0, 0, -544, 0, 0, 0, 0, 4096, 0, 0, -511)

        sage: bitstring = sum(map(lambda x: bsdr_from_int(x, 13), d), [])
        sage: # len(bitstring)-bitstring.count(0)
        10

    """
    ell = ceil(log(q, 2))
    positions = [1]*kappa + [0]*(n*ell-kappa)
    shuffle(positions)

    d = []
    if s_hat is None:
        for i in range(n):
            d.append(sum((-1)**ZZ.random_element(0, 2) * positions[i*ell+j] * 2**j for j in range(ell)))
            # d.append((-1)**ZZ.random_element(0, 2)*sum(positions[i*ell+j]*2**j for j in range(ell)))
    else:
        for i in range(n):
            bin_s_hat = s_hat[i].digits(base=2, padto=ell)
            d.append(sum(((-1)**(bin_s_hat[j]))*positions[i*ell+j]*2**j for j in range(ell)))

    return vector(ZZ, d)


def instance(params, seed=None):
    """
    Generate a new LWE-like challenge for coldboot attacks, `c = W·d + s`, with `s` small and `d`
    coming from cold boot bit flips.

    :param params: instance parameters
    :param seed: a random seed (or ``None``)

    EXAMPLE::

        sage: load("poc.py")
        sage: ip = InstanceParams(n=16, q=7681, eta=4, kappa=10)
        sage: instance(ip, seed=0) == instance(ip, seed=0)
        True

        sage: (W,c), (d,s) = instance(ip, seed=1)
        sage: c == W*d + s
        True

    """
    if seed is not None:
        set_random_seed(seed)

    if params.q == 7681 and params.n <= 256:
        # page 4 of Kyber spec
        gamma = GF(params.q)(62)
        omega = GF(params.q)(3844)
        n_ = params.n
        while n_ < 256:
            gamma = gamma**2
            omega = omega**2
            n_ = 2*n_
        assert(omega**(params.n) == 1)
        assert(gamma**(2*params.n) == 1)
        assert(omega**(params.n//2) == -1)
        assert(gamma**(params.n) == -1)
    elif params.q == 12289 and params.n <= 1024:
        # newhope spec
        gamma = GF(params.q)(7)
        omega = GF(params.q)(49)
        n_ = params.n
        while n_ < 1024:
            gamma = gamma**2
            omega = omega**2
```

```
            n_ = 2*n_
        assert(omega**(params.n) == 1)
        assert(gamma**(2*params.n) == 1)
        assert(omega**(params.n//2) == -1)
        assert(gamma**(params.n) == -1)
    else:
        omega, gamma = None, None

    W = intt_matrix(params.n, params.q, omega=omega, gamma=gamma)
    inv_W = ntt_matrix(params.n, params.q, omega=omega, gamma=gamma)

    D = lambda eta: sum(-ZZ.random_element(0, 2) + ZZ.random_element(0, 2) for _ in range(eta))  # noqa
    s = vector(ZZ, [D(params.eta) for i in range(params.n)])
    s_hat = inv_W * s
    d = bitflipf(params.n, params.q, params.kappa, vector(ZZ, s_hat))
    c = W * d + s
    return (W, c), (d, s)



"Preprocessing handles the combinatorial aspect of the attack."


class PreprocParams(object):
    def __init__(self, q=7681, ell=None, alpha=0, beta=1, b=None, shave_first=False, fold=3, even=True):
        """

        :param q: Integer modulus, used to derive `ℓ` if that is not provided
        :param ell: We write elements in base `2^ℓ`
        :param alpha: We guess $\alpha$ bits in the `β` top-most bits
        :param beta: We guess $\alpha$ bits in the `β` top-most bits
        :param b: The highest power of 2^ℓ we consider is (b-1)
        :param shave_first: shave before or after folding (after seems better)
        :param fold: number of folding levels

        """
        self.ell = PreprocParams.ellf(q, ell=ell)
        self.q = q
        self.b = PreprocParams.bf(q, ell=ell, b=b)
        self.alpha = alpha
        self.beta = beta
        self.shave_first = shave_first
        self.fold = fold
        self.even = even

    @staticmethod
    def ellf(q, ell=None):
        """
        By default `2^ℓ ≈ sqrt(q)`
        """
        if ell is None:
            ell = ceil(log(q, 2.)/2.)
        return ell

    @staticmethod
    def bf(q, ell=None, b=None):
        """Return the length of writing `q` in base `2^ℓ`

        :param q: an integer modulus
        :param ell: the vector will be rewritten in base `2^ℓ`
        :param b: if not ``None`` this value is simply returned, i.e. this function computes nothing.

        """
        ell = PreprocParams.ellf(q, ell)

        if b is not None:
            return b
        else:
            return ceil(log(q, 2)/ell)


def shave(v, params):
    """
    Remove the $\alpha$ most significant bits from `v`. This is as though successfully guessing the $\alpha$
    biggest parts of `v` lying in a band of the top $\beta$ bits.

    .. note :: If ``even`` is set in params, this function first ensures that △_i % 2 ≡ 0

    :param v: a vector over the integers
    :param params: preprocessing parameters

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: d = bitflipf(n=16, q=7681, kappa=10); d
        (4095, 4096, 0, 4, -4096, 0, 0, -544, 0, 0, 0, 0, 4096, 0, 0, -511)
        sage: snort(d, 7681)
        (-1, 32, 0, 32, 0, 0, 4, 0, 0, -32, 0, 0, 0, 0, -32, -4, 0, 0, 0, 0, 0, 0, 0, 0, 32, 0, 0, 0, 0, 1, -4)

        sage: pp = PreprocParams(q=7681, alpha=2, beta=2)
        sage: d_, _ = shave(d, pp); d_
        (4096, 4096, 0, 4, -4096, 0, 0, -544, 0, 0, 0, 0, 4096, 0, 0, -512)

        sage: snort(d_, 7681)
        (0, 32, 0, 32, 0, 0, 4, 0, 0, -32, 0, 0, 0, 0, -32, -4, 0, 0, 0, 0, 0, 0, 0, 0, 32, 0, 0, 0, 0, 0, -4)

        sage: snort(d, 7681) - snort(d_, 7681)
        (-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0)

        sage: d_, o_ = shave(d, pp)
        sage: d == d_ + o_
        True
```

```
    We contrast the behaviour above with that of ``even==False``

        sage: pp = PreprocParams(q=7681, alpha=2, beta=2, even=False)
        sage: d_, _ = shave(d, pp); d_
        (-1, 0, 0, 4, -4096, 0, 0, -544, 0, 0, 0, 0, 4096, 0, 0, -511)

        sage: snort(d_, 7681)
        (-1, 0, 0, 0, 0, 0, 4, 0, 0, -32, 0, 0, 0, 0, -32, -4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 32, 0, 0, 0, 0, 1, -4)

        sage: snort(d, 7681) - snort(d_, 7681)
        (0, 32, 0, 32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

        sage: d_, o_ = shave(d, pp)
        sage: d == d_ + o_
        True

    """
    v = snort(v, params.q, ell=params.ell, b=params.b)
    d = vector(ZZ, len(v))

    fixed = 0

    if params.even:
        for i in range(params.alpha):
            for j, s_ in enumerate(v):
                if s_ % 2 and j % 2 == 0:
                    break
            else:
                break

            k = 0
            fixed += 1

            if v[j] > 0:
                d[j] += 2**k
                v[j] -= 2**k
            else:
                d[j] -= 2**k
                v[j] += 2**k

    for i in range(params.alpha-fixed):
        vv = map(lambda x: bsdr_from_int(x, params.ell)[0], v)

        j, k = 0, 0
        for j_ in range(len(vv)):
            for k_ in range(k, params.ell)[::-1]:
                if vv[j_][k_] and k_ > k:
                    j, k = j_, k_

        if k < params.ell-params.beta:
            break

        d[j] += vv[j][k]*2**k
        v[j] -= vv[j][k]*2**k

    s = sneeze(v, params.q, ell=params.ell, b=params.b)
    d = sneeze(d, params.q, ell=params.ell, b=params.b)
    return s, d


def shave_cost(ip, pp):
    """
    The cost of shaving.

    :param ip: instance parameters
    :param pp: preprocessing parameters

    EXAMPLE:

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: ip = InstanceParams(n=256, q=7681, eta=4, kappa=19)
        sage: ceil(shave_cost(ip, PreprocParams(q=7681, alpha=5, beta=2, fold=3)).log(2).n())
        34

        sage: ceil(shave_cost(ip, PreprocParams(q=7681, alpha=5, beta=2, fold=2)).log(2).n())
        39

        sage: ceil(shave_cost(ip, PreprocParams(q=7681, alpha=10, beta=2, fold=2)).log(2).n())
        68

        sage: ceil(shave_cost(ip, PreprocParams(q=7681, alpha=5, beta=3, fold=2)).log(2).n())
        41

        sage: ceil(shave_cost(ip, PreprocParams(q=previous_prime(2^14), alpha=5, beta=2, fold=2)).log(2).n())
        40

    """

    lq = ceil(log(pp.q, 2))
    if pp.shave_first:
        n = ip.n
        fct = 1
    else:
        n = ip.n//2**pp.fold
        fct = 2

    if lq == pp.b*pp.ell:
        nbits = n * (pp.b * pp.beta)
    else:
        diff = pp.b*pp.ell - lq
        nbits = n * (pp.b-1) * pp.beta + n * (pp.beta-diff)

    nbits = max(nbits, 0)  # for beta = 0 the above can be negative
```

```
    if pp.even:
        nbits += n  # We need to handle the n lowest order bits
    # We get an additional factor of 2^i because we also need to guess signs
    return sum(fct*i * binomial(nbits, i) for i in range(pp.alpha+1))


def preprocess(pk, sk, params):
    """

    Apply all preprocessing (folding & shaving)

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: ip = InstanceParams(n=16, q=7681, eta=4, kappa=10)
        sage: (W,c), (d,s) = instance(ip, seed=0)

        sage: (W_, c_), (d_, s_) = preprocess((W,c), (d,s), PreprocParams(q=7681, fold=1))
        sage: c_ == W_*d_ + s_
        True

        sage: (W_, c_), (d_, s_) = preprocess((W,c), (d,s), PreprocParams(q=7681, fold=2))
        sage: c_ == W_*d_ + s_
        True

        sage: (W_, c_), (d_, s_) = preprocess((W,c), (d,s), PreprocParams(q=7681, alpha=1, beta=4))
        sage: c_ == W_*d_ + s_
        True

        sage: (W_, c_), (d_, s_) = preprocess((W,c), (d,s), PreprocParams(q=7681, alpha=1, beta=4, shave_first=True))
        sage: c_ == W_*d_ + s_
        True

    """

    W, c = pk
    d, s = sk

    assert c == W*d + s
    assert ~W*c == d + ~W*s

    q = W.base_ring().order()

    # We do all computations on the forward NTT, thus we flip everything around

    W, c = ~W, ~W*c
    d, s = s, d

    if params.shave_first and params.alpha:
        s, correction = shave(s, params=params)
        c -= correction

    assert c == W*d + s

    for _ in range(params.fold):
        n = len(c)
        W_, c_, d_, s_ = [], [], [], []
        for i in range(n/2):
            W_.append(W[i][0:n:2] + W[i+n/2][0:n:2])
            c_.append(c[i] + c[i+n/2])
            s_.append(s[i] + s[i+n/2])
            d_.append(d[2*i])
        c = vector(c_)
        W = matrix(W_)
        d = balance(vector(d_), q)
        s = vector(s_)

        assert c == W*d + s

    if not params.shave_first and params.alpha:
        s, correction = shave(s, params=params)
        c -= correction

    # We do all computations on the inverse NTT, thus we flip everything around

    W, c = ~W, ~W*c
    d, s = s, d

    return (W, c), (d, s)


def challenge(ip, pp, seed=None):
    """
    A challenge for the attack.

    :param ip: instance parameters
    :param pp: processing parameters
    :param seed: random seed

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: ip = InstanceParams(n=16, q=7681, eta=4, kappa=10)
        sage: pp = PreprocParams(q=7681, fold=1, alpha=4, beta=3)
        sage: (W,c), (d,s) = challenge(ip, pp, seed=1)
        sage: W*d + s == c
        True

    """
    pk, sk = instance(ip, seed=seed)
    pk, sk = preprocess(pk, sk, pp)
    return pk, sk
```

```
def lattice_basis((A, c), params, theta=1):
    """
    Construct a primal attack uSVP lattice.

    :param A: LWE-like matrix
    :param c: LWE-like vector
    :param params: preprocessing parameters
    :param theta: scale the columns of the secret by this factor

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: ip = InstanceParams(n=8, q=7681, eta=4, kappa=10)
        sage: pp = PreprocParams(q=7681, fold=1, alpha=4, beta=3, even=False)
        sage: (W,c), (d,s) = challenge(ip, pp, seed=1)
        sage: lattice_basis((W, c), pp, theta=1)
        [   1    0    0    0    0    0    0    0 6721 3121 1383 4649    0]
        [   0    1    0    0    0    0    0    0   16   76  361 3635    0]
        [   0    0    1    0    0    0    0    0 6721 4649 6298 3121    0]
        [   0    0    0    1    0    0    0    0   16 3635 7320   76    0]
        [   0    0    0    0    1    0    0    0 6721 4560 1383 3032    0]
        [   0    0    0    0    0    1    0    0   16 7605  361 4046    0]
        [   0    0    0    0    0    0    1    0 6721 3032 6298 4560    0]
        [   0    0    0    0    0    0    0    1   16 4046 7320 7605    0]
        [   0    0    0    0    0    0    0    0 7681    0    0    0    0]
        [   0    0    0    0    0    0    0    0    0 7681    0    0    0]
        [   0    0    0    0    0    0    0    0    0    0 7681    0    0]
        [   0    0    0    0    0    0    0    0    0    0    0 7681    0]
        [   0    0    0    0    0    0    0    0 4977 4668 5607 1754   -1]

        sage: lattice_basis((W, c), pp, theta=2)
        [    1     0     0     0     0     0     0     0 13442  6242  2766  9298     0]
        [    0     1     0     0     0     0     0     0    32   152   722  7270     0]
        [    0     0     1     0     0     0     0     0 13442  9298 12596  6242     0]
        [    0     0     0     1     0     0     0     0    32  7270 14640   152     0]
        [    0     0     0     0     1     0     0     0 13442  9120  2766  6064     0]
        [    0     0     0     0     0     1     0     0    32 15210   722  8092     0]
        [    0     0     0     0     0     0     1     0 13442  6064 12596  9120     0]
        [    0     0     0     0     0     0     0     1    32  8092 14640 15210     0]
        [    0     0     0     0     0     0     0     0 15362     0     0     0     0]
        [    0     0     0     0     0     0     0     0     0 15362     0     0     0]
        [    0     0     0     0     0     0     0     0     0     0 15362     0     0]
        [    0     0     0     0     0     0     0     0     0     0     0 15362     0]
        [    0     0     0     0     0     0     0     0  9954  9336 11214  3508    -1]

        sage: pp = PreprocParams(q=7681, fold=1, alpha=4, beta=3, even=True)
        sage: lattice_basis((W, c), pp, theta=1)
        [   2    0    0    0    0    0    0    0 5761 6242 2766 1617    0]
        [   0    1    0    0    0    0    0    0   16   76  361 3635    0]
        [   0    0    2    0    0    0    0    0 5761 1617 4915 6242    0]
        [   0    0    0    1    0    0    0    0   16 3635 7320   76    0]
        [   0    0    0    0    2    0    0    0 5761 1439 2766 6064    0]
        [   0    0    0    0    0    1    0    0   16 7605  361 4046    0]
        [   0    0    0    0    0    0    2    0 5761 6064 4915 1439    0]
        [   0    0    0    0    0    0    0    1   16 4046 7320 7605    0]
        [   0    0    0    0    0    0    0    0 7681    0    0    0    0]
        [   0    0    0    0    0    0    0    0    0 7681    0    0    0]
        [   0    0    0    0    0    0    0    0    0    0 7681    0    0]
        [   0    0    0    0    0    0    0    0    0    0    0 7681    0]
        [   0    0    0    0    0    0    0    0 4977 4668 5607 1754   -1]

        sage: lattice_basis((W, c), pp, theta=2)
        [    2     0     0     0     0     0     0     0 11522 12484  5532  3234     0]
        [    0     1     0     0     0     0     0     0    32   152   722  7270     0]
        [    0     0     2     0     0     0     0     0 11522  3234  9830 12484     0]
        [    0     0     0     1     0     0     0     0    32  7270 14640   152     0]
        [    0     0     0     0     2     0     0     0 11522  2878  5532 12128     0]
        [    0     0     0     0     0     1     0     0    32 15210   722  8092     0]
        [    0     0     0     0     0     0     2     0 11522 12128  9830  2878     0]
        [    0     0     0     0     0     0     0     1    32  8092 14640 15210     0]
        [    0     0     0     0     0     0     0     0 15362     0     0     0     0]
        [    0     0     0     0     0     0     0     0     0 15362     0     0     0]
        [    0     0     0     0     0     0     0     0     0     0 15362     0     0]
        [    0     0     0     0     0     0     0     0     0     0     0 15362     0]
        [    0     0     0     0     0     0     0     0  9954  9336 11214  3508    -1]

    """

    m = A.nrows()
    K = A.base_ring()
    q = K.order()
    n = A.ncols()
    d = m + n*params.b + 1
    num = QQ(theta).numerator()
    den = QQ(theta).denominator()

    L = matrix(ZZ, d, d)

    b = params.b

    for i in range(n):
        for j in range(b):
            row = 2**(j*params.ell) * A.T[i]
            L[i*b+j, i*b+j] = den
            for k in range(m):
                L[i*b+j, n*b + k] = num*ZZ(row[k])

    for i in range(m):
        L[n*b + i, n*b + i] = num*q
        L[-1, n*b + i] = num*ZZ(c[i])
```

```
    L[-1, -1] = -1

    if params.even:
        for i in range(0, 2*n, 2):
            for j in range(3*n):
                L[i, j] = (2*L[i, j]) % (num*q)

    return L


def bitflip_norm_expectation(ip, pp, theta=1):
    """
    Return expectation of squared norm of $\Delta^{(\ell)}$

    :param ip: instance parameters
    :param pp: processing parameters
    :param theta: lattice scaling factor

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: ip = InstanceParams(n=256, q=7681, eta=4, kappa=19)
        sage: bitflip_norm_expectation(ip, PreprocParams(q=7681, fold=3, alpha=6, beta=1, even=False), theta=1)
        2536
        sage: bitflip_norm_expectation(ip, PreprocParams(q=7681, fold=3, alpha=6, beta=1, even=False), theta=2)
        2920

    """
    ell = ZZ(pp.ell)
    beta = ZZ(pp.beta)

    if log(ip.q, 2) == pp.ell * pp.b:
        lhs = (ell - beta)/ell * ip.kappa * ((4**(ell - beta) - 1)/RR(ell - beta)/RR(3))
    else:
        lhs = (ell - beta)/ell * ip.kappa/2. * ((4**(ell - beta) - 1)/RR(ell - beta)/RR(3))
        lhs += max((ell - beta)/ell * ip.kappa/2. * ((4**(ell - beta - 1) - 1)/RR(ell - beta - 1)/RR(3)), 0)
    rhs = ip.n//2**pp.fold * theta**2 * ip.sd**2
    return ceil(lhs + rhs)


@cached_function
def bitflip_norm_estimate(ip, pp, theta=1, ntrials=1024, return_all=False):
    """
    Return median of squared norm of $\Delta^{(\ell)}$

    :param ip: instance parameters
    :param pp: processing parameters
    :param theta: lattice scaling factor
    :param ntrials: sample
    :param return_all: return all samples instead of the median

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: ip = InstanceParams(n=256, q=7681, eta=4, kappa=19)
        sage: pp = PreprocParams(q=7681, fold=3, alpha=6, beta=1, even=False)
        sage: bitflip_norm_estimate(ip, pp, ntrials=512, theta=1)
        3588
        sage: bitflip_norm_estimate(ip, pp, ntrials=512, theta=2)
        4119
        sage: pp = PreprocParams(q=7681, fold=3, alpha=6, beta=1, even=True)
        sage: bitflip_norm_estimate(ip, pp, ntrials=512, theta=1)
        3761
        sage: bitflip_norm_estimate(ip, pp, ntrials=512, theta=2)
        4158

    """
    t = []

    q = ip.q
    n = ZZ(ip.n//2**pp.fold)

    for _ in range(ntrials):
        d = bitflipf(ip.n, ip.q, ip.kappa)

        if pp.shave_first and pp.alpha:
            d, _ = shave(d, params=pp)

        for _ in range(pp.fold):
            d = balance(d[:len(d)/2] + d[len(d)/2:], q)

        if not pp.shave_first and pp.alpha:
            d, _ = shave(d, params=pp)

        d = snort(d, ip.q, pp.ell, b=pp.b)
        t.append(d.norm()**2 + n * (theta*ip.sd)**2 + 1)

    if return_all:
        return tuple(t)
    else:
        return ceil(median(t))


def enumeration_parameters(B, ip, pp, theta=1):
    from fpylll import IntegerMatrix, GSO, Pruning
    from fpylll.fplll.pruner import svp_probability

    B = IntegerMatrix.from_matrix(B)
    M = GSO.Mat(B)
    M.update_gso()

    t = bitflip_norm_estimate(ip, pp, theta=theta)
```

```
    # HACK: block size 60 is somewhat arbitrary, for small lattices we pick something smaller
    bs = min(60, ceil(M.d*0.75))
    # there no point enumerating beyond the length of the shortest vector in the projected sublattice
    target_norm = min(float(bs)/float(M.d) * t, M.get_r(M.d-bs, M.d-bs))
    pruner = Pruning.Pruner(target_norm, 2**40, [M.r(M.d-bs)], 0.9, flags=Pruning.ZEALOUS|Pruning.CVP)
    coefficients = pruner.optimize_coefficients([1. for _ in range(bs)])
    probability = svp_probability(coefficients)
    cost = pruner.single_enum_cost(coefficients)

    return target_norm, coefficients, cost, probability


def predict_attack((d, s), B, ip, pp, theta=1, verbose=False):
    """
    We need to satisfy two conditions (assuming ``even=True``):

    1. Our target ``t`` must be shorter than the shortest vector in the basis
    2. Our target ``t`` must be shorter than (-2^ℓ, 1)

    .. note:: All norms are squared

    """
    shave_cost_ = shave_cost(ip, pp)

    q = ip.q
    ell = pp.ell

    num = QQ(theta).numerator()
    den = QQ(theta).denominator()
    b = pp.b  # number of components in base 2^ℓ
    ell = pp.ell

    actual = snort(d, q, ell=ell, b=b).norm()**2 + (num*s).norm()**2 + 1

    b1 = B[0].norm()**2
    b2 = den**2 * (2**(2*ell) + 1)
    # b3 = (den*vector(ZZ(q).digits(2**ell))).norm()**2
    t = bitflip_norm_estimate(ip, pp, theta=theta)

    b = min([b1, b2])

    target_norm, coefficients, enum_cost, enum_prob = enumeration_parameters(B, ip, pp, theta=theta)

    if verbose:
        fmt =  u"θ: %d, α: %d, β: %d "
        fmt += u"| |Δ|: %7.1f, B[0]: %7.1f, 2^ℓ+1: %7.1f, |Δ|/B[0]: %7.3f, |this|: %7.1f, "
        fmt += u"#enum: 2^%.1f, Pr: %5.1f%%, shave_cost: %5.1f" # noqa
        print fmt%(theta, pp.alpha, pp.beta, t, b1, b2, t/b, actual,
                   log(enum_cost, 2), 100*enum_prob, log(shave_cost_, 2))

    return target_norm, coefficients



"Lattice Reduction"


def progressive_lattice_reduction(B, max_block_size=90, verbose=True, fn=None):
    """
    Run progressive-BKZ with blocksizes increasing by five in each iteration.

    :param B: lattice basis
    :param max_block_size: maximum block size to consider
    :param verbose: print progress information
    :param fn: save the (intermediate) reduced basis to this file

    """
    from fpylll import IntegerMatrix
    from fpylll.algorithms.bkz2 import BKZReduction as BKZ2

    B = IntegerMatrix.from_matrix(B)
    d = min(B.nrows, max_block_size)
    bkz = BKZ2(B)

    block_sizes = tuple(range(20, d, 5)) + (d,)

    for block_size in block_sizes:
        bkz(BKZ.Param(block_size=block_size, strategies=BKZ.DEFAULT_STRATEGY,
                      flags=(BKZ.VERBOSE if verbose else 0)|BKZ.AUTO_ABORT))
        if fn:
            B.to_matrix(matrix(ZZ, B.nrows, B.ncols)).save(fn)
    return B.to_matrix(matrix(ZZ, B.nrows, B.ncols))


def offset_vector(A, t, target_norm, pruning_coefficients):
    """
    Compute the (hopefully shortest) vector Λ(A)-t

    :param A:
    :param t:
    :param target_norm:
    :param pruning_coefficients:

    """

    from fpylll import IntegerMatrix, GSO, Enumeration, EnumerationError

    A = IntegerMatrix.from_matrix(A)
    M = GSO.Mat(A)
    M.update_gso()

    tt = M.from_canonical(t)

    try:
```

```
        block_size = len(pruning_coefficients)
        i = M.d-block_size
        enum = Enumeration(M)
        enum_sol = enum.enumerate(i, M.d,
                                  target_norm, 1,
                                  target=tt,
                                  pruning=pruning_coefficients)
        enum_sol = map(int, enum_sol[0][1])
        # take out what we got so far
        t -= vector(A.multiply_left(enum_sol, start=i))

        # handle the rest using Babai's nearest plane
        babai_sol = M.babai(t, dimension=M.d-i)
        return vector(A.multiply_left(babai_sol)) - t
    except EnumerationError:
        return t


def parse_short_vector(v, ip, pp, theta=1):
    """
    Read off the solution from the short vector

    :param v:
    :param ip:
    :param pp:
    :param theta:

    """
    num = QQ(theta).numerator()
    den = QQ(theta).denominator()

    sv = vector(ZZ, v)
    ell = pp.ell
    b = pp.b
    n = ip.n//2**pp.fold
    s = vector(ZZ, [-1*sv[i]//den for i in range(b*n)])
    e = vector(ZZ, [sv[i+b*n]//num for i in range(n)])

    s = sneeze(s, ip.q, ell=ell, b=b)
    return (s, e)



"Kyber instances"


def _kyber_lattice_basis((n, q, theta, even)):
    """
    Construct and save a reduced lattice basis for an NTT instance derived from Kyber parameters.
    """

    if q == 12289:
        N = 1024
    else:
        N = 256

    fold = log(N/n, 2)
    ip = InstanceParams(n=N, q=q, eta=4, kappa=20)
    pp = PreprocParams(q=q, b=2, fold=fold, alpha=0, beta=0, even=even)

    pk, sk = challenge(ip, pp, seed=0)
    W, c = pk
    B = lattice_basis((W, c), pp, theta=theta)
    B, _ = B.submatrix(0, 0, B.nrows()-1, B.ncols()-1), B[-1][:-1]  # noqa

    if q == 12289:
        B.save("basesnh/%s/B-%02d-%d.sobj"%("even" if even else "odd", n, theta))
    else:
        B.save("bases/%s/B-%02d-%d.sobj"%("even" if even else "odd", n, theta))

    # HACK We're removing the unusually short vector
    B = B.submatrix(2, 2, B.nrows()-2, B.ncols()-2)

    if q == 12289:
        progressive_lattice_reduction(B, fn="basesnh/%s/R-%02d-%s-fixed.sobj"%("even" if even else "odd", n, theta))
    else:
        progressive_lattice_reduction(B, fn="bases/%s/R-%02d-%s-fixed.sobj"%("even" if even else "odd", n, theta))
    return True


def kyber_lattices_bases(N=(8, 16, 32), THETA=(1, 2, 3, 4, 5, 6, 7, 8), q=7681, even=False, workers=1):
    """
    Construct and save reduced lattices bases for NTT instances derived from Kyber parameters.
    """
    from multiprocessing import Pool
    pool = Pool(workers)
    jobs = [(n, q, theta, even) for n in N for theta in THETA]
    list(pool.imap_unordered(_kyber_lattice_basis, jobs))


def kyber_run_attack(ip, pp, seed=None, theta=1, verbose=False, pubk=None, seck=None):
    """
    Run attack for one challenge

    :param ip: instance parameter
    :param pp: preprocessing parameter
    :param seed: seed for choosing instance
    :param theta: lattice scaling factor
    :param verbose: print characteristics of the attack
    :returns: ratio between the squared norm of shortest vector found and squared norm of the target vector

    EXAMPLE::

        sage: load("poc.py")
```

```
        sage: set_random_seed(1337)
        sage: ip = InstanceParams(n=256, q=7681, eta=4, kappa=10)
        sage: pp = PreprocParams(q=7681, fold=3, alpha=4, beta=1, even=False)
        sage: kyber_run_attack(ip, pp, seed=1, theta=3)[2]
        1.00000000000000

    """
    if pubk is None and seck is None:
        pk, sk = challenge(ip, pp, seed=seed)
        W, c = pk
        d, s = sk
        n_ = ip.n//2**pp.fold
    else:
        W, c = pubk
        d, s = seck
        n_ = W.dimensions()[0]

    # HACK we set △_0 to zero
    c -= W.column(0) * d[0]
    d[0] = 0

    # assert(W*d + s == c)

    B = lattice_basis((W, c), pp, theta=theta)  # We only use this to get t
    B, t = B.submatrix(0, 0, B.nrows()-1, B.ncols()-1), B[-1][:-1]
    if ip.q == 12289:   # newhope bases
        B = load("basesnh/%s/R-%02d-%d-fixed.sobj"%("even" if pp.even else "odd", n_, theta))
    else:   # kyber bases
        B = load("bases/%s/R-%02d-%d-fixed.sobj"%("even" if pp.even else "odd", n_, theta))
    t = t[2:]  # We dropped △_0

    norm, c = predict_attack((d, s), B, ip, pp, theta=theta, verbose=verbose)
    # print "d", d
    # print "snort(d)", snort(d, ip.q, ell=pp.ell, b=pp.b)
    # print "s", s

    # the actual attack
    b = offset_vector(B, t, norm, c)
    d_, s_ = parse_short_vector([0, 0] + list(b), ip, pp, theta=theta)

    # print "d_", d_
    # print "snort(d_)", snort(d_, ip.q, ell=pp.ell, b=pp.b)
    # print "s_", s_

    return d_, s_, RR(s_.norm()**2/s.norm()**2)


def kyber_lattice_challenge(ip, pp, seed=None, theta=1):
    """
    Return lattices/vectors used in attack.

    :param ip: instance parameter
    :param pp: preprocessing parameter
    :param seed: seed for choosing instance
    :param theta: lattice scaling factor
    :param verbose: print characteristics of the attack
    """
    pk, sk = challenge(ip, pp, seed=seed)

    W, c = pk
    d, s = sk

    # HACK we set △_0 to zero
    c -= W.column(0) * d[0]
    d[0] = 0

    assert(W*d + s == c)

    n_ = ip.n//2**pp.fold

    B = lattice_basis((W, c), pp, theta=theta)  # We only use this to get t
    B, t = B.submatrix(0, 0, B.nrows()-1, B.ncols()-1), B[-1][:-1]
    B = load("bases/%s/R-%02d-%d-fixed.sobj"%("even" if pp.even else "odd", n_, theta))
    t = t[2:]  # We dropped △_0
    ds = vector(tuple(snort(d, ip.q, ell=pp.ell, b=pp.b)) + tuple(theta*s))

    return B, t, ds[2:]


def kyber_test_attack((theta, alpha, beta), ntrials=8, kappa=19, newhope=False, verbose=True, **kwds):
    """
    Test attack performance.

    EXAMPLE::

        sage: load("poc.py")
        sage: set_random_seed(1337)
        sage: kyber_test_attack((3, 4, 2))
        θ: 3, α: 4, β: 2 | |△|:  4328.0, B[0]:  7528.0, 2^ℓ+1: 16385.0, |△|/B[0]:   ...
        (0.75, 173451777, 19251)

    :param theta, alpha, beta: parameters of the attack
    :param ntrials: number of experiments to run
    :param fold: level of folding
    :param kappa: number of bit flips
    :param newhope: use newhope parameters
    :param verbose: print more intermediate information
    :param even: use even variant of the attack

    """
    if newhope:
        n, q, eta, fold = 1024, 12289, 16, 5
    else:
        n, q, eta, fold = 256, 7681, 4, 3
```

```
ip = InstanceParams(n=n, q=q, eta=eta, kappa=kappa)
pp = PreprocParams(q=q, alpha=alpha, beta=beta, fold=fold, **kwds)

assert(kappa > alpha)

if newhope:
    B = load("basesnh/%s/R-%02d-%d-fixed.sobj"%("even" if pp.even else "odd", ip.n//2**pp.fold, theta))
else:
    B = load("bases/%s/R-%02d-%d-fixed.sobj"%("even" if pp.even else "odd", ip.n//2**pp.fold, theta))
target_norm, coefficients, enum_cost, enum_prob = enumeration_parameters(B, ip, pp, theta=theta)

count = 0

for i in range(ntrials):
    r = kyber_run_attack(ip, pp, seed=i, verbose=(i==0 if verbose else False), theta=theta)[2]
    count += int(r == 1)
return float(count)/ntrials, shave_cost(ip, pp), ceil(enum_cost)
```