CRYSTALS-Dilithium

Algorithm Specifications and Supporting Documentation (Version 3.1)

Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler and Damien Stehlé

[1] Florida Atlantic University
[2] CWI, The Netherlands
[3] Ruhr Universität Bochum, Germany
[4] Google, USA
[5] IBM Research Europe, Switzerland
[6] Max Planck Institute for Security and Privacy, Germany & Radboud University, The Netherlands
[7] IBM Research Europe and ETH, Switzerland
[8] ENS de Lyon, France

February 8, 2021

The most up-to-date versions of our specification document and code can be accessed at:

https://pq-crystals.org/dilithium/
and
https://github.com/pq-crystals/dilithium/tree/round3

Changelog (Round 1 → Round 2)

We first describe the differences between the round 1 submission of Dilithium and the current version. The only conceptual design change that we made is to allow the option for the scheme to be either randomized or deterministic (whereas the round 1 version was only deterministic). The difference between the two versions of the scheme is one line in which a seed is chosen at random (in the randomized version) or as a hash of a key together with the message (in the deterministic version). To allow for this small difference between the deterministic and randomized version, the function ExpandMask now takes a 48 byte seed as input instead of a key and the message. As a minor additional change we have harmonized the nonces of the various expansion functions for the matrix $\mathbf{A}$, the masking vector $\mathbf{y}$ and the secret vectors $\mathbf{s}_1$, $\mathbf{s}_2$ to all be 16-bit integers.

The other changes were in the implementation. We made various optimizations in the signing algorithm. The most important optimization is how the rejection condition based on the low part of the vector $\mathbf{w}$ and the hint vector is computed. Our AVX2 optimized implementation now makes more use of vectorization and includes a simpler assembler NTT implementation using macros.

We also introduced a version of the algorithm that uses AES, rather than SHAKE to expand the public matrix $\mathbf{A}$, the secret vectors $\mathbf{s}_1$, $\mathbf{s}_2$ and the masking vector $\mathbf{y}$ from a seed. This is mainly done to showcase the improved efficiency of our scheme once hardware-support for SHAKE becomes widely available (as it is for AES).

Changelog (Round 2 → Round 3)

- Combined the first two parameter sets (which used 4×3 and 5×4 public key matrices) into one NIST level 2 set which has a 4×4 public key matrix. We also introduced a level 5 parameter set.
- Instead of always outputting a challenge polynomial with 60 non-zero coefficients (and thus having 257-bit entropy), we now output ones with 39 and 49, which have 192 and 225 bits of entropy, for security levels 2 and 3, respectively. This entropy is enough for the respected security levels (because only second-preimage resistance is needed in the proof) and has the effect of somewhat decreasing the rejection probability.
- Decreased the number of dropped bits from the public key $\mathbf{t}$ from 14 to 13, which has the effect of increasing the SIS hardness of the scheme at the expense of slightly increasing the public key size.
- The masking polynomial $\mathbf{y}$ is now sampled from a range where each coefficient has a power-of-2 number of possibilities, making the sampling noticeably simpler to implement. Because the previous range was already very close to a power-of-2, this change has no effect on the security of the scheme.
- Changed the way the "challenge" c is sampled and transmitted. It is now sampled in two stages. In the first, a hash outputting a 32-byte challenge seed $\tilde{c}$ is created by the prover, and this is what is included in the signature. The polynomial challenge used in the signature scheme is then created using SHAKE with $\tilde{c}$ as the seed. This saves 8 bytes in the signature size.
- Updated the concrete security analysis, providing refined estimates in the gate cost metric, based on recent progress on predicting the behavior and the costs of lattice algorithms.
- Made some miscellaneous small changes to the NIST level 3 parameter set.

1 Introduction

We present the digital signature scheme Dilithium, whose security is based on the hardness of finding short vectors in lattices. Our scheme was designed with the following criteria in mind:

Simple to implement securely. The most compact lattice-based signature schemes [DDLL13, DLP14] crucially require the generation of secret randomness from the discrete Gaussian distribution. Generating such samples in a way that is secure against side-channel attacks is highly non-trivial and can easily lead to insecure implementations, as demonstrated in [BHLY16, EFGT17, PBY17]. While it may be possible that a very careful implementation can prevent such attacks, it is unreasonable to assume that a universally-deployed scheme containing many subtleties will always be expertly implemented. Dilithium therefore only uses uniform sampling, as was originally proposed for signatures in [Lyu09, GLP12]. Furthermore, all other operations (such as polynomial multiplication and rounding) are easily implemented in constant time.

Be conservative with parameters. Since we are aiming for long-term security, we have analyzed the applicability of lattice attacks from a very favorable, to the attacker, viewpoint. In particular, we are considering (quantum) algorithms whose space requirements are on the same order as the time ones. Such algorithms are currently unrealistic, and there seem to be serious obstacles in removing the space requirement, but we are allowing for the possibility that improvements may occur in the future.

Minimize the size of public key + signature. Since many applications require the transmission of both the public key and the signature (e.g. certificate chains), we designed our scheme to minimize the sum of these parameters. Under the restriction that we avoid (discrete) Gaussian sampling, to the best of our knowledge, Dilithium has the smallest combination of signature and public key sizes of any lattice-based scheme with the same security levels.

Be modular – easy to vary security. The two operations that constitute nearly the entirety of the signing and verification procedures are expansion of an XOF (we use SHAKE-128 and SHAKE-256), and multiplication in the polynomial ring $\mathbb{Z}_q[X]/(X^n+1)$. Highly efficient implementations of our algorithm will therefore need to optimize these operations and make sure that they run in constant time. For all security levels, our scheme uses the same ring with $q = 2^{23} - 2^{13} + 1$ and $n = 256$. Varying security simply involves doing more/less operations over this ring and doing more/less expansion of the XOF. In other words, once an optimized implementation is obtained for some security level, it is easy to obtain an optimized implementation for a higher/lower level.

1.1 Overview of the Basic Approach

The design of the scheme is based on the "Fiat-Shamir with Aborts" approach [Lyu09, Lyu12] and bears most resemblance to the schemes proposed in [GLP12, BG14]. For readers who are unfamiliar with the general framework of such signature schemes, we present a simplified (and less efficient) version of our scheme in Fig. 1. This version is a slightly modified version of the scheme from [BG14]. We will now go through each of its components to give the reader an idea of how such schemes work.

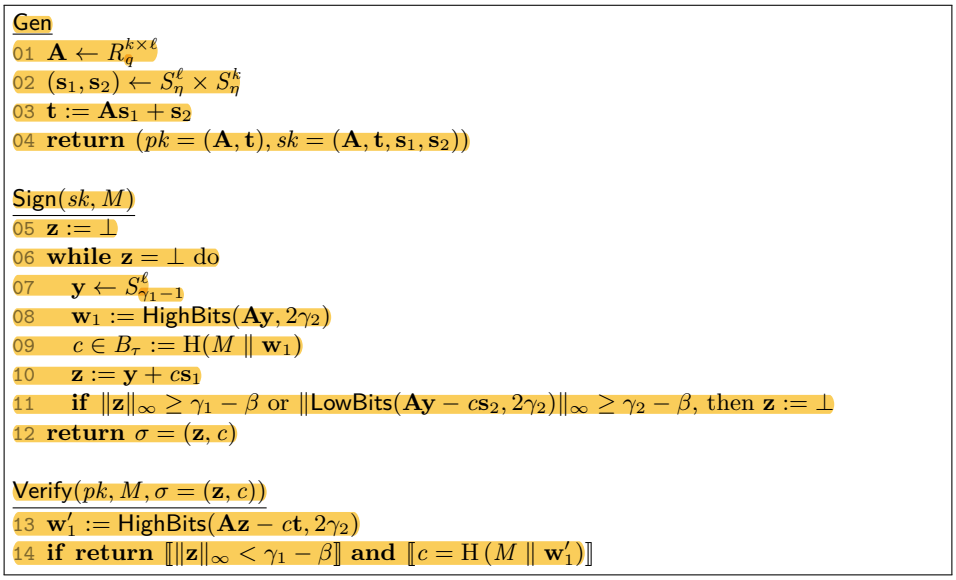

Figure 1: Template for our signature scheme without public key compression.

Key Generation. The key generation algorithm generates a $k \times \ell$ matrix $\mathbf{A}$ each of whose entries is a polynomial in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. As previously mentioned, we will always have $q = 2^{23} - 2^{13} + 1$ and $n = 256$. Afterwards, the algorithm samples random secret key vectors $\mathbf{s}_1$ and $\mathbf{s}_2$. Each coefficient of these vectors is an element of R_q with small coefficients of size at most η. Finally, the second part of the public key is computed as $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$. All algebraic operations in this scheme are assumed to be over the polynomial ring R_q.

Signing Procedure. The signing algorithm generates a masking vector of polynomials $\mathbf{y}$ with coefficients less than γ_1. The parameter γ_1 is set strategically – it is large enough that the eventual signature does not reveal the secret key (i.e. the signing algorithm is zero-knowledge), yet small enough so that the signature is not easily forged. The signer then computes $\mathbf{A}\mathbf{y}$ and sets $\mathbf{w}_1$ to be the "high-order" bits of the coefficients in this vector. In particular, every coefficient w in $\mathbf{A}\mathbf{y}$ can be written in a canonical way as $w = w_1 \cdot 2\gamma_2 + w_0$ where $|w_0| \leq \gamma_2$; $\mathbf{w}_1$ is then the vector comprising all the w_1's. The challenge c is then created as the hash of the message and $\mathbf{w}_1$. The output c is a polynomial in R_q with exactly τ ± 1's and the rest 0's. The reason for this distribution is that c has small norm and comes from a domain of size $\log_2 \binom{256}{\tau} + \tau$, which we would like to be between 128 and 256. The potential signature is then computed as $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$.

If $\mathbf{z}$ were directly output at this point, then the signature scheme would be insecure due to the fact that the secret key would be leaked. To avoid the dependency of $\mathbf{z}$ on the secret key, we use rejection sampling. The parameter β is set to be the maximum possible coefficient of $c\mathbf{s}_i$. Since c has τ ± 1's and the maximum coefficient in $\mathbf{s}_i$ is η, it's easy to see that $\beta \leq \tau \cdot \eta$. If any coefficient of $\mathbf{z}$ is larger than $\gamma_1 - \beta$, then we reject and restart the signing procedure. Also, if any coefficient of the low-order bits of $\mathbf{A}\mathbf{z} - c\mathbf{t}$ is greater than $\gamma_2 - \beta$, we restart. The first check is necessary for security, while the second is necessary for both security and correctness. The while loop in the signing procedure keeps being repeated until the preceding two conditions are satisfied. The parameters are set such that the expected number of repetitions is not too high (e.g. around 4).

Verification. The verifier first computes $\mathbf{w}_1'$ to be the high-order bits of $\mathbf{Az} - c\mathbf{t}$, and then accepts if all the coefficients of $\mathbf{z}$ are less than $\gamma_1 - \beta$ and if c is the hash of the message and $\mathbf{w}_1'$. Let us look at why verification works, in particular as to why $\mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2) = \mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2)$. The first thing to notice is that $\mathbf{Az} - c\mathbf{t} = \mathbf{Ay} - c\mathbf{s}_2$. So all we really need to show is that

$$\mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2) = \mathsf{HighBits}(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2). \tag{1}$$

The reason for this is that a valid signature will have $\|\mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta$. And since we know that the coefficients of $c\mathbf{s}_2$ are smaller than β, we know that adding $c\mathbf{s}_2$ is not enough to cause any carries by increasing any low-order coefficient to have magnitude at least γ_2. Thus Eq. (1) is true and the signature verifies correctly.

1.2 Dilithium

The basic template in Fig. 1 is rather inefficient, as is. The most glaring (but trivially fixed) inefficiency is that the public key consists of a matrix of $k \cdot \ell$ polynomials, which could have a rather large representation. The fix is simply to have $\mathbf{A}$ generated from some seed ρ using SHAKE-128, and this is a standard technique. The public key is therefore $(\rho, \mathbf{t})$ and its size is dominated by $\mathbf{t}$. The novelty of Dilithium over the previous schemes (e.g. [BG14] and qTESLA [ABB+19], which is a particular instantiation of the [BG14] framework) is that we also shrink the bit-representation size of $\mathbf{t}$ by a factor slightly larger than two at the expense of increasing the signature by less than 100 bytes.

The main observation for obtaining this very favorable trade-off is that when the verifier computes $\mathbf{w}_1'$ in Line 13, the high-order bits of $\mathbf{Az} - c\mathbf{t}$ do not depend too much on the *low order* bits of $\mathbf{t}$ because $\mathbf{t}$ is being multiplied by a very low-weight polynomial c. In our scheme, some low-order bits of $\mathbf{t}$ are not included in the public key, and so the verifier cannot always correctly compute the high-order bits of $\mathbf{Az} - c\mathbf{t}$. To make up for this, the signer includes some "hints" as part of the signature, which are essentially the carries caused by adding in the product of c with the missing low-order bits of $\mathbf{t}$. With this hint, the verifier is able to correctly compute $\mathbf{w}_1'$.

Additionally, we give an option for making our scheme deterministic using the standard technique of adding a seed to the secret key and using this seed together with the message to produce the randomness $\mathbf{y}$ in Line 07. The recent result of Kiltz et al. [KLS18] showed that the fewer different signatures the adversary sees for the same messages, the tighter the reduction is in the quantum random oracle model between the signature scheme and the underlying hardness assumptions. While it's not clear as to whether there is an improved quantum attack for randomized signatures, we suggest the deterministic version as the default option except in scenarios where an adversary can mount side-channel attacks that exploit determinism [SBB+18, PSS+18]. Our full scheme in Fig. 4 also makes use of basic optimizations such as pre-hashing the message M so as to not rehash it with every signing attempt.

Implementation Considerations. The main algebraic operation performed in the scheme is a multiplication of a matrix $\mathbf{A}$, whose elements are polynomials in $\mathbb{Z}_q[X]/(X^{256}+1)$, by a vector $\mathbf{y}$ of such polynomials. In our NIST Level 3 parameter setting, $\mathbf{A}$ is a 6×5 matrix and therefore consists of 30 polynomials. Thus the multiplication $\mathbf{Ay}$ involves 30 polynomial multiplications. As in many lattice-based schemes that are based on operations over polynomial rings, we have chosen our ring so that the multiplication operation has a very efficient implementation via the Number Theoretic Transform (NTT), which is just a version of FFT that works over the finite field $\mathbb{Z}_q$ rather than over the complex numbers. To enable the "fully-splitting" NTT algorithm, we need to choose a prime q so that the group $\mathbb{Z}_q^*$ has an element of order $2n = 512$; or equivalently $q \equiv 1 \pmod{512}$. If

r is such an element, then $X^{256}+1 = (X-r)(X-r^3)\cdots(X-r^{511})$ and thus one can equivalently represent any polynomial $a \in \mathbb{Z}_q[X]/(X^{256}+1)$ in its CRT (Chinese Remainder Theorem) form as $(a(r), a(r^3), \ldots, a(r^{2n-1}))$. The advantage of this representation is that the product of two polynomials is coordinate-wise. Therefore the most expensive parts of polynomial multiplication are the transformations $a \to \hat{a}$ and the inverse $\hat{a} \to a$ – these are the NTT and inverse NTT operations.

The above-mentioned advantage of using the NTT representation is even more pronounced due to the fact that when doing the multiplication $\mathbf{Ay}$, one needs to compute much fewer NTTs than the dimension of the matrix. Since the matrix is uniformly random, one can generate (and store) it already in NTT form. Doing the multiplication of this matrix by a vector $\mathbf{y} \in R_q^5$ then only involves doing 5 NTT computations to convert $\mathbf{y}$ into the NTT representation[1], then doing pointwise multiplications to obtain $\mathbf{Ay}$ in NTT form, followed by 6 inverse NTT multiplications.

The other major time-consuming operation is the expansion of a seed ρ into the polynomial matrix $\mathbf{A}$. The matrix $\mathbf{A}$ is needed for both signing and verification, therefore a good implementation of SHAKE-128 is important for the efficiency of the scheme.

For our AVX2 optimized implementation of the NTT we take the approach of using integer, rather than floating point, arithmetic. Although we pack only 4 coefficients into one vector register of 256 bits, which is the same density that is also used by floating point implementations, we can improve on the multiplication speed by about a factor of 2. We achieved this speed-up by carefully scheduling the instructions and interleaving the multiplications and reductions during the NTT so that parts of the multiplication latencies are hidden.

Security. Following the basic approach from [Lyu12], the security of the signature scheme in Figure 1 can be proved, in the Random Oracle model (ROM), based on the hardness of two problems. The first is the standard LWE (over polynomial rings) problem which asks to distinguish $(\mathbf{A}, \mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2)$ from $(\mathbf{A}, \mathbf{u})$, where $\mathbf{u}$ is uniformly random. The other problem is, what was called the SelfTargetMSIS problem in [KLS18], which is the problem of finding a vector $\begin{bmatrix}\mathbf{z}\\ c\\ \mathbf{v}\end{bmatrix}$ with small coefficients and a message (digest) μ satisfying

$$\mathrm{H}\left(\mu \parallel [\ \mathbf{A}\ |\ \mathbf{t}\ |\ \mathbf{I}] \cdot \begin{bmatrix}\mathbf{z}\\ c\\ \mathbf{v}\end{bmatrix}\right) = c,$$

where $\mathbf{A}$ and $\mathbf{t}$ are uniformly random and $\mathbf{I}$ is the identity matrix. In the ROM, one can get a (non-tight) reduction using the forking lemma [PS00, BN06] from the standard MSIS problem of finding a $\mathbf{z}'$ with small coefficients satisfying $\mathbf{A}\mathbf{z}' = 0$ to SelfTargetMSIS. One can follow this exact approach to prove Dilithium secure in the ROM based on the hardness of MLWE and SIS.

In the Quantum Random Oracle model (QROM), where the adversary can query H in superposition, the situation is a little different. It was shown in [KLS18] that Dilithium is still based on MLWE and SelfTargetMSIS in the QROM, even with a tight reduction when the scheme is deterministic. But one can no longer directly use the forking lemma (since it's a type of rewinding) to give a quantum reduction from MSIS to SelfTargetMSIS. There are still good reasons to believe that the SelfTargetMSIS problem, and therefore Dilithium, is secure in the QROM. Firstly, there are no natural signature schemes constructed from Σ-protocols using the Fiat-Shamir transform that are secure in the ROM but not in the QROM. Also, it is possible to set parameters of Dilithium (while leaving the design of the

[1] The vectors, like $\mathbf{y}$, which we multiply $\mathbf{A}$ with contain only coefficients and so cannot be generated directly in NTT form.

scheme the same) so that the SelfTargetMSIS problem becomes information-theoretically hard, thus leaving this version of Dilithium secure in the QROM based on just MLWE. An instantiation of such parameters in [KLS18] results in a scheme with signatures and public keys that are 2X and 5X larger, respectively. While we do not deem this to be a good trade-off, the existence of such a scheme gives us added confidence in the security of the optimized Dilithium.

Very recently, two new works narrowed the gap even more between security in the ROM and the QROM. The work of [DFMS19] showed that if the underlying Σ-protocol is *collapsing* and has special soundness, then its Fiat-Shamir transform is a secure signature in the QROM. Special soundness of the Dilithium Σ-protocol is directly implied by the hardness of MSIS [Lyu12, DKL+18]. Furthermore, [DFMS19] conjecture, that the Dilithium Σ-protocol is collapsing. The work of [LZ19] further showed that the collapsing property does have a reduction from MLWE. The reduction is rather non-tight, but it does give even more affirmation that there is nothing fundamentally insecure about the construction of Dilithium or any natural scheme built via the Fiat-Shamir framework whose security can be proven in the ROM. In our opinion, evidence is certainly mounting that the distinction between signatures secure in the ROM and QROM will soon become treated in the same way as the distinction between schemes secure in the standard model and ROM – there will be some theoretical differences, but security in practice will be the same.

Table 1: Output sizes, security, and performance of Dilithium. The formulas for the sizes of the public key and signature are given in Section 5.4. The numbers in parentheses for SIS security are for the strongly-unforgeable (i.e. hard to come up with a second distinct signature for an already-seen message/signature pair) version of the signature scheme, where the coefficient length comes from Eq. (8), rather than Eq. (7) for the unparenthesized version. Due to the rejection sampling, there is a noticeable difference between the median and average signing times, and so we measure both.

NIST Security Level	2	3	5
Output Size			
public key size (bytes)	1312	1952	2592
signature size (bytes)	2420	3293	4595
LWE Hardness (Core-SVP and refined)			
BKZ block-size b (GSA)	423	624	863
Classical Core-SVP	123	182	252
Quantum Core-SVP	112	165	229
BKZ block-size b (simulation)	433	638	883
$\log_2$ Classical Gates (see App. C.5)	159	217	285
$\log_2$ Classical Memory (see App. C.5)	98	139	187
SIS Hardness (Core-SVP)			
BKZ block-size b	423 (417)	638 (602)	909 (868)
Classical Core-SVP	123 (121)	186 (176)	265 (253)
Quantum Core-SVP	112 (110)	169 (159)	241 (230)
Performance (Unoptimized Reference Code, Skylake)			
Gen median cycles	300,751	544,232	819,475
Sign median cycles	1,081,174	1,713,783	2,383,399
Sign average cycles	1,355,434	2,348,703	2,856,803
Verify median cycles	327,362	522,267	871,609
Performance (AVX2, Skylake)			
Gen median cycles	124,031	256,403	298,050
Sign median cycles	259,172	428,587	538,986
Sign average cycles	333,013	529,106	642,192
Verify median cycles	118,412	179,424	279,936
Performance (AVX2+AES, Skylake)			
Gen median cycles	70,548	153,856	153,936
Sign median cycles	194,892	296,201	344,578
Sign average cycles	251,144	366,470	418,157
Verify median cycles	72,633	102,396	151,066

2 Basic Operations

2.1 Ring Operations

We let R and R_q respectively denote the rings $\mathbb{Z}[X]/(X^n+1)$ and $\mathbb{Z}_q[X]/(X^n+1)$, for q an integer. Throughout this document, the value of n will always be 256 and q will be the prime $8380417 = 2^{23} - 2^{13} + 1$. Regular font letters denote elements in R or R_q (which includes elements in $\mathbb{Z}$ and $\mathbb{Z}_q$) and bold lower-case letters represent column vectors with coefficients in R or R_q. By default, all vectors will be column vectors. Bold upper-case letters are matrices. For a vector $\mathbf{v}$, we denote by $\mathbf{v}^T$ its transpose. The boolean operator $[\![\text{statement}]\!]$ evaluates to 1 if statement is true, and to 0 otherwise.

Modular reductions. For an even (resp. odd) positive integer α, we define $r' = r \bmod^{\pm} \alpha$ to be the unique element r' in the range $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$ (resp. $-\frac{\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$) such that $r' \equiv r \bmod \alpha$. We will sometimes refer to this as a *centered* reduction modulo α.[2] For any positive integer α, we define $r' = r \bmod^{+} \alpha$ to be the unique element r' in the range $0 \leq r' < \alpha$ such that $r' \equiv r \bmod \alpha$. When the exact representation is not important, we simply write $r \bmod \alpha$.

Sizes of elements. For an element $w \in \mathbb{Z}_q$, we write $\|w\|_\infty$ to mean $|w \bmod^{\pm} q|$. We define the ℓ_∞ and ℓ_2 norms for $w = w_0 + w_1 X + \ldots + w_{n-1} X^{n-1} \in R$:

$$\|w\|_\infty = \max_i \|w_i\|_\infty, \quad \|w\| = \sqrt{\|w_0\|_\infty^2 + \ldots + \|w_{n-1}\|_\infty^2}.$$

Similarly, for $\mathbf{w} = (w_1, \ldots, w_k) \in R^k$, we define

$$\|\mathbf{w}\|_\infty = \max_i \|w_i\|_\infty, \quad \|\mathbf{w}\| = \sqrt{\|w_1\|^2 + \ldots + \|w_k\|^2}.$$

We will write S_η to denote all elements $w \in R$ such that $\|w\|_\infty \leq \eta$. We will write $\tilde{S}_\eta$, to denote the set $\{w \bmod^{\pm} 2\eta : w \in R\}$. Note that S_η and $\tilde{S}_\eta$ are very similar except that $\tilde{S}_\eta$ does not contain any polynomials with $-\eta$ coefficients.

2.2 NTT domain representation

Our modulus q is chosen such that there exists a 512-th root of unity r modulo q. Concretely, we always work with $r = 1753$. This implies that the cyclotomic polynomial $X^{256}+1$ splits into linear factors $X - r^i$ modulo q with $i = 1, 3, 5, \ldots, 511$. By the Chinese remainder theorem our cyclotomic ring R_q is thus isomorphic to the product of the rings $\mathbb{Z}_q[X]/(X - r^i) \cong \mathbb{Z}_q$. In this product of rings it is easy to multiply elements since the multiplication is pointwise there. The isomorphism

$$a \mapsto \big(a(r), a(r^3), \ldots, a(r^{511})\big) : R_q \to \prod_i \mathbb{Z}_q[X]/(X - r^i)$$

can be computed quickly with the help of the Fast Fourier Transform. Since $X^{256} + 1 = X^{256} - r^{256} = (X^{128} - r^{128})(X^{128} + r^{128})$ one can first compute the map

$$\mathbb{Z}_q[X]/(X^{256}+1) \to \mathbb{Z}_q[X]/(X^{128} - r^{128}) \times \mathbb{Z}_q[X]/(X^{128} + r^{128})$$

and then continue separately with the two reduced polynomials of degree less than 128 noting that $X^{128} + r^{128} = X^{128} - r^{384}$. The Fast Fourier Transform is also called

[2] We draw the reader's attention to the fact that for even α, the range includes $\alpha/2$ but not $-\alpha/2$. This is a somewhat less standard choice, but defining things in this way makes some parts of the scheme (in particular, the bit-packing of the public key) more efficient.

Number Theory Transform (NTT) in this case where the ground field is a finite field. Natural fast NTT implementations do not output vectors with coefficients in the order $a(r), a(r^3), \ldots, a(r^{511})$. Therefore we define the NTT domain representation $\hat{a} = \mathtt{NTT}(a) \in \mathbb{Z}_q^{256}$ of a polynomial $a \in R_q$ to have coefficients in the order as output by our reference NTT. Concretely,

$$\hat{a} = \mathtt{NTT}(a) = (a(r_0), a(-r_0), \ldots, a(r_{127}), a(-r_{127}))$$

where $r_i = r^{\mathrm{brv}(128+i)}$ with $\mathrm{brv}(k)$ the bitreversal of the 8 bit number k. With this notation, and because of the isomorphism property, we have $ab = \mathtt{NTT}^{-1}(\mathtt{NTT}(a)\mathtt{NTT}(b))$. For vectors $\mathbf{y}$ and matrices $\mathbf{A}$, the representations $\hat{\mathbf{y}} = \mathtt{NTT}(\mathbf{y})$ and $\hat{\mathbf{A}} = \mathtt{NTT}(\mathbf{A})$ mean that every polynomial y_i and $a_{i,j}$ comprising $\mathbf{y}$ and $\mathbf{A}$ is in NTT domain representation. We give further detail about our NTT implementations in Section 5.6.

2.3 Hashing

Our scheme uses several different algorithms that hash strings in $\{0,1\}^*$ onto domains of various forms. Below we give the high level descriptions of these functions and defer the details of how exactly they are used in the signature scheme to Section 5.3.

Hashing to a Ball. Let B_τ denote the set of elements of R that have τ coefficients that are either -1 or 1 and the rest are 0. We have $|B_\tau| = 2^\tau \cdot \binom{256}{\tau}$. For our signature scheme, we will need a cryptographic hash function that hashes onto B_τ in two steps. The first step consists of using a 2^{nd} pre-image resistant cryptographic hash function to map $\{0,1\}^*$ onto the domain $\{0,1\}^{256}$. The second stage consists of an XOF (e.g. SHAKE-256) that maps the output of the first stage to an element of B_τ. The reason for this two-step process is that the output of the first stage allows for a very simple, compact description of an element in B_τ (under the assumption that the XOF is indistinguishable from a purely-random function). The algorithm we will use to create a random element in B_τ is sometimes referred to as an "inside-out" version of the Fisher-Yates shuffle [Knu97], and its high-level description is in Fig. 2.[3] The XOF in the second stage is used to create the randomness needed for this algorithm (in Steps 03 and 04) using the output of the first stage as the seed.

SampleInBall(ρ)
01 Initialize $\mathbf{c} = c_0 c_1 \ldots c_{255} = 00 \ldots 0$
02 **for** $i := 256 - \tau$ to 255
03 $\quad j \leftarrow \{0, 1, \ldots, i\}$
04 $\quad s \leftarrow \{0, 1\}$
05 $\quad c_i := c_j$
06 $\quad c_j := (-1)^s$
07 **return c**

Figure 2: Create a random 256-element array with τ ± 1's and $256 - \tau$ $0's$ using the input seed ρ (and an XOF) to generate the randomness needed in Steps 03 and 04.

Expanding the Matrix A. The function ExpandA maps a uniform seed $\rho \in \{0,1\}^{256}$ to a matrix $\mathbf{A} \in R_q^{k \times l}$ in NTT domain representation. The matrix $\mathbf{A}$ is only needed for multiplication. Hence, for the sake of faster implementations, the expansion function ExpandA does not output $\mathbf{A} \in R_q^{k \times l} = (\mathbb{Z}_q[X]/(X^{256}+1))^{k \times l}$. Instead it outputs $\hat{\mathbf{A}} \in \mathbb{Z}_q^{256}$,

[3] Normally, the algorithm should begin at $i = 0$, but since there are $256 - \tau$ 0's, the first $256 - \tau - 1$ iterations would just be setting components of $\mathbf{c}$ to 0.

which is interpreted as the NTT domain representation of $\mathbf{A}$. As $\mathbf{A}$ needs to be sampled uniformly and the NTT is an isomorphism, $\mathsf{ExpandA}$ also needs to sample uniformly in this representation. To be compatible to Dilithium, an implementation whose NTT produces differently ordered vectors than our reference NTT needs to sample coefficients in a non-consecutive order.

Sampling the vectors $\mathbf{s_1}$ and $\mathbf{s_2}$. The function $\mathsf{ExpandS}$, used for generating the secret vectors in key generation, maps a seed ρ' to $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k$.

Sampling the vectors $\mathbf{y}$. The function $\mathsf{ExpandMask}$, used for deterministically generating the randomness of the signature scheme, maps a seed ρ' and a nonce κ to $\mathbf{y} \in \tilde{S}_{\gamma_1}^l$.

Hashing. The function H used in our signature scheme is the extended output function (XOF) SHAKE-256 mapping onto the specified domain.

2.4 High/Low Order Bits and Hints

To reduce the size of the public key, we will need some simple algorithms that extract "higher-order" and "lower-order" bits of elements in $\mathbb{Z}_q$. The goal is that when given an arbitrary element $r \in \mathbb{Z}_q$ and another small element $z \in \mathbb{Z}_q$, we would like to be able to recover the higher order bits of $r + z$ without needing to store z. We therefore define algorithms that take r, z and produce a 1-bit hint h that allows one to compute the higher order bits of $r + z$ just using r and h. This hint is essentially the "carry" caused by z in the addition.

There are two different ways in which we will break up elements in $\mathbb{Z}_q$ into their "high-order" bits and "low-order" bits. The first algorithm, $\mathsf{Power2Round}_q$, is the straightforward bit-wise way to break up an element $r = r_1 \cdot 2^d + r_0$ where $r_0 = r \bmod^\pm 2^d$ and $r_1 = (r - r_0)/2^d$.

Notice that if we choose the representatives of r_1 to be non-negative integers between 0 and $\lfloor q/2^d \rfloor$, then the distance (modulo q) between any two $r_1 \cdot 2^d$ and $r_1' \cdot 2^d$ is usually $\geq 2^d$, except for the border case. In particular, the distance modulo q between $\lfloor q/2^d \rfloor \cdot 2^d$ and 0 could be very small. This is problematic in the case that we would like to produce a 1-bit hint, as adding a small number to r can actually cause the high-order bits of r to change by more than 1.

We avoid having the high-order bits change by more than 1 with a simple tweak. We select an α that is a divisor of $q - 1$ and write $r = r_1 \cdot \alpha + r_0$ in the same way as before. For the sake of simplicity, we assume that α is even (which is possible, as q is odd). The possible $r_1 \cdot \alpha$'s are now $\{0, \alpha, 2\alpha, \ldots, q - 1\}$. Note that the distance between $q - 1$ and 0 is 1, and so we remove $q - 1$ from the set of possible $r_1 \cdot \alpha$'s, and simply round the corresponding r's to 0. Because $q - 1$ and 0 differ by 1, all this does is possibly increase the magnitude of the remainder r_0 by 1. This procedure is called $\mathsf{Decompose}_q$. Using this procedure as a sub-routine, we can define the $\mathsf{MakeHint}_q$ and $\mathsf{UseHint}_q$ routines that produce a hint and, respectively, use the hint to recover the high-order bits of the sum. For notational convenience, we also define $\mathsf{HighBits}_q$ and $\mathsf{LowBits}_q$ routines that simply extract r_1 and r_0, respectively, from the output of $\mathsf{Decompose}_q$.

The below Lemmas state the properties of these supporting algorithms that are necessary for the correctness and security of our scheme. Their proofs can be found in Appendix A.

Lemma 1. *Suppose that q and α are positive integers satisfying $q > 2\alpha$, $q \equiv 1 \pmod{\alpha}$ and α even. Let $\mathbf{r}$ and $\mathbf{z}$ be vectors of elements in R_q where $\|\mathbf{z}\|_\infty \leq \alpha/2$, and let $\mathbf{h}, \mathbf{h}'$ be vectors of bits. Then the $\mathsf{HighBits}_q$, $\mathsf{MakeHint}_q$, and $\mathsf{UseHint}_q$ algorithms satisfy the following properties:*

$\underline{\mathsf{Power2Round}_q(r, d)}$
08 $r := r \bmod^+ q$
09 $r_0 := r \bmod^\pm 2^d$
10 **return** $\big((r - r_0)/2^d, r_0\big)$

$\underline{\mathsf{MakeHint}_q(z, r, \alpha)}$
11 $r_1 := \mathsf{HighBits}_q(r, \alpha)$
12 $v_1 := \mathsf{HighBits}_q(r + z, \alpha)$
13 **return** $[\![r_1 \neq v_1]\!]$

$\underline{\mathsf{UseHint}_q(h, r, \alpha)}$
14 $m := (q - 1)/\alpha$
15 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
16 **if** $h = 1$ and $r_0 > 0$ **return** $(r_1 + 1) \bmod^+ m$
17 **if** $h = 1$ and $r_0 \leq 0$ **return** $(r_1 - 1) \bmod^+ m$
18 **return** r_1

$\underline{\mathsf{Decompose}_q(r, \alpha)}$
19 $r := r \bmod^+ q$
20 $r_0 := r \bmod^\pm \alpha$
21 **if** $r - r_0 = q - 1$
22 **then** $r_1 := 0$; $r_0 := r_0 - 1$
23 **else** $r_1 := (r - r_0)/\alpha$
24 **return** (r_1, r_0)

$\underline{\mathsf{HighBits}_q(r, \alpha)}$
25 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
26 **return** r_1

$\underline{\mathsf{LowBits}_q(r, \alpha)}$
27 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
28 **return** r_0

Figure 3: Supporting algorithms for Dilithium.

1. $\mathsf{UseHint}_q(\mathsf{MakeHint}_q(\mathbf{z}, \mathbf{r}, \alpha), \mathbf{r}, \alpha) = \mathsf{HighBits}_q(\mathbf{r} + \mathbf{z}, \alpha)$.
2. *Let* $\mathbf{v}_1 = \mathsf{UseHint}_q(\mathbf{h}, \mathbf{r}, \alpha)$. *Then* $\|\mathbf{r} - \mathbf{v}_1 \cdot \alpha\|_\infty \leq \alpha + 1$. *Furthermore, if the number of* 1*'s in* $\mathbf{h}$ *is* ω*, then all except at most* ω *coefficients of* $\mathbf{r} - \mathbf{v}_1 \cdot \alpha$ *will have magnitude at most* $\alpha/2$ *after centered reduction modulo* q.
3. *For any* $\mathbf{h}, \mathbf{h}'$*, if* $\mathsf{UseHint}_q(\mathbf{h}, \mathbf{r}, \alpha) = \mathsf{UseHint}_q(\mathbf{h}', \mathbf{r}, \alpha)$*, then* $\mathbf{h} = \mathbf{h}'$.

Lemma 2. *If* $\|\mathbf{s}\|_\infty \leq \beta$ *and* $\|\mathsf{LowBits}_q(\mathbf{r}, \alpha)\|_\infty < \alpha/2 - \beta$*, then*

$$\mathsf{HighBits}_q(\mathbf{r}, \alpha) = \mathsf{HighBits}_q(\mathbf{r} + \mathbf{s}, \alpha).$$

Lemma 3. *Let* $(\mathbf{r}_1, \mathbf{r}_0) = \mathsf{Decompose}_q(\mathbf{r}, \alpha)$*,* $(\mathbf{w}_1, \mathbf{w}_0) = \mathsf{Decompose}_q(\mathbf{r} + \mathbf{s}, \alpha)$*, and* $\|\mathbf{s}\|_\infty \leq \beta$*. Then*

$$\|\mathbf{s} + \mathbf{r}_0\|_\infty < \alpha/2 - \beta \iff \mathbf{w}_1 = \mathbf{r}_1 \wedge \|\mathbf{w}_0\|_\infty < \alpha/2 - \beta.$$

3 Signature

The Key Generation, Signing, and Verification algorithms for our signature scheme are presented in Fig. 4. We present the deterministic version of the scheme in which the randomness used in the signing procedure is generated (using SHAKE-256) as a deterministic function of the message and a small secret key. Since our signing procedure may need to be repeated several times until a signature is produced, we also append a counter in order to make the SHAKE-256 output differ with each signing attempt of the same message. Also due to the fact that each (possibly long) message may require several iterations to sign, we compute an initial digest of the message using a collision-resistant hash function, and use this digest in place of the message throughout the signing procedure.

As discussed in Section 1.2, the main design improvement of Dilithium over the scheme in Fig. 1 is that the public key size is approximately halved at the expense of fewer than a hundred extra bytes in the signature. To accomplish the size reduction, the key generation algorithm outputs $\mathbf{t}_1 := \mathsf{Power2Round}_q(\mathbf{t}, d)$ as the public key instead of $\mathbf{t}$ as in Fig. 1. This means that instead of $\lceil \log q \rceil$ bits per coefficient, the public key requires $\lceil \log q \rceil - d$

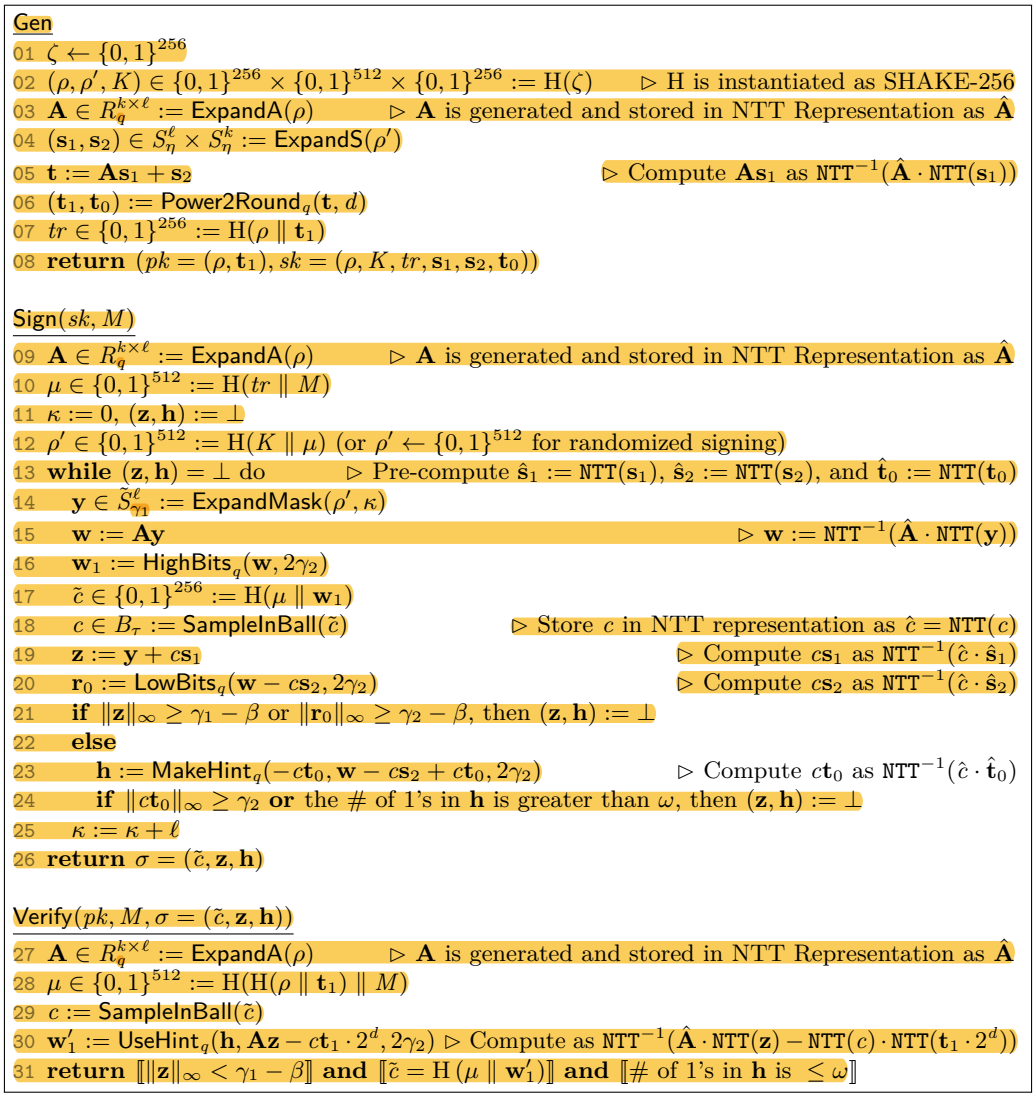

Figure 4: The pseudo-code for deterministic and randomized versions of Dilithium. The only difference between the two versions is in Line 12, where ρ' is either a function of the key and message, or is chosen completely at random.

bits. In our instantiations, $q \approx 2^{23}$ and $d = 13$, which means that instead of 23 bits in each public key coefficient, there are instead 10.

The main complication due to not having the entire $\mathbf{t}$ in the public key is that the verification algorithm is no longer able to exactly compute $\mathbf{w}_1'$ in Line 13 in Fig. 1. In order to do this, the verification algorithm will need the high order bits of $\mathbf{Az} - c\mathbf{t}$, but it can only compute $\mathbf{Az} - c\mathbf{t}_1 \cdot 2^d = \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0$. Even though the high-order bits of $c\mathbf{t}_0$ are 0,its presence in the sum creates "carries" which may have an effect on the higher bits. The signer thus sends these carries as a hint to the verifier. Heuristically, based on our parameter choices, there should not be more than ω positions in which a carry is caused. The signer therefore simply sends the positions in which these carries occur (these are the extra bytes in Dilithium compared to the signature in Fig. 1), which allows the verifier to compute the high order bits of $\mathbf{Az} - c\mathbf{t}$.

3.1 Implementation Notes and Efficiency Trade-offs

To keep the size of the public (and secret) key small, both the Sign and Verify procedures begin with extracting the matrix $\mathbf{A}$ (or more accurately, its NTT domain representation $\hat{\mathbf{A}}$) from the seed ρ. If storage space is not a factor, then $\hat{\mathbf{A}}$ can be pre-computed and be part of the secret/public key. The signer can additionally pre-compute the NTT domain representations of $\mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0$ to slightly speed up the signing operation. At the other extreme, if the signer wants to store as small a secret key as possible, he only needs to store a 32-byte secret seed ζ which is used to generate the randomness to create ρ, K, and $\mathbf{s}_1, \mathbf{s}_2$ in the key generation algorithm. Furthermore, one can also keep the memory for intermediate computations low by only keeping the parts of the NTT domain representation that one is currently working with.

3.2 Deterministic vs. Randomized Signatures

Dilithium gives an option of producing either deterministic (i.e. the signature of a particular message is always the same) or randomized signatures. The only implementational difference between the two options occurs on Line 12 where the seed ρ' is either derived from the message and a key (in deterministic signing) or is chosen completely at random (for randomized signing).

One may want to consider randomized signatures in situations where the side channel attacks of [SBB+18, PSS+18] exploiting determinism are applicable. Another situation where one may want to avoid determinism is when the signer does not wish to reveal *the message* that is being signed. While there is no timing leakage of the secret key, there is timing leakage of the message if the scheme is deterministic. Since the randomness of the scheme is derived from the message, the number of aborts for a particular message will always be the same.

3.3 Correctness

In this section, we prove the correctness of the signature scheme.

If $\|c\mathbf{t}_0\|_\infty < \gamma_2$, then by Lemma 1 we know that

$$\mathsf{UseHint}_q(\mathbf{h}, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\,.$$

Since $\mathbf{w} = \mathbf{Ay}$ and $\mathbf{t} = \mathbf{As}_1 + \mathbf{s}_2$, we have that

$$\mathbf{w} - c\mathbf{s}_2 = \mathbf{Ay} - c\mathbf{s}_2 = \mathbf{A}(\mathbf{z} - c\mathbf{s}_1) - c\mathbf{s}_2 = \mathbf{Az} - c\mathbf{t}, \tag{2}$$

and $\mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0 = \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d$. Therefore the verifier computes

$$\mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\,.$$

Table 2: Parameters of Dilithium

NIST Security Level	2	3	5
Parameters			
q [modulus]	8380417	8380417	8380417
d [dropped bits from $\mathbf{t}$]	13	13	13
τ [# of ± 1's in c]	39	49	60
challenge entropy $[\log \binom{256}{\tau} + \tau]$	192	225	257
γ_1 [$\mathbf{y}$ coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, ℓ) [dimensions of $\mathbf{A}$]	$(4, 4)$	$(6, 5)$	$(8, 7)$
η [secret key range]	2	4	2
β $[\tau \cdot \eta]$	78	196	120
ω [max. # of 1's in the hint $\mathbf{h}$]	80	55	75
Repetitions (from Eq. (5))	4.25	5.1	3.85

Furthermore, because β is set such that $\|c\mathbf{s}_2\|_\infty \leq \beta$ and the signer checks in Line 21 that $\mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2) < \gamma_2 - \beta$, Lemma 2 implies that

$$\mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2 + c\mathbf{s}_2, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2) = \mathbf{w}_1. \quad (3)$$

Therefore, the $\mathbf{w}_1'$ computed by the verifier in the input to the hash function is the same as the $\mathbf{w}_1$ of the signer. And thus the verification procedure will always accept.

3.4 Number of Repetitions

We now want to compute the probability that Step 21 will set $(\mathbf{z}, \mathbf{h})$ to $\perp$. The probability that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ can be computed by considering each coefficient separately. For each coefficient σ of $c\mathbf{s}_1$, the corresponding coefficient of $\mathbf{z}$ will be between $-\gamma_1 + \beta + 1$ and $\gamma_1 - \beta - 1$ (inclusively) whenever the corresponding coefficient of $\mathbf{y}_i$ is between $-\gamma_1 + \beta + 1 - \sigma$ and $\gamma_1 - \beta - 1 - \sigma$. The size of this range is $2(\gamma_1 - \beta) - 1$, and the coefficients of $\mathbf{y}$ have $2\gamma_1 - 1$ possibilities. Thus the probability that every coefficient of $\mathbf{y}$ is in the good range is

$$\left(\frac{2(\gamma_1 - \beta) - 1}{2\gamma_1 - 1}\right)^{256 \cdot \ell} = \left(1 - \frac{\beta}{\gamma_1 - 1/2}\right)^{\ell n} \approx \mathrm{e}^{-256 \cdot \beta \ell / \gamma_1}, \quad (4)$$

where we used the fact that our values of γ_1 are large compared to $1/2$.

We now move to computing the probability that we have

$$\|\mathbf{r}_0\|_\infty = \|\mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta\,.$$

If we (heuristically) assume that the low order bits are uniformly distributed modulo $2\gamma_2$, then there is a

$$\left(\frac{2(\gamma_2 - \beta) - 1}{2\gamma_2}\right)^{256 \cdot k} \approx \mathrm{e}^{-256 \cdot \beta k / \gamma_2}$$

probability that all the coefficients are in the good range (using the fact that our values of β are large compared to $1/2$). Therefore, the probability that Step 21 passes is

$$\approx \mathrm{e}^{-256 \cdot \beta(\ell/\gamma_1 + k/\gamma_2)}\,. \quad (5)$$

It is more difficult to formally compute the probability that Step 24 results in a restart. The parameters were set such that heuristically $(\mathbf{z}, \mathbf{h}) = \perp$ with probability between 1 and 2%. Therefore the vast majority of the loop repetitions will be caused by Step 21.

We would like to stress that the expected number of iterations is independent of the secret key $\mathbf{s}_1, \mathbf{s}_2$ and therefore no information about them can be gained by an attack that counts the iterations.

4 Dilithium Challenge and Extended Security Parameters

Table 3: Challenge and Extended Security Parameters for Dilithium. The numbers in parentheses for SIS security are for the strongly-unforgeable (i.e. hard to come up with a second distinct signature for an already-seen message/signature pair) version of the signature scheme.

Challenge and Extended Sets	1- -	1-	5+	5++
q	8380417	8380417	8380417	8380417
d	10	13	13	13
weight of c	24	30	60	60
challenge entropy	135	160	257	257
γ_1	2^{17}	2^{17}	2^{19}	2^{19}
γ_2	$(q-1)/128$	$(q-1)/128$	$(q-1)/32$	$(q-1)/32$
(k, ℓ)	$(2, 2)$	$(3, 3)$	$(9, 8)$	$(10, 9)$
η	6	3	2	2
β	144	90	120	120
ω	10	80	85	90
pk size (bytes)	864	992	2912	3232
sig size (bytes)	1196	1843	5246	5892
Exp. reps (from Eq. (5))	5.2	4.87	4.59	5.48
BKZ block-size b to break SIS	190 (165)	305 (305)	1055 (1005)	1200 (1145)
Core-SVP Classical	55 (49)	89 (89)	308 (293)	360 (334)
BKZ block-size b to break LWE	200	305	1020	1175
Core-SVP Classical	58	89	298	343

In addition to the parameters presented in Table 1 and Table 2 which correspond to various NIST security levels, we also give parameters that are below and above these levels. The purpose of the two parameter sets below NIST level 1 (called 1- - and 1-) is to encourage cryptanalysis and to be an early warning in case the SIS and LWE problems are easier than expected. The 1- - level has less than 60-bits of Core-SVP security, and it is therefore conceivable that a non-trivial computational effort could break these parameters in the next several years. It can also be a good gauge of progress on the computational problems underlying Dilithium. The 1- security level, on the other hand, has approximately 90 bits of Core-SVP security and BKZ block-size of around 300. Breaking SVP or LWE at this level in the near future without a gargantuan computational effort would indicate that a big cryptanalytical leap has been made, and one should therefore consider the NIST level 2 proposed parameters to be in danger. In this case, an immediate switch to levels 3 or 5 would be strongly encouraged.

At the other end of the spectrum, we also propose parameters that are above NIST level 5, which we call 5+ and 5++. For these parameter sets, we do not increase the hardness

of any symmetric primitives (e.g. PRFs or collision-resistant hash functions) that are used inside our protocol, but only increase the security of the underlying lattice problems. The goal of these parameter sets is to show what the parameters would be should there be a moderate improvement in lattice cryptanalysis. In terms of Core-SVP hardness, the 5++ level has about twice the required security of NIST level 3, and is therefore a good representative of what the parameters may look like if moderate improvements in lattice-based cryptanalysis necessitate a change. Such a change would be recommended in case some future cryptanalytic effort begins to threaten the security of the level 3 parameter set.

5 Implementation Details

5.1 Alternative way of decomposing and computing the hints

In the pseudo-code for signing in Figure 4, the decomposition function Decompose is called 3 times per iteration of the rejection sampling loop. We have presented the algorithm in this way for the ease of exposition and for compatibility with the security proofs in [KLS18, DKL+18]. In the implementation, we use an alternative and faster way of computing the hint vector $\mathbf{h}$ and the rejection condition based on the low part $\mathbf{w}_0$ of $\mathbf{w}$. First note that Lemma 3 says that instead of computing $(\mathbf{r}_1, \mathbf{r}_0) = \mathsf{Decompose}_q(\mathbf{w} - c\mathbf{s}_2, \alpha)$ and checking whether $\|\mathbf{r}_0\|_\infty < \gamma_2 - \beta$ and $\mathbf{r}_1 = \mathbf{w}_1$, it is equivalent to just check that $\|\mathbf{w}_0 - c\mathbf{s}_2\|_\infty < \gamma_2 - \beta$, where $\mathbf{w}_0$ is the low part of $\mathbf{w}$. If this check passes, $\mathbf{w}_0 - c\mathbf{s}_2$ is the low part of $\mathbf{w} - c\mathbf{s}_2$. Next, recall that by the definition of the MakeHint function, a coefficient of a polynomial in $\mathbf{h}$ as computed in Figure 4 is non-zero precisely if the high parts of the corresponding coefficients of $\mathbf{w} - c\mathbf{s}_2$ and $\mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0$ differ. Now, we have already computed the full decomposition $\mathbf{w} = \alpha\mathbf{w}_1 + \mathbf{w}_0$ of $\mathbf{w}$, and we know that $\alpha\mathbf{w}_1 + (\mathbf{w}_0 - c\mathbf{s}_2)$ is the correct decomposition of $\mathbf{w} - c\mathbf{s}_2$. But then, $\alpha\mathbf{w}_1 + (\mathbf{w}_0 - c\mathbf{s}_2 + c\mathbf{t}_0)$ is the correct decomposition of $\mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0$ (i.e. the high part is $\mathbf{w}_1$) if and only if each coefficient of $\mathbf{w}_0 - c\mathbf{s}_2 + c\mathbf{t}_0$ lies in the interval $(-\gamma_2, \gamma_2]$, or, when some coefficient is $-\gamma_2$ and the corresponding coefficient of $\mathbf{w}_1$ is zero. The last condition is due to the border case in the Decompose function. On the other hand, if these conditions are not true, then the high parts must differ and it then follows that for computing the hint vector $\mathbf{h}$ it suffices to just check these conditions on the coefficients of $\mathbf{w}_0 - c\mathbf{s}_2 + c\mathbf{t}_0$.

5.2 Bit-packing

We now describe how we encode vectors as byte strings. This is needed for absorbing them into SHAKE and defining the data layout of the keys and signature. To reduce the computation time spent on SHAKE and the sizes of keys and signatures, we use bit-packing. The general rule that we will follow is that if the range what a element x is in consists exclusively of non-negative integers, then we simply pack the integer x. If x is from a range $[-a, b]$ that may contain some negative integers, then we pack the positive integer $b - x$.

We begin with the vector $\mathbf{w}_1$. It consists of k polynomials $w_{1,1}, \ldots, w_{1,k}$ in R_q with coefficients that are roundings of elements in $\mathbb{Z}_q$ with respect to $\alpha = 2\gamma_2$. It follows that its coefficients lie in $\{0, \ldots, 15\}$ for NIST levels 3 and 5, and in $\{0, \ldots, 43\}$ for NIST level 2. These can be represented by 4 and, respectively, 6, bits each. This allows $\mathbf{w}_1$ to be packed in a string of $k \cdot 256 \cdot 4/8 = k \cdot 128$, respectively $k \cdot 192$ bytes. For NIST security levels 3 and 5, each byte encodes two consecutive coefficients of a polynomial $w_{1,i}$ in its low 4 bits and high 4 bits, respectively. See Figure 5 for an explanation of the exact bit packing. For NIST security level 2, the packing looks as in Figure 6.

Next we turn to the vector $\mathbf{t}_1$, which is the power-of-two rounding of $\mathbf{t}$. Note that $q - 1 = 2^{23} - 2^{13} = (2^{10} - 1)2^{13}$, which shows that the coefficients of the k polynomials

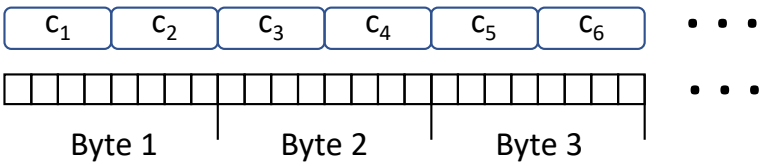

Figure 5: Bit-packing $\mathbf{w}_1$ for NIST levels 3 and 5. The k polynomials comprising $\mathbf{w}_1$ are $w_{1,1}, \ldots, w_{1,k}$ and we let $c_1, \ldots, c_{256}$ be the coefficients of $w_{1,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $w_{1,2}$, etc.

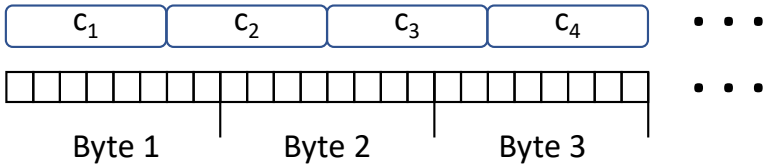

Figure 6: Bit-packing $\mathbf{w}_1$ for NIST level 2

of $\mathbf{t}_1$ lie in $\{0, \ldots, 2^{10} - 1\}$ and can be represented by 10 bits each. These 10 bits per coefficient, in little-endian byte-order, are bit-packed. In total $\mathbf{t}_1$ needs $k \cdot 256 \cdot 10/8 = 320k$ bytes. See Figure 7 for an explanation of the exact bit packing.

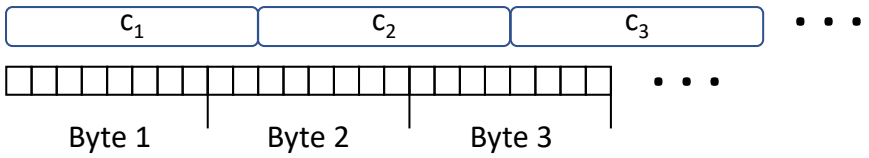

Figure 7: Bit-packing $\mathbf{t}_1$. The k polynomials comprising $\mathbf{t}_1$ are $t_{1,1}, \ldots, t_{1,k}$ and we let $c_1, \ldots, c_{256}$ be the coefficients of $t_{1,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $t_{1,2}$, etc.

The coefficients of the polynomials of $\mathbf{t}_0$ can be written in the form $2^{12} - v$ with $v \in \{0, \ldots, 2^{13} - 1\}$. These v in little endian byte-order are bit-packed. This results in $256 \cdot 13/8$ bytes per polynomial and $k \cdot 256 \cdot 13/8 = 416k$ bytes for $\mathbf{t}_0$. See Figure 8 for an explanation of the exact bit packing.

The polynomials in $\mathbf{s}_1$ and $\mathbf{s}_2$ have coefficients with infinity norm at most η. So every coefficient of these polynomials is equivalent modulo q to $\eta - c$ with some $c \in \{0, \ldots, 2\eta\}$. In the bit packing the values for c are stored so that each polynomial needs $256\lceil \log(2\eta + 1) \rceil / 8$ bytes. This amounts to $256 \cdot 3/8 = 96$ bytes for NIST levels 2 and 5, and $256 \cdot 4/8 = 128$ bytes for level 3. The bit-packing is done similarly to the case of $\mathbf{w}_1$, $\mathbf{t}_1$ and $\mathbf{t}_0$. See Figure 9 for an explanation of the exact bit packing.

Finally, $\mathbf{z}$ contains polynomials whose coefficients are equivalent modulo q to $\gamma_1 - c$ with $c \in \{0, \ldots, 2\gamma_1 - 1\}$ and these values c are bit packed. For NIST levels 3 and 5, bit-packing $\mathbf{z}$ requires $l \cdot 256 \cdot 20/8 = 640l$ bytes and blocks of 2 coefficients are stored in 5 consecutive bytes. See Figure 10 for an explanation of the exact bit packing. For NIST level 2, the bit-packing of $\mathbf{z}$ requires $l \cdot 256 \cdot 18/8 = 576l$ bytes

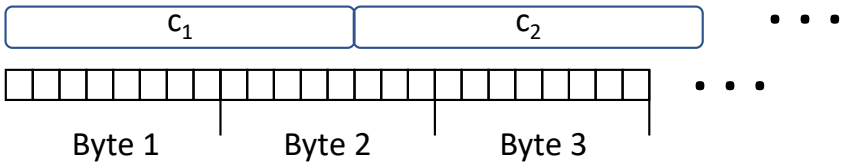

Figure 8: Bit-packing $\mathbf{t}_0$. The k polynomials comprising $\mathbf{t}_0$ are $t_{0,1}, \ldots, t_{0,k}$ with c_i defined such that $2^{13} - c_1, \ldots, 2^{13} - c_{256}$ are the coefficients of $t_{0,1}$ (with the lower powers first), $2^{13} - c_{257}, \ldots, 2^{13} - c_{512}$ are the coefficients of $t_{0,2}$, etc.

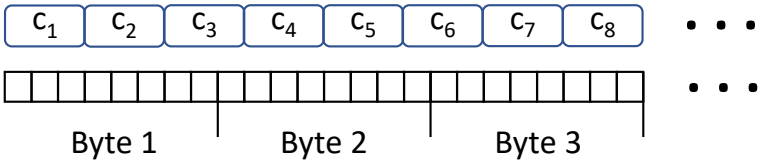

Figure 9: Bit-packing $\mathbf{s}_i$. The ℓ polynomials comprising $\mathbf{s}_1$ are $s_{1,1}, \ldots, s_{1,\ell}$ and we let $\eta - c_1, \ldots, \eta - c_{256}$ be the coefficients of $s_{1,1}$ (with the lower powers first), $\eta - c_{257}, \ldots, \eta - c_{512}$ be the coefficients of $s_{1,2}$, etc. where $c_i \in \{0, \ldots, 2\eta\}$. The k polynomials comprising $\mathbf{s}_2$ are $s_{2,1}, \ldots, s_{2,\ell}$ and we let $\eta - c_1, \ldots, \eta - c_{256}$ be the coefficients of $s_{2,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $s_{2,2}$, etc. $c_i \in \{0, \ldots, 2\eta\}$. The above picture is for parameter sets where c_i requires three bits per coefficient. Packing four bits per coefficient is done in the obvious manner and looks like the packing of $\mathbf{w}_i$ in Figure 5.

5.3 Hashing

Hashing to a Ball. The function H in Line 17 of Figure 4 absorbs the concatenation of μ and $\mathbf{w}_1$ into SHAKE-256, and produces a 32-byte output $\tilde{c}$. We now precisely specify the operation of the function $\mathsf{SampleInBall} : \tilde{c} \mapsto c \in B_\tau$ described in Figure 2 as it is used in our signature scheme. The SampleInBall routine absorbs the 32 bytes of $\tilde{c}$ into SHAKE-256. Throughout its operations the function squeezes SHAKE-256 in order to obtain a stream of random bytes of variable length. The first τ bits in the first 8 bytes of this random stream are interpreted as τ random sign bits $s_i \in \{0, 1\}$, $i = 0, \ldots, \tau - 1$. The remaining $64 - \tau$ bits are discarded. In each iteration of the for loop in the Algorithm in Figure 2, we use rejection sampling on elements from $\{0, \ldots, 255\}$ until we obtain a $j \in \{0, \ldots, i\}$. An element in $\{0, \ldots, 255\}$ is obtained by interpreting the next byte of the random stream from SHAKE-256 as a number in this set. For the sign s the corresponding s_{i-196} is used.

Expanding the Matrix A. The function ExpandA maps a uniform seed $\rho \in \{0, 1\}^{256}$ to a matrix $\mathbf{A} \in R_q^{k \times l}$ in NTT domain representation. It computes each polynomial $\hat{a}_{i,j} \in R_q$ of $\hat{\mathbf{A}}$ separately. For the coefficient $\hat{a}_{i,j}$ it absorbs the 32 bytes of ρ immediately followed by two bytes representing $0 \leq 256 * i + j < 2^{16}$ in little-endian byte order into SHAKE-128. In the AES variant ρ is used as the key and $256i + j$ is extended to a 12-byte nonce for AES in counter mode. The output stream of SHAKE-128 or AES256ctr is interpreted as a sequence of integers between 0 and $2^{23} - 1$. This is done by setting the highest bit of every third byte to zero and interpreting blocks of 3 consecutive bytes in little endian byte order. So for example the three bytes b_0, b_1 and b_2 are used to get the integer $0 \leq b_2' \cdot 2^{16} + b_1 \cdot 2^8 + b_0 \leq 2^{23} - 1$ where b_2' is the logical AND of b_2 and $2^{128} - 1$. Finally, ExpandA performs rejection sampling on these 23-bit integers to sample the 256 coefficients

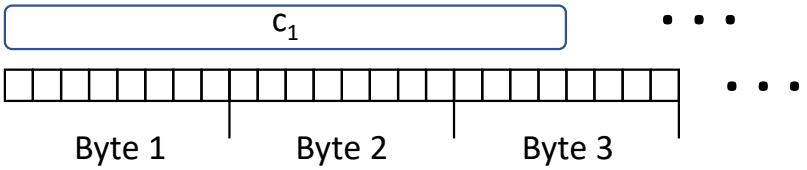

Figure 10: Bit-packing $\mathbf{z}$. The ℓ polynomials comprising $\mathbf{z}$ are $z_1, \ldots, z_\ell$ and we let $\gamma_1 - c_1, \ldots, \gamma_1 - c_{256}$ be the coefficients of z_1 (with the lower powers first), $\gamma_1 - c_{257}, \ldots, \gamma_1 - c_{512}$ be the coefficients of z_2, etc. In the above, each coefficient of $\mathbf{z}$ uses 20 bits. The packing for level 2, where each coefficient is 18 bits looks analogously.

$a_{i,j}(r_0), a_{i,j}(-r_0), \ldots, a_{i,j}(r_{127})a_{i,j}(-r_{127})$ of $\hat{a}_{i,j}$ uniformly from the set $\{0, \ldots, q-1\}$ in the order of our NTT domain representation.

Sampling the vectors $\mathbf{s_1}$ and $\mathbf{s_2}$. The function ExpandS maps ρ' to $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k$. It computes each of the $\ell + k$ polynomials in $\mathbf{s}_1$ and $\mathbf{s}_2$ independently. For the i-th polynomial, $0 \le i < \ell + k$, it absorbs the 64 bytes of ρ' concatenated with 2 bytes representing i in little endian byte order into SHAKE-256. In the AES variant, the first 32 bytes of ρ' are used as the key and i is extended to a 12 byte nonce for AES-256 in counter mode. Then the output bytes are used to create a sequence of uniformly random positive numbers in the range $\{0, \ldots, 2\eta\}$ by performing rejection sampling. Concretely, the lower and uppper four bits of each output byte are interpreted as two integers in $\{0, \ldots, 15\}$. Then, in the case of $\eta = 2$, an integer is accepted when it is less than 15 and then reduced modulo 5. In the case of $\eta = 4$, an integer is accepted when it is less then $2\eta + 1 = 9$. Finally, the polynomial coefficients are obtained by subtracting the positive numbers from η.

Sampling the vectors $\mathbf{y}$. The function ExpandMask maps (ρ', κ) to $\mathbf{y} \in \tilde{S}_{\gamma_1}^l$, where $\kappa \ge 0$. It computes each of the l coefficients of $\mathbf{y}$, which are polynomials in $\tilde{S}_{\gamma_1}$, independently. For the i-th polynomial, $0 \le i < \ell$, it absorbs the 64 bytes of ρ' concatenated with the 2 bytes representing $\kappa + i$ in little endian byte order into SHAKE-256. In the AES variant, the first 32 bytes of ρ' are used as the key and $\kappa + i$ is zero-padded to a 12 byte nonce for AES-256 in counter mode. Then the output bytes are used to create a positive number in the range $\{0, \ldots, 2\gamma_1 - 1\}$, and then the integers comprising $\mathbf{y}$ are obtained by subtracting this positive number from γ_1. Because γ_1 is a power of 2, no rejection sampling is required. Because γ_1 is 2^{17} and 2^{19}, for different security levels, generating the consecutive integers comprising $\mathbf{y}$ involves either taking 18 or 20 bits at a time from the byte stream.

Other uses of H. In Line 07, H is called with inputs $\rho \parallel \mathbf{t}_1$. The function absorbs the 32 bytes of ρ followed by the $k \cdot 256 \cdot 9/8$ bytes of the bit-packed representation of $\mathbf{t}_1$ into SHAKE-256 and takes the first 32 bytes of the first output block of SHAKE-256 as the output hash. In line 10, the input is $tr \parallel M$. Here the concatenation of tr and the message string M are absorbed into SHAKE-256 and the first 64 output bytes are used as the resulting hash. The input $K \parallel \mu$ on line 12 is handled in the same way.

5.4 Data layout of keys and signature

Public key. The public key, containing ρ and $\mathbf{t}_1$, is stored as the concatenation of the bit-packed representations of ρ and $\mathbf{t}_1$ in this order. Therefore, it has a size of $32 + 320k$ bytes.

Secret key. The secret key contains ρ, K, tr, $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{t}_0$ and is also stored as the concatenation of the bit-packed representation of these quantities in the given order. Consequently, a secret key requires $32 + 32 + 32 + 32((k + \ell) \cdot \lceil \log(2\eta + 1) \rceil + 13k)$ bytes. As mentioned previously, the signer also has the option of simply storing the 32-byte value ζ and then simply re-deriving all the other elements of the secret key.

Signature. The signature byte string is the concatenation of a bit packed representation of $\tilde{c}$ and encodings of $\mathbf{z}$ and $\mathbf{h}$, in this order. The packing of $\mathbf{z}$ was described in Fig. 10 and the challenge $\tilde{c}$ has 32 bytes and so does not require any special packing procedure. We now describe the encoding of $\mathbf{h}$, which needs $\omega + k$ bytes. Together all the polynomials in the vector $\mathbf{h}$ have at most ω non-zero coefficients. It is thus sufficient to store the locations of these non-zero coefficients. Each of the first ω bytes of the byte string representing $\mathbf{h}$ is the index i of the next non-zero coefficient in its polynomial, i.e. $0 \leq i \leq 255$, or zero if there are no more non-zero coefficients. The bytes numbers ω up to $\omega + k - 1$ record the k positions j of the polynomial boundaries in the string of ω coefficient indices, where $0 \leq j \leq \omega$.

Therefore, a signature requires $32l \cdot (1 + \log \gamma_1) + \omega + k + 32$ bytes.

5.5 Constant time implementation

Our reference implementation does not branch depending on secret data and does not access memory locations that depend on secret data. For the modular reductions that are needed for the arithmetic in R_q we never use the '%' operator of the C programming language. Instead we use Montgomery reductions without the correction steps and special reduction routines that are specific to our modulus q. For computing the rounding functions described in Section 2.4, we have implemented branching-free algorithms. On the other hand, when it is safe to reveal information, we have not tried to make the code constant-time. This includes the computation of the challenges and the rejection conditions in the signing algorithm. When performing rejection sampling, our code reveals which of the conditions was the reason for the rejection, and in case of the norm checks, which coefficient violated the bound. This is safe since the rejection probabilities for each coefficient are independent of secret data. The challenges reveal information about $\mathrm{H}(\mu \parallel \mathbf{w}_1)$ also in the case of rejected $\mathbf{y}$, but this does not reveal any information about the secret key when H is modelled as a random oracle and $\mathbf{w}_1$ has high min-entropy.

5.6 Reference implementation

Our reference NTT is a natural iterative implementation for 32 bit signed integers that uses Cooley-Tukey butterflies in the forward transform and Gentleman-Sande butterflies in the inverse transform. For modular reductions after multiplying with a precomputed root of unity we use the Montgomery algorithm as was already done before in e.g. [ADPS16]. In order that the reduced values are correct representatives, the precomputed roots contain the Montgomery factor $2^{32} \bmod q$. We also use Montgomery reductions after the pointwise product of the polynomials in the NTT domain representations. Since we cannot get the Montgomery factor in at this point, these products are in fact Hensel remainders. We then make use of the fact that the NTT transform is linear and multiply by an additional Montgomery factor after the inverse NTT when we divide out the factor 256.

The implementations of the functions ExpandA initially squeeze a number of output blocks of SHAKE-128 that give enough randomness with high probability. In the case of ExpandA, which samples uniform polynomials and hence needs at least $3 \cdot 256 = 768$ random bytes per polynomial, 5 blocks from SHAKE-128 of 168 bytes each are at least needed for one polynomial. They suffice with probability greater than $1 - 2^{-132}$.

5.7 AVX2 optimized implementation

We have written an optimized implementation of Dilithium for CPUs that support the AVX2 instruction set. Since the two most time-consuming operations are polynomial multiplication and the expansion of the matrix and vectors, the optimized implementation speeds up these two operations.

For polynomial multiplication, we use a vectorized version of the NTT. This NTT achieves a full multiplication of two polynomials including three NTTs and the pointwise multiplication in less than 5000 Haswell cycles and is about a factor of 4.5 faster than the reference C code compiled using gcc with full machine-specific optimizations turned on. Contrary to some other implementations (e.g. [ADPS16]), we do not use floating point instructions. When using floating point instructions, modular reductions are easily done by multiplying with a floating point inverse of q and rounding to get the quotient from which the remainder can be computed with another multiplication and a subtraction. Instead of this approach we use integer instructions only and the same Montgomery reduction methodology as in the reference C code. When compared to the floating point NTT from [ADPS16] applied to the Dilithium prime $q = 2^{23} - 2^{13} + 1$, our integer NTT is about two times faster.

At any time our AVX2 optimized NTT has 32 signed integer coefficients, of 32 bits each, loaded into 8 AVX2 vector registers. Each of these vector registers then contains 4 extended 64 bit coefficients. So after three levels of NTT the reduced polynomials fit completely into these 8 registers and we can transform them to linear factors without further loads and stores. In the second to last and last level the polynomials have degree less than 4. This means that every polynomial fits into one register but only half of the coefficients need to be multiplied by roots of unity. For this reason we shuffle the vectors in order to group together coefficients that need to be multiplied. The instruction that we use for this task are `perm2i128`, `vpunpcklqdq` and `vpunpckhqdq`. The vectors are shuffled already in levels 4 and 5 which has the advantage of cheaper loads for the constant roots of unity. The multiplications are performed using the `vpmuldq` instruction. This instruction computes a full 64 bit product of two 32 bit integers. It has a latency of 5 cycles on both Haswell and Skylake. In each level of the NTT half of the coefficients need to be multiplied. Therefore we can do four vector multiplications and Montgomery reductions in parallel. This hides some of the latency of the multiplication instructions.

For faster matrix and vector expansion, we use a vectorized SHAKE implementation that operates on 4 parallel sponges and hence can absorb and squeeze blocks in and out of these 4 sponges at the same time. For sampling this means that up to four coefficients can be sampled simultaneously.

5.8 Computational Efficiency

We have performed timing experiments with our reference implementation on a Skylake CPU. The results are presented in Table 1. They include the number of CPU cycles needed by the three operations key generation, signing and signature verification. These numbers are the medians of 1000 executions each. Signing was performed with a message size of 32 bytes. The computer we have used is a laptop with an Intel Core i7-6600U CPU that runs at a base clock frequency of up 2600 Mhz. The code was compiled with gcc 7.3.0.

6 Security Reductions

The standard security notion for digital signatures is UF-CMA security, which is security under chosen message attacks. In this security model, the adversary gets the public key and has access to a signing oracle to sign messages of his choice. The adversary's goal is to come up with a valid signature of a new message. A slightly stronger security requirement that

is sometimes useful is SUF-CMA (Strong Unforgeability under Chosen Message Attacks), which also allows the adversary to win by producing a different signature of a message that he has already seen.

It can be shown that in the (classical) random oracle model, Dilithium is SUF-CMA secure based on the hardness of the standard MLWE and MSIS lattice problems. The reduction, however, is not tight. Furthermore, since we also care about quantum attackers, we need to consider the security of the scheme when the adversary can query the hash function on a superposition of inputs (i.e. security in the quantum random oracle model – QROM). Since the classical security proof uses the "forking lemma" (which is essentially rewinding), the reduction does not transfer over to the quantum setting.

There are no counter-examples of schemes whose security is actually affected by the non-tightness of the reduction. For example, schemes like Schnorr signatures [Sch89], GQ signatures [GQ88], etc. all set their parameters ignoring the non-tightness of the reduction. Furthermore, the only known uses of the additional power of quantum algorithms against schemes whose security is based on quantum-resistant problems under a classical reduction involve "Grover-type" algorithms that improve exhaustive search (although it has been shown that there cannot be a "black-box" proof that the Fiat-Shamir transform is secure in the QROM [ARU14]).

The reason that there haven't been any attacks taking advantage of the non-tightness of the reduction is possibly due to the fact that there is an intermediate problem which is *tightly* equivalent, even under quantum reductions, to the UF-CMA security of the signature scheme. This problem is essentially a "convolution" of the underlying mathematical problem (such as MSIS or discrete log) with a cryptographic hash function H. It would appear that as long as there is no relationship between the structure of the math problem and H, solving this intermediate problem is not easier than solving the mathematical problem.[4]

Below, we will introduce the assumptions upon whose hardness the SUF-CMA security of our scheme is based. The first two assumptions, MLWE and MSIS, are standard lattice problems which are a generalization of LWE, Ring-LWE, SIS, and Ring-SIS. The third problem, SelfTargetMSIS is the aforementioned problem that's based on the combined hardness of MSIS and the hash function H. In the classical ROM, there is a (non-tight) reduction from MSIS to SelfTargetMSIS.

6.1 Assumptions

The MLWE Problem. For integers m, k, and a probability distribution $D : R_q \to [0, 1]$, we say that the advantage of algorithm A in solving the decisional $\mathsf{MLWE}_{m,k,D}$ problem over the ring R_q is

$$\begin{aligned}\mathrm{Adv}^{\mathsf{MLWE}}_{m,k,D} := \big|&\Pr[b = 1 \mid \mathbf{A} \leftarrow R_q^{m\times k}; \mathbf{t} \leftarrow R_q^m; b \leftarrow \mathsf{A}(\mathbf{A}, \mathbf{t})] \\ &- \Pr[b = 1 \mid \mathbf{A} \leftarrow R_q^{m\times k}; \mathbf{s}_1 \leftarrow D^k; \mathbf{s}_2 \leftarrow D^m; b \leftarrow \mathsf{A}(\mathbf{A}, \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2)]\big| .\end{aligned}$$

The MSIS Problem. To an algorithm A we associate the advantage function $\mathrm{Adv}^{\mathsf{MSIS}}_{m,k,\gamma}$ to solve the (Hermite Normal Form) $\mathsf{MSIS}_{m,k,\gamma}$ problem over the ring R_q as

$$\mathrm{Adv}^{\mathsf{MSIS}}_{m,k,\gamma}(\mathsf{A}) := \Pr\left[0 < \|\mathbf{y}\|_\infty \le \gamma \wedge [\ \mathbf{I} \mid \mathbf{A}\ ] \cdot \mathbf{y} = \mathbf{0} \mid \mathbf{A} \leftarrow R_q^{m\times k}; \mathbf{y} \leftarrow \mathsf{A}(\mathbf{A})\right] .$$

The SelfTargetMSIS Problem. Suppose that $\mathrm{H} : \{0, 1\}^* \to B_\tau$ is a cryptographic hash function. To an algorithm A we associate the advantage function

[4] In the ROM, there is indeed a (non-tight) reduction using the forking lemma that states that solving this problem is as hard as solving the underlying mathematical problem.

$$\mathrm{Adv}^{\mathsf{SelfTargetMSIS}}_{\mathrm{H},m,k,\gamma}(\mathsf{A}) :=$$
$$\Pr\left[\begin{matrix}0 \le \|\mathbf{y}\|_\infty \le \gamma \\ \wedge\, \mathrm{H}(\mu \parallel [\ \mathbf{I} \mid \mathbf{A}\ ] \cdot \mathbf{y}) = c\end{matrix} \;\middle|\; \mathbf{A} \leftarrow R_q^{m\times k}; \left(\mathbf{y} := \begin{bmatrix}\mathbf{r}\\ c\end{bmatrix}, \mu\right) \leftarrow \mathsf{A}^{|\mathrm{H}(\cdot)\rangle}(\mathbf{A})\right] .$$

Classical Hardness of SelfTargetMSIS. We now sketch the *classical* reduction from MSIS to SelfTargetMSIS. In this scenario, the attacker A is classical and only has classical access to the function H modeled as a random oracle. This reduction is a standard application of rewinding / forking lemma, and therefore we only give a sketch.

Suppose A makes queries $\mathrm{H}(\mu_1 \parallel \mathbf{w}_1), \ldots, \mathrm{H}'(\mu_k \parallel \mathbf{w}_k)$ and receives randomly-chosen responses $c_1, \ldots, c_k$, and finally outputs $\mu_i, \mathbf{y} = \begin{bmatrix}\mathbf{r}\\ c_i\end{bmatrix}$ satisfying $\mathrm{H}(\mu_i \parallel [\ \mathbf{I} \mid \mathbf{A}\ ] \cdot \mathbf{y}) = c_i$ for some $1 \le i \le k$. The reduction then rewinds A to the point where he made the query $\mathrm{H}(\mu_i \parallel \mathbf{w}_i)$ and reprograms the response to another randomly-chosen c_i'. The forking lemma then states that the successful A has a $\approx 1/k$ chance to create a forgery on the same i. That is, he will output $\mu_i, \mathbf{y}' = \begin{bmatrix}\mathbf{r}'\\ c_i'\end{bmatrix}$ satisfying $\mathrm{H}(\mu_i \parallel [\ \mathbf{I} \mid \mathbf{A}\ ] \cdot \mathbf{y}') = c_i'$. We therefore have that $[\ \mathbf{I} \mid \mathbf{A}\ ] \cdot (\mathbf{y} - \mathbf{y}') = 0$ and $\mathbf{y} - \mathbf{y}' \neq 0$ because $c_i \neq c_i'$. Furthermore, all the coefficients of $\mathbf{y}_i - \mathbf{y}_i'$ are small, thus giving a solution to MSIS.

6.2 Signature Scheme Security

The concrete security of Dilithium was analyzed in [KLS18], where it was shown that if H is a quantum random oracle (i.e., a quantum-accessible perfect hash function), the advantage of an adversary A breaking the SUF-CMA security of the signature scheme is

$$\mathrm{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Dilithium}}(\mathsf{A}) \le \mathrm{Adv}^{\mathsf{MLWE}}_{k,\ell,D}(\mathsf{B}) + \mathrm{Adv}^{\mathsf{SelfTargetMSIS}}_{\mathrm{H},k,\ell+1,\zeta}(\mathsf{C}) + \mathrm{Adv}^{\mathsf{MSIS}}_{k,\ell,\zeta'}(\mathsf{D}) + 2^{-254} \ ,^{5} \tag{6}$$

for D a uniform distribution over S_η, and

$$\zeta = \max\{\gamma_1 - \beta, 2\gamma_2 + 1 + 2^{d-1} \cdot \tau\}, \tag{7}$$

$$\zeta' = \max\{2(\gamma_1 - \beta), 4\gamma_2 + 2\}. \tag{8}$$

Furthermore, if the running times and success probabilities (i.e. advantages) of $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$ are $t_\mathsf{A}, t_\mathsf{B}, t_\mathsf{C}, t_\mathsf{D}, \epsilon_\mathsf{A}, \epsilon_\mathsf{B}, \epsilon_\mathsf{C}, \epsilon_\mathsf{D}$, then the lower bound on $t_\mathsf{A}/\epsilon_\mathsf{A}$ is within a small multiplicative factor of $\min t_i/\epsilon_i$ for $i \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}\}$.

Intuitively, the MLWE assumption is needed to protect against key-recovery, the SelfTargetMSIS is the assumption upon which new message forgery is based, and the MSIS assumption is needed for strong unforgeability. We will now sketch some parts of the security proof that are relevant to the concrete parameter setting.

6.2.1 UF-CMA Security Sketch

It was shown in [KLS18] that for zero-knowledge deterministic signature schemes, if an adversary having quantum access to H and classical access to a signing oracle can produce a forgery of a new message, then there is also an adversary who can produce a forgery without access to the signing oracle (so he only gets the public key).[6] The latter security model is

[5] To simplify the concrete security bound, we assume that ExpandA produces a uniform matrix $\mathbf{A} \in R_q^{k\times\ell}$, $\mathsf{ExpandMask}(K, \cdot)$ is a perfect pseudo-random function, and H is a perfect collision-resistant hash function.

[6] It was also shown in [KLS18] that the "deterministic" part of the requirement can be relaxed. The security proof simply loses a factor of the number of different signatures produced per message in its tightness. Thus, for example, if one were to implement the signature scheme (with the same secret key) on several devices with different random-number generators, the security of the scheme would not be affected much.

called UF-NMA – unforgeability under no-message attack. By the MLWE assumption, the public key $(\mathbf{A}, \mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2)$ is indistinguishable from $(\mathbf{A}, \mathbf{t})$ where $\mathbf{t}$ is chosen uniformly at random. The proof that our signature scheme is zero-knowledge is fairly standard and follows the framework from [Lyu09, Lyu12, BG14]. It is formally proved in [KLS18][7] and we sketch the proof in Appendix B.

If we thus assume that the $\mathsf{MLWE}_{k,\ell,D}$ problem is hard, where D is the distribution that samples a uniform integer in the range $[-\eta, \eta]$, then to prove UF-NMA security, we only need to analyze the hardness of the experiment where the adversary receives a random $(\mathbf{A}, \mathbf{t})$ and then needs to output a valid message/signature pair $M, (\mathbf{z}, \mathbf{h}, c)$ such that

- $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$
- $\mathrm{H}(\mu \parallel \mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)) = c$
- # of 1's in $\mathbf{h}$ is $\leq \omega$

Lemma 1 implies that one can rewrite

$$2\gamma_2 \cdot \mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2) = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d + \mathbf{u}, \tag{9}$$

where $\|\mathbf{u}\|_\infty \leq 2\gamma_2 + 1$. Furthermore, only ω coefficients of $\mathbf{u}$ will have magnitude greater than γ_2. If we write $\mathbf{t} = \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0$ where $\|\mathbf{t}_0\|_\infty \leq 2^{d-1}$, then we can rewrite Eq. (9) as

$$\mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d + \mathbf{u} = \mathbf{A}\mathbf{z} - c(\mathbf{t} - \mathbf{t}_0) + \mathbf{u} = \mathbf{A}\mathbf{z} - c\mathbf{t} + (c\mathbf{t}_0 + \mathbf{u}) = \mathbf{A}\mathbf{z} - c\mathbf{t} + \mathbf{u}'. \tag{10}$$

Note that the worst-case upper-bound for $\mathbf{u}'$ is

$$\|\mathbf{u}'\|_\infty \leq \|c\mathbf{t}_0\|_\infty + \|\mathbf{u}\|_\infty \leq \|c\|_1 \cdot \|\mathbf{t}_0\|_\infty + \|\mathbf{u}\|_\infty \leq \tau \cdot 2^{d-1} + 2\gamma_2 + 1.$$

Thus a (quantum) adversary who is successful at creating a forgery of a new message is able to find $\mathbf{z}, c, \mathbf{u}', M$ such that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$, $\|c\|_\infty = 1$, $\|\mathbf{u}'\|_\infty \leq \tau \cdot 2^{d-1} + 2\gamma_2 + 1$, and $M \in \{0,1\}^*$ such that

$$\mathrm{H}\left(\mu \parallel \frac{1}{2\gamma_2} \cdot [\ \mathbf{A} \mid \mathbf{t} \mid \mathbf{I}_k\ ] \cdot \begin{bmatrix} \mathbf{z} \\ c \\ \mathbf{u}' \end{bmatrix}\right) = c. \tag{11}$$

For simplicity of notation, we can define the function H' such that $\mathrm{H}(\mu \parallel x) = \mathrm{H}'(\mu \parallel 2\gamma_2 x)$, and so Eq. (11) becomes

$$\mathrm{H}'\left(\mu \parallel [\ \mathbf{A} \mid \mathbf{t} \mid \mathbf{I}_k\ ] \cdot \begin{bmatrix} \mathbf{z} \\ c \\ \mathbf{u}' \end{bmatrix}\right) = c.[8] \tag{12}$$

Since $(\mathbf{A}, \mathbf{t})$ is completely random, this is exactly the definition of the SelfTargetMSIS problem from above.

A standard forking lemma argument sketched in Section 6.1 can be used to show that an adversary solving the above problem in the (standard) random oracle model can be used to solve the MSIS problem. While giving a reduction using the forking lemma is a good "sanity check", it is not particularly useful for setting parameters due to its lack of tightness. So how does one set parameters? The Fiat-Shamir transform has been used for over 3 decades (and we have been aware of the non-tightness of the forking lemma for two of them), yet the parameter settings for schemes employing it have ignored this loss

[7] In that paper, it is actually proved that the underlying zero-knowledge proof is zero-knowledge and then the security of the signature scheme follows via black box transformations.

[8] Notice that the difference between H and H' is just a change in the format of the input.

in tightness. Implicitly, therefore, these schemes rely on the exact hardness of analogues (based on various assumptions such as discrete log [Sch89], one-wayness of RSA [GQ88], etc.) of the problem in Eq. (12).

The intuition for the security of the problem in Eq. (12) (and its discrete log, RSA, etc. analogues) is as follows: since H is a cryptographic hash function whose structure is completely independent of the algebraic structure of its inputs, choosing some μ "strategically" should not help – so the problem would be equally hard if the μ were fixed. Then, again relying on the independence of H' and the algebraic structure of its inputs, the only approach for obtaining a solution appears to be picking some $\mathbf{w}$, computing $\mathrm{H}'(\mu \parallel \mathbf{w}) = c$, and then finding $\mathbf{z}, \mathbf{u}'$ such that $\mathbf{Az} + \mathbf{u}' = \mathbf{w} + c\mathbf{t}$.[9] The hardness of finding such $\mathbf{z}, \mathbf{u}'$ with ℓ_∞-norms less than in Eq. (7) such that

$$\mathbf{Az} + \mathbf{u}' = \mathbf{t}' \tag{13}$$

for some $\mathbf{t}'$ is the problem whose concrete security we will be analyzing. Note that this bound is somehwat conservative because in Eq. (12) $\|\mathbf{z}\|_\infty < \gamma_1 - \beta \ll \tau \cdot 2^{d-1} + 2\gamma_2 + 1$. Furthermore, only ω coefficients of $\mathbf{u}'$ can be larger than $\tau \cdot 2^{d-1} + \gamma_2$. Thus most of the coefficients of the solution $(\mathbf{z}, \mathbf{u}')$ to Eq. (13) generated by a real forgery must have coefficients noticeably smaller than the maximum in Eq. (7).

6.2.2 The Addition of the Strong Unforgeabilty Property

To handle the strong-unforgeability requirement, one needs to handle an additional case. Intuitively, the reduction from UF-CMA to UF-NMA used the fact that a forgery of a new message will necessarily require the use of a challenge c for which the adversary has never seen a valid signature (i.e., $(\mathbf{z}, \mathbf{h}, c)$ was never an output by the signing oracle). To prove strong-unforgeability, we also have to consider the case where the adversary sees a signature $(\mathbf{z}, \mathbf{h}, c)$ for μ and then only changes $(\mathbf{z}, \mathbf{h})$. In other words, the adversary ends up with two valid signatures such that

$$\mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2) = \mathsf{UseHint}_q(\mathbf{h}', \mathbf{Az}' - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2).$$

By Lemma 1, the above equality can be shown to imply that there exist $\|\mathbf{z}\|_\infty \leq 2(\gamma_1 - \beta)$ and $\|\mathbf{u}\|_\infty \leq 4\gamma_2 + 2$ such that $\mathbf{Az} + \mathbf{u} = \mathbf{0}$.

6.3 Concrete Security Analysis

In Appendix C, we describe the best known lattice attacks against the problems in Eq. (6) upon which the security of our signature scheme is based. The best attacks involve finding short vectors in some lattice. The main difference between the MLWE and MSIS problems is that the MLWE problem involves finding a short vector in a lattice in which an "unusually short" vector exists. The MSIS problem, on the other hand, involves just finding a short vector in a random lattice. In knapsack terminology, the MLWE problem is a low-density knapsack, while MSIS is a high-density knapsack instance. The analysis for the two instances is slightly different and we analyze the MLWE problem in Appendix C.2 and the MSIS problem (as well as SelfTargetMSIS) in Appendix C.3.

We first follow the *core-SVP hardness* methodology from [ADPS16], based on the asymptotic costs of sieving [BDGL16a, Laa15]. It is significantly more conservative than prior ones used in lattice-based cryptography. We describe how we use it for MLWE and MSIS in sections C.2 and C.3. However, recent progresses in sieving in practice seems to mandate more refined estimates; we sketch such refinements in Section C.5 (the full details are provided in the Kyber Round 3 specification document). It expresses security in terms of the number of gates required for the computation, in the RAM model; because the

[9] This is indeed the intuition in the proof sketch in the classical random oracle model in Section 6.1.

algorithm of [ADPS16] makes random access to an exponentially large amount of memory, it is quite likely that any realistic implementation will be significantly more expensive.

While the MLWE and MSIS problems are defined over polynomial rings, we do not currently have any way of exploiting this ring structure, and therefore the best attacks are mounted by simply viewing these problems as LWE and SIS problems. The LWE and SIS problems are exactly as in the definitions of MLWE and MSIS in Section 6.1 with the ring R_q being replaced by $\mathbb{Z}_q$.

6.4 Conservative Choices for the MSIS Problem.

The security of Dilithium is based on the hardness of LWE and SIS-type problems. lattice-based encryption schemes are based solely on the LWE problem and its derivatives with added structure (e.g. MLWE, NTRU, etc.), it's natural that LWE-type problems will receive more attention in the future. While both problems are quite similar in that they require finding short vectors in a particular lattice and, furthermore, the best algorithms have the same core component, we believe it makes sense to be a little more conservative with the parameters on the SIS side. We have already been quite conservative in discounting the fact that only very few (i.e. ω) coefficients of the solution can be at the maximum ℓ_∞-length and the fact that we lost an additive factor of $\tau \cdot 2^{d-1}$ because we wanted to obtain a reduction from the inhomogeneous SIS problem with the target $\mathbf{t}$ rather than $\mathbf{t}_1$, even though we don't know of any improved attack against the latter.

There is one more place in which we could have been even more conservative against attacks that are not known to exist. The *classical* reduction from SelfTargetMSIS to MSIS sketched in section 6.1 increases the ℓ_∞-norm of the extracted solution by a factor of 2. More specifically, the length of the vector $\mathbf{y}_i - \mathbf{y}'_i$ is twice the length of the ζ in Eq. (7). There are no known algorithms that take advantage of this, and we did not take this reduction loss into account when computing the hardness of the SIS problem. For the sake of added conservativism, we now give the hardness of SIS assuming that there is indeed some possible exploit of this factor loss. Instead of classical Core-SVP SIS security 124, 186, and 265 for NIST levels 2,3, and 5, one would have corresponding security of 115, 172, and 256. So even in this ultra-conservative case, there is still ample SIS security in our parameter sets.

6.5 Changing Security Levels

The most straightforward way of raising/lowering the security of Dilithium is by changing the values of (k, ℓ) and then adapting the value of η (and then β and ω) accordingly as in Table 2. Increasing (k, ℓ) by 1 each results in the public key increasing by ≈ 300 bytes and the signature by ≈ 700 bytes; and increases security by ≈ 30 bits.

A different manner in which to increase security would be by lowering the values of γ_1 and/or γ_2. This would make forging signatures (whose hardness is based on the underlying SIS problem) more difficult. Rather than increasing the size of the public key / signature, the negative effect of lowering the γ_i is that signing would require more repetitions. One could similarly increase the value of η in order to make the LWE problem harder at the expense of more repetitions.[10] Because the increase in running time is rather dramatic (e.g. halving both γ_i would end up squaring the number of required repetitions as per Eq. (5)), we recommend increasing (k, ℓ) when needing to "substantially" increase security. Changing the γ_i should be reserved only for slight "tweaks" in the security levels.

[10] The two changes in this paragraph may require lowering d so as to keep the value $\|c\mathbf{t}_0\|_\infty$ smaller than γ_2 with high probability. Lowering d will increase the size of the public key.

References

[ABB+19] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Patrick Longa, and Jefferson E. Ricardini. The lattice-based digital signature scheme qtesla. Cryptology ePrint Archive, Report 2019/085, 2019. https://eprint.iacr.org/2019/085. 5

[ADH+19] Martin R Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 717–746. Springer, 2019. 37

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*, pages 327–343. USENIX Association, 2016. http://cryptojedi.org/papers/#newhope. 21, 22, 26, 27, 35, 37, 38

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In *ICALP (1)*, pages 403–415, 2011. 36

[AGPS19] Martin R Albrecht, Vlad Gheorghiu, Eamonn W Postlethwaite, and John M Schanck. Estimating quantum speedups for lattice sieves. Technical report, Cryptology ePrint Archive, Report 2019/1161, 2019. 37, 38

[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*, volume 10211 of *LNCS*, pages 65–102. Springer, 2017. https://eprint.iacr.org/2017/815. 37

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015. https://eprint.iacr.org/2015/046. 37

[ARU14] Andris Ambainis, Ansis Rosmanis, and Dominique Unruh. Quantum attacks on classical proof systems: The hardness of quantum rewinding. In *FOCS*, pages 474–483, 2014. 23

[BDGL16a] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA '16 Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete Algorithms*, pages 10–24. SIAM, 2016. https://eprint.iacr.org/2015/1128. 26, 37

[BDGL16b] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA*, pages 10–24. SIAM, 2016. 33

[BG14] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In *CT-RSA*, pages 28–47, 2014. 3, 5, 25, 33

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In *CHES*, pages 323–345, 2016. 3

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003. 36

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006*, pages 390–399. ACM, 2006. 6

[BS16] Jean-François Biasse and Fang Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In *SODA*, pages 893–902. SIAM, 2016. 36

[CDPR16] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 559–585. Springer, 2016. 36

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short Stickelberger class relations and application to ideal-SVP. In *EUROCRYPT (1)*, volume 10210 of *Lecture Notes in Computer Science*, pages 324–348, 2017. 36

[CGS14] Peter Campbell, Michael Groves, and Dan Shepherd. Soliloquy: A cautionary tale. In *ETSI 2nd Quantum-Safe Crypto Workshop*, pages 1–9, 2014. 36

[CN11a] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011. 33

[CN11b] 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 *LNCS*, pages 1–20. Springer, 2011. http://www.iacr.org/archive/asiacrypt2011/70730001/70730001.pdf. 37

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *CRYPTO (1)*, pages 40–56, 2013. 3

[DFMS19] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Security of the fiat-shamir transformation in the quantum random-oracle model. Cryptology ePrint Archive, Report 2019/190, 2019. https://eprint.iacr.org/2019/190. 7

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):238–268, 2018. 7, 17

[DLP14] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. In *ASIACRYPT*, pages 22–41, 2014. 3

[DSDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. Lwe with side information: Attacks and concrete security estimation. 2020. https://eprint.iacr.org/2020/292.pdf. 37, 38

[Duc18] Léo Ducas. Shortest vector from lattice sieving: a few dimensions for free. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 125–145. Springer, 2018. 37, 38

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *CCS*, pages 1857–1874, 2017. 3

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *CHES*, pages 530–547, 2012. 3

[GQ88] Louis C. Guillou and Jean-Jacques Quisquater. A "paradoxical" indentity-based signature scheme resulting from zero-knowledge. In *CRYPTO*, pages 216–231, 1988. 23, 26

[HPS11] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the shortest and closest lattice vector problems. In *IWCC*, volume 6639 of *Lecture Notes in Computer Science*, pages 159–190. Springer, 2011. 33

[KF17] Paul Kirchner and Pierre-Alain Fouque. Revisiting lattice attacks on overstretched NTRU parameters. In *EUROCRYPT (1)*, volume 10210 of *Lecture Notes in Computer Science*, pages 3–26, 2017. 36

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of fiat-shamir signatures in the quantum random-oracle model. In *EUROCRYPT*, pages 552–586, 2018. 5, 6, 7, 17, 24, 25

[Knu97] Donald Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3 edition, 1997. page 145. 10

[Laa15] Thijs Laarhoven. *Search problems in cryptography*. PhD thesis, Eindhoven University of Technology, 2015. 26, 33

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In *ASIACRYPT*, pages 598–616, 2009. 3, 25

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In *EUROCRYPT*, pages 738–755, 2012. 3, 6, 7, 25, 33

[LZ19] Qipeng Liu and Mark Zhandry. Revisiting post-quantum fiat-shamir. Cryptology ePrint Archive, Report 2019/262, 2019. https://eprint.iacr.org/2019/262. 7

[MLB17] Artur Mariano, Thijs Laarhoven, and Christian Bischof. A parallel variant of LDSieve for the SVP on lattices. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 23–30. IEEE, 2017. 37

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongswan's implementation of post-quantum signatures. In *CCS*, pages 1843–1855, 2017. 3

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *J. Cryptology*, 13(3):361–396, 2000. 6

[PSS+18] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *EuroS&P*, pages 338–352, 2018. 5, 14

[SBB+18] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking ed25519 in wolfssl. In *CT-RSA*, pages 1–20, 2018. 5, 14

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO*, pages 239–252, 1989. 23, 26

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994. 33

[YD17] Yang Yu and Léo Ducas. Second order statistical behavior of lll and bkz. In *International Conference on Selected Areas in Cryptography*, pages 3–22. Springer, 2017. 37

A Proofs for Rounding Algorithm Properties

The three lemmas below prove each of the three parts of Lemma 1.

Lemma 4. *Let $r, z \in \mathbb{Z}_q$ with $\|z\|_\infty \leq \alpha/2$. Then*

$$\mathsf{UseHint}_q\left(\mathsf{MakeHint}_q(z, r, \alpha), r, \alpha\right) = \mathsf{HighBits}_q(r + z, \alpha).$$

Proof. The output of $\mathsf{Decompose}_q$ is an integer r_1 such that $0 \leq r_1 < (q-1)/\alpha$ and another integer r_0 such that $\|r_0\|_\infty \leq \alpha/2$. Because $\|z\|_\infty \leq \alpha/2$, the integer $v_1 := \mathsf{HighBits}_q(r + z, \alpha)$ either stays the same as r_1 or becomes $r_1 \pm 1$ modulo $m = (q-1)/\alpha$. More precisely, if $r_0 > 0$, then $-\alpha/2 < r_0 + z \leq \alpha$. This implies that v_1 is either r_1 or $r_1 + 1 \bmod m$. If $r_0 \leq 0$, then we have $-\alpha \leq r_0 + z \leq \alpha/2$. In this case, we have $v_1 = r_1$ or $r_1 - 1 \bmod m$.

The $\mathsf{MakeHint}_q$ routine checks whether $r_1 = v_1$ and outputs 0 if this is so, and 1 if $r_1 \neq v_1$. The $\mathsf{UseHint}_q$ routine uses the "hint" h to either output r_1 (if $y = 0$) or, depending on whether $r_0 > 0$ or not, output either $r_1 + 1 \bmod^+ m$ or $r_1 - 1 \bmod^+ m$.

The lemma below shows that r is not too far away from the output of the $\mathsf{UseHint}_q$ algorithm. This will be necessary for the security of the scheme.

Lemma 5. *Let $(h, r) \in \{0,1\} \times \mathbb{Z}_q$ and let $v_1 = \mathsf{UseHint}_q(h, r, \alpha)$. If $h = 0$, then $\|r - v_1 \cdot \alpha\|_\infty \leq \alpha/2$; else $\|r - v_1 \cdot \alpha\|_\infty \leq \alpha + 1$.*

Proof. Let $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$. We go through all three cases of the $\mathsf{UseHint}_q$ procedure.

Case 1 ($h = 0$): We have $v_1 = r_1$ and

$$r - v_1 \cdot \alpha = r_1 \cdot \alpha + r_0 - r_1 \cdot \alpha = r_0\,,$$

which by definition has absolute value at most $\alpha/2$.

Case 2 ($h = 1$ and $r_0 > 0$): We have $v_1 = r_1 + 1 - \kappa \cdot (q-1)/\alpha$ for $\kappa = 0$ or 1. Thus

$$\begin{aligned} r - v_1 \cdot \alpha &= r_1 \cdot \alpha + r_0 - (r_1 + 1 - \kappa \cdot (q-1)/\alpha) \cdot \alpha \\ &= -\alpha + r_0 + \kappa \cdot (q-1). \end{aligned}$$

After centered reduction modulo q, the latter has magnitude $\leq \alpha$.

Case 3 ($h = 1$ and $r_0 \leq 0$): We have $v_1 = r_1 - 1 + \kappa \cdot (q-1)/\alpha$ for $\kappa = 0$ or 1. Thus

$$\begin{aligned} r - v_1 \cdot \alpha &= r_1 \cdot \alpha + r_0 - (r_1 - 1 + \kappa \cdot (q-1)/\alpha) \cdot \alpha \\ &= \alpha + r_0 - \kappa \cdot (q-1). \end{aligned}$$

After centered reduction modulo q, the latter quantity has magnitude $\leq \alpha + 1$.

The next lemma will play a role in proving the strong existential unforgeability of our signature scheme. It states that two different h, h' cannot lead to $\mathsf{UseHint}_q(h, r, \alpha) = \mathsf{UseHint}_q(h', r, \alpha)$.

Lemma 6. *Let $r \in \mathbb{Z}_q$ and $h, h' \in \{0, 1\}$. If $\mathsf{UseHint}_q(h, r, \alpha) = \mathsf{UseHint}_q(h', r, \alpha)$, then $h = h'$.*

Proof. Note that $\mathsf{UseHint}_q(0, r, \alpha) = r_1$ and $\mathsf{UseHint}_q(1, r, \alpha)$ is equal to $(r_1 \pm 1) \bmod^+ (q-1)/\alpha$. Since $(q-1)/\alpha \geq 2$, we have that $r_1 \neq (r_1 \pm 1) \bmod^+ (q-1)/\alpha$.

We now prove Lemma 2 and Lemma 3.

Proof. (Of Lemma 2) We prove the lemma for integers, rather than vectors of polynomials, since the $\mathsf{HighBits}$ function works independently on each coefficient. If $\|\mathsf{LowBits}_q(r, \alpha)\|_\infty < \alpha/2 - \beta$, then $r = r_1 \cdot \alpha + r_0$ where $-\alpha/2 + \beta < r_0 \leq \alpha/2 + \beta$. Then $r + s = r_1 \cdot \alpha + (r_0 + s)$ and $-\alpha/2 < r_0 + s \leq \alpha/2$. Therefore $r + s \bmod^\pm \alpha = r_0 + s$, and thus

$$(r + s) - ((r + s) \bmod^\pm \alpha) = r_1 \cdot \alpha = r - (r \bmod^\pm \alpha),$$

and the claim in the Lemma follows. □

Proof. (Of Lemma 3) We first prove the forward direction. By the same proof as inLemma 2, since $\|r_0 + s\|_\infty < \alpha/2$, we know that $w_1 = r_1$. We can therefore write $(r+s) = r_1 \cdot \alpha + r_0 + s$. Since $\|r_0 + s\|_\infty < \alpha$, we know that

$$w_0 = \mathsf{LowBits}_q(r + s, \alpha) = r_1 \cdot \alpha + r_0 + s \bmod^\pm \alpha = r_0 + s.$$

To prove the other direction, write

$$r_1 \cdot \alpha + r_0 + s = r + s = w_1 \cdot \alpha + w_0 = r_1 \cdot \alpha + w_0,$$

and therefore $r_0 + s = w_0$. □

B Zero-Knowledge Proof

The security of our scheme does not rely on the part of the public key $\mathbf{t}_0$ being secret and so we will be assuming that the public key is $\mathbf{t}$ rather than $\mathbf{t}_1$.

We want to first compute the probability that some particular $(\mathbf{z}, c)$ is generated in Step 19 taken over the randomness of $\mathbf{y}$ and the random oracle H which is modeled as a random function. We have

$$\Pr[\mathbf{z}, c] = \Pr[c] \cdot \Pr[\mathbf{y} = \mathbf{z} - c\mathbf{s}_1 \mid c].$$

Whenever $\mathbf{z}$ has all its coefficients less than $\gamma_1 - \beta$ then the above probability is exactly the same for every such tuple $(\mathbf{z}, c)$. This is because $\|c\mathbf{s}_i\|_\infty \leq \beta$, and thus $\|\mathbf{z} - c\mathbf{s}_1\|_\infty \leq \gamma_1 - 1$, which is a valid value of $\mathbf{y}$. Therefore, if we only output $\mathbf{z}$ when all its coefficients have magnitudes less than $\gamma_1 - \beta$, then the resulting distribution will be uniformly random over $S^\ell_{\gamma_1 - \beta - 1} \times B_\tau$.

The simulation of the signature follows [Lyu12, BG14]. The simulator picks a uniformly random $(\mathbf{z}, c)$ in $S^{\ell}_{\gamma_1-\beta-1} \times B_\tau$, after which it also makes sure that

$$\|\mathbf{r}_0\|_\infty = \|\mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta.$$

By Equation (2), we know that $\mathbf{w} - c\mathbf{s}_2 = \mathbf{A}\mathbf{z} - c\mathbf{t}$, and therefore the simulator can perfectly simulate this as well.

If $\mathbf{z}$ does indeed satisfy $\|\mathsf{LowBits}_q(\mathbf{w}-c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2-\beta$, then as long as $\|c\mathbf{s}_2\|_\infty \leq \beta$ (which is how we set β), we will have

$$\mathbf{r}_1 = \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2) = \mathbf{w}_1.$$

We can then program

$$\mathrm{H}(\mu \parallel \mathbf{w}_1) \leftarrow c\,.$$

Unless we have already set the value of $\mathrm{H}(\mu \parallel \mathbf{w}_1)$ to something else (which only happens with negligible probability), the resulting pair $(\mathbf{z}, c)$ has the same distribution as in a genuine signature of μ.

All the other steps (after Step 21) of the signing algorithm are performed using public information and are therefore simulatable.

C Concrete Security

C.1 Lattice Reduction and Core-SVP Hardness

The best known algorithm for finding very short non-zero vectors in Euclidean lattices is the Block–Korkine–Zolotarev algorithm (BKZ) [SE94], proposed by Schnorr and Euchner in 1991. More recently, it was proven to quickly converge to its fix-point [HPS11] and improved in practice [CN11a]. Yet, what it achieves asymptotically remains unchallenged.

BKZ with block-size b makes calls to an algorithm that solves the Shortest lattice Vector Problem (SVP) in dimension b. The security of our scheme relies on the necessity to run BKZ with a large block-size b and the fact that the cost of solving SVP is exponential in b. The best known classical SVP solver [BDGL16b] runs in time $\approx 2^{c_C \cdot b}$ with $c_C = \log_2 \sqrt{3/2} \approx 0.292$. The best known quantum SVP solver [Laa15, Sec. 14.2.10] runs in time $\approx 2^{c_Q \cdot b}$ with $c_Q = \log_2 \sqrt{13/9} \approx 0.265$. One may hope to improve these run-times, but going below $\approx 2^{c_P \cdot b}$ with $c_P = \log_2 \sqrt{4/3} \approx 0.2075$ would require a theoretical breakthrough. Indeed, the best known SVP solvers rely on covering the b-dimensional sphere with cones of center-to-edge angle $\pi/3$: this requires $2^{c_P \cdot b}$ cones. The subscripts C, Q, P respectively stand for Classical, Quantum and Paranoid.

The strength of BKZ increases with b. More concretely, given as input a basis $(\mathbf{c}_1, \ldots, \mathbf{c}_n)$ of an n-dimensional lattice, BKZ repeatedly uses the b-dimensional SVP-solver on lattices of the form $(\mathbf{c}_{i+1}(i), \ldots, \mathbf{c}_j(i))$ where $i \leq n$, $j = \min(n, i + b)$ and where $\mathbf{c}_k(i)$ denotes the projection of $\mathbf{c}_k$ orthogonally to the vectors $(\mathbf{c}_1, \ldots, \mathbf{c}_i)$. The effect of these calls is to flatten the curve of the $\ell_i = \log_2 \|\mathbf{c}_i(i-1)\|$'s (for $i = 1, \ldots, n$). At the start of the execution, the ℓ_i's typically decrease fast, at least locally. As BKZ preserves the determinant of the $\mathbf{c}_i$'s, the sum of the ℓ_i's remains constant throughout the execution, and after a (small) polynomial number of SVP calls, BKZ has made the ℓ_i's decrease less. It can be heuristically estimated that for sufficiently large b, the local slope of the ℓ_i's converges to

$$\mathit{slope}(b) = \frac{1}{b-1} \log_2 \left(\frac{b}{2\pi \mathrm{e}} (\pi \cdot b)^{1/b} \right),$$

unless the local input ℓ_i's are already too small or too large. The quantity $\mathit{slope}(b)$ decreases with b, implying that the larger b the flatter the output ℓ_i's.

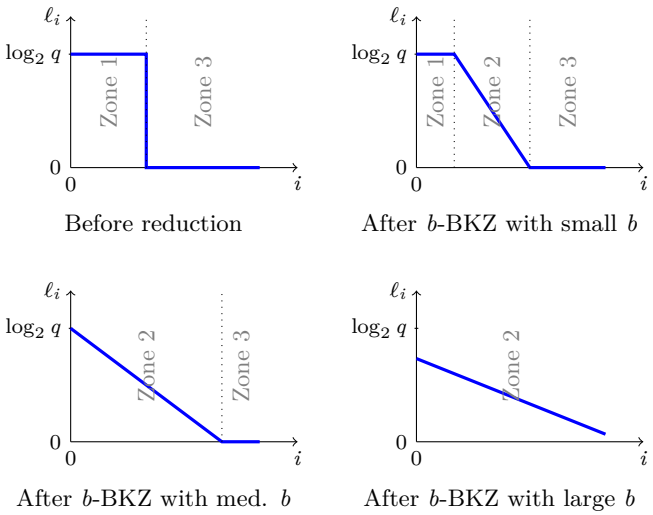

Figure 11: Evolution of Gram-Schmidt length in log-scale under BKZ reduction for various blocksizes. The area under the curves remains constant, and the slope in Zone 2 decrease with the blocksize. Note that Zone 3 may disappear before Zone 1, depending on the shape of the input basis.

In our case, the input ℓ_i's are of the following form (cf. Fig. 11). The first ones are all equal to $\log_2 q$ and the last ones are all equal to 0. BKZ will flatten the jump, decreasing ℓ_i's with small i's and increasing ℓ_i's with large i's. However, the local slope $slope(b)$ may not be sufficiently small to make the very first ℓ_i's decrease and the very last ℓ_i's increase. Indeed, BKZ will not increase (resp. increase) some ℓ_i's if these are already smaller (resp. larger) than ensured by the local slope guarantee. In our case, the ℓ_i's are always of the following form at the end of the execution:

- The first ℓ_i's are constant equal to $\log_2 q$ (this is the possibly empty Zone 1).
- Then they decrease linearly, with slope $slope(b)$ (this is the never-empty Zone 2).
- The last ℓ_i's are constant equal to 0 (this is the possibly empty Zone 3).

The graph is continuous, i.e., if Zone 1 (resp. Zone 3) is not empty, then Zone 2 starts with $\ell_i = \log_2 q$ (resp. ends with $\ell_i = 0$).

C.2 Solving MLWE

Any $\mathsf{MLWE}_{\ell,k,D}$ instance for some distribution D can be viewed as an LWE instance of dimensions $256{\cdot}\ell$ and $256{\cdot}k$. Indeed, the above can be rewritten as finding $\mathrm{vec}(\mathbf{s}_1), \mathrm{vec}(\mathbf{s}_2) \in \mathbb{Z}^{256\cdot\ell} \times \mathbb{Z}^{256\cdot k}$ from $(\mathrm{rot}(\mathbf{A}), \mathrm{vec}(\mathbf{t}))$, where $\mathrm{vec}(\cdot)$ maps a vector of ring elements to the vector obtained by concatenating the coefficients of its coordinates, and $\mathrm{rot}(\mathbf{A}) \in \mathbb{Z}_q^{256\cdot k \times 256\cdot \ell}$ is obtained by replacing all entries $a_{ij} \in R_q$ of $\mathbf{A}$ by the 256×256 matrix whose z-th column is $\mathrm{vec}\left(x^{z-1} \cdot a_{ij}\right)$.

Given an LWE instance, there are two lattice-based attacks. The primal attack and the dual attack. Here, the primal attack consists in finding a short non-zero vector in the lattice $\Lambda = \{\mathbf{x} \in \mathbb{Z}^d : \mathbf{Mx} = \mathbf{0} \bmod \mathbf{q}\}$ where $\mathbf{M} = (\mathrm{rot}(\mathbf{A})_{[1:m]} | \mathbf{I}_m | \mathrm{vec}(\mathbf{t})_{[1:m]})$ is an $m \times d$ matrix where $d = 256 \cdot \ell + m + 1$ and $m \leq 256 \cdot k$. Indeed, it is sometime not optimal to use all the given equations in lattice attacks.

We tried all possible number m of rows, and, for each trial, we increased the blocksize of b until the value ℓ_{d-b} obtained as explained above was deemed sufficiently large. As explained

in [ADPS16, Sec. 6.3], if $2^{\ell_{d-b}}$ is greater than the expected norm of $(\mathrm{vec}(\mathbf{s}_1), \mathrm{vec}(\mathbf{s}_2))$ after projection orthogonally to the first $d-b$ vectors, it is likely that the MLWE solution can be easily extracted from the BKZ output.

The dual attack consists in finding a short non-zero vector in the lattice $\Lambda' = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}^m \times \mathbb{Z}^d : \mathbf{M}^T\mathbf{x} + \mathbf{y} = 0 \bmod q)\}$, $\mathbf{M} = (\mathrm{rot}(\mathbf{A})_{[1:m]})$ is an $m \times d$ matrix where $d = 256 \cdot \ell$ and $m \leq 256 \cdot k$. Again, for each value of m, we increased the value of b until the value ℓ_1 obtained as explained above was deemed sufficiently small according the analysis of [ADPS16, Sec. 6.3].

C.3 Solving MSIS and SelfTargetMSIS

As per the discussion in Section 6.2.1, the best known attack against the SelfTargetMSIS problem involves either breaking the security of H or solving the problem in Eq. (13). The latter amounts to solving the $\mathsf{MSIS}_{k,\ell+1,\zeta}$ problem for the matrix $[\ \mathbf{A} \mid \mathbf{t}'\ ]$. [11]

Note that the MSIS instance can be mapped to a $\mathsf{SIS}_{256\cdot k,256\cdot(\ell+1),\zeta}$ instance by considering the matrix $\mathrm{rot}(\mathbf{A}|\mathbf{t}') \in \mathbb{Z}_q^{256\cdot k \times 256\cdot(\ell+1)}$. The attack against the $\mathsf{MSIS}_{k,\ell,\zeta'}$ instance in Eq. (6) can similarly be mapped to a $\mathsf{SIS}_{256\cdot k,256\cdot \ell,\zeta'}$ instance by considering the matrix $\mathrm{rot}(\mathbf{A}) \in \mathbb{Z}_q^{256\cdot k \times 256\cdot \ell}$. The attacker may consider a subset of w columns, and let the solution coefficients corresponding to the dismissed columns be zero.

Remark 1. An unusual aspect here is that we are considering the infinity norm, rather than the Euclidean norm. Further, for our specific parameters, the Euclidean norms of the solutions are above q. In particular, the vector $(q, 0, \ldots, 0)^T$ belongs to the lattice, has Euclidean norm below that of the solution, but its infinity norm above the requirement. This raises difficulties in analyzing the strength of BKZ towards solving our infinity norm SIS instances: indeed, even with small values of b, the first ℓ_i's are short (they correspond to q-vectors), even though they are not solutions.

For each number w of selected columns and for each value of b, we compute the estimated BKZ output ℓ_i's, as explained above. We then consider the smallest i such that ℓ_i is below $\log_2 q$ and the largest j such that ℓ_j above 0. These correspond to the vectors that were modified by BKZ, with smallest and largest indices, respectively. In fact, for the same cost as a call to the SVP-solver, we can obtain $\sqrt{4/3}^b$ vectors with Euclidean norm $\approx 2^{\ell_i}$ after projection orthogonally to the first $i-1$ basis vectors. Now, let us look closely at the shape of such a vector. As the first $i-1$ basis vectors are the first $i-1$ canonical unit vectors multiplied by q, projecting orthogonally to these consists in zeroing the first $i-1$ coordinates. The remaining $w-i+1$ coordinates have total Euclidean norm $\approx 2^{\ell_i} \approx q$, and the last $w-j$ coordinates are 0. We heuristically assume that these coordinates have similar magnitudes $\sigma \approx 2^{\ell_i}/\sqrt{j-i+1}$; we model each such coordinate as a Gaussian of standard deviation σ. We assume that each one of our $\sqrt{4/3}^b$ vectors has its first $i-1$ coordinates independently uniformly distributed modulo q, and finally compute the probability that all coordinates in both ranges $[0, i-1]$ and $[i, j]$ are less than B in absolute value. Our cost estimate is the inverse of that probability multiplied by the run-time of our b-dimensional SVP-solver.

Forgetting q-vectors. For all the parameter sets in Table 1 and Table 2, the best parametrization of the attack above kept the basis in a shape with a non-trivial Zone 1. We note that the coordinates in this range have a quite lower probability of passing the ℓ_∞ constraint than coordinates in Zone 2. We therefore considered a strategy consisting of "forgetting" the q-vectors, by re-randomizing the input basis before running the BKZ

[11] Note that a solution to Eq. (13) would require the coefficient in from of $\mathbf{t}'$ to be ± 1, while we're allowing any small polynomial. Furthermore, as discussed after Eq. (13), some parts of the real solution are smaller than the bound ζ, but we're ignoring this for the sake of being conservative with our analysis.

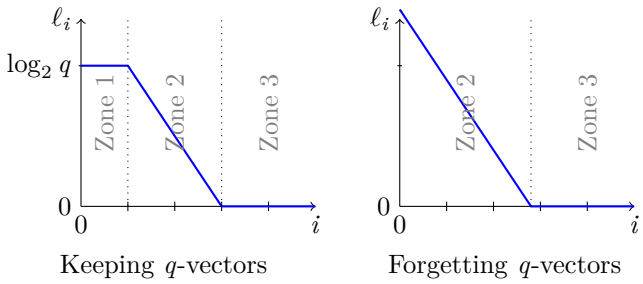

Figure 12: Effect of forgetting q-vectors by randomization, under the same BKZ-blocksize b.

algorithm. For the same blocksize b, this makes Zone 1 of the output basis disappear (BKZ does not find the q-vectors), at the cost of producing a basis with first vectors of larger Euclidean norms. This is depicted in Fig. 12.

It turns out that this strategy always improves over the previous strategy for the parameter ranges considered in Table 2. We therefore used this strategy for our security estimates.

C.4 On Other Attacks

For our parameters, the BKW [BKW03] and Arora–Ge [AG11] families of algorithms are far from competitive.

Algebraic attacks. One specificity of our LWE and SIS instances is that they are inherited from MLWE and MSIS instances. One may wonder whether the extra algebraic structure of the resulting lattices can be exploited by an attacker. The line of work of [CGS14, BS16, CDPR16, CDW17] did indeed find new cryptanalytic results on certain algebraic lattices, but [CDW17] mentions serious obstacles towards breaking cryptographic instances of Ring-LWE. By switching from Ring-LWE to MLWE, we get even further away from those weak algebraic lattice problems.

Dense sublattice attacks. Kirchner and Fouque [KF17] showed that the existence of many linearly independent and unexpectedly short lattice vectors (much shorter than Minkowski's bound) helps BKZ run better than expected in some cases. This could happen for our primal LWE attack, by extending $\mathbf{M} = (\mathrm{rot}(\mathbf{A})_{[\mathbf{1:m}]}|\mathbf{I_m}|\mathrm{vec}(\mathbf{t})_{[\mathbf{1:m}]})$ to $(\mathrm{rot}(\mathbf{A})_{[1:m]}|\mathbf{I}_m|\mathrm{rot}(\mathbf{t})_{[1:m]})$: the associated lattice now has 256 linearly independent short vectors rather than a single one. The Kirchner-Fouque analysis of BKZ works best if both q and the ratio between the number of unexpectedly short vectors and the lattice dimension are high. In the NTRU case, for example, the ratio is $1/2$, and, for some schemes derived from NTRU, the modulus q is also large. We considered this refined analysis of BKZ in our setup, but, to become relevant for our parameters, it requires a parameter b which is higher than needed with the usual analysis of BKZ. Note that [KF17] also arrived to the conclusion that this attack is irrelevant in the small modulus regime, and is mostly a threat to fully homomorphic encryption schemes and cryptographic multilinear maps.

Note that, once again, the switch from Ring-LWE to MLWE takes us further away from lattices admitting unconventional attacks. Indeed, the dimension ratio of the dense sub-lattice is $1/2$ in NTRU, at most $1/3$ in lattices derived from Ring-LWE, and at most $1/(\ell+2)$ in lattices derived from MLWE.

Specialized attack against ℓ_∞-SIS. At last, we would like to mention that it is not clear whether the attack sketched in Appendix C.3 above for SIS in infinity norm is optimal. Indeed, as we have seen, this approach produces many vectors, with some rather large uniform coordinates (at indices $1, \ldots, i$), and smaller Gaussian ones (at indices $i, \ldots, j$). In our current analysis, we simply hope that one of the vector satisfies the ℓ_∞ bound. Instead, one could combine them in ways that decrease the size of the first (large) coefficients, while letting the other (small) coefficients grow a little bit.

This situation created by the use of ℓ_∞-SIS (see Remark 1) has — to the best of our knowledge — not been studied in detail. After a preliminary analysis, we do not consider such an improved attack a serious threat to our concrete security claims, especially in light of the approximations already made in favor of the adversary.

C.5 Beyond Core-SVP Hardness

At the time the core-SVP hardness measure was introduced by [ADPS16], the best implementations of sieving [BDGL16a, MLB17] had performance significantly worse than the $2^{.292b}$ CPU cycles proposed as a conservative estimate by this methodology. This was due to substantial polynomial, or even sub-exponential, overheads hidden in the complexity analysis of $2^{.292b+o(b)}$ given in [BDGL16a]. Before the work of [ADPS16], security estimates of lattice schemes were typically based on the cost of SVP via enumeration given in [CN11b, APS15], leading to much more aggressive parameters. Beyond the cost of SVP-calls, this methodology also introduced a different prediction of when BKZ solves LWE which was later confirmed in [AGVW17] and refined in [DSDGR20].

While doubts were expressed to whether sieving would ever outperform the super-exponential, yet practically smaller, costs of enumeration [CN11b] for relevant cryptographic dimensions, significant progress on sieving algorithms [Duc18, ADH+19] has brought the cross-over point down to dimension about $b = 80$. In fact, the current SVP records are now held by algorithms that employ sieving[12]. These progresses mandates a revision and refinement of Dilithium security estimates, especially regading classical attacks. In particular, while it was pretty clear from experiments than the costs hidden in the $o(b)$ before those improvement were positive both in practice and asymptotically, the dimensions for free technique of [Duc18] offers a sub-exponential speed-up, making it a priori unclear whether the total $o(b)$ term is positive or negative, both asymptotically and concretely.

C.5.1 Cost of MLWE

We follow the same methodology as for the Kyber Round 3 specification document to count the number of gates required to solve MLWE. This analysis refines the core-SVP methodology in the following ways:

- It uses the probabilistic simulation of [DSDGR20] rather the the GSA-intersect model of [ADPS16, AGVW17] to determine the BKZ blocksize b for a successful attack. The required blocksize is somewhat larger (by an added term between 10 and 20 for our parameters), because of a "tail" phenomenon [YD17],
- It accounts for the "few dimensions for free" introduced in [Duc18], which permits to solve SVP in dimension β while running sieving in a somewhat smaller dimension $b' = b - o(b)$,
- It relies on the concrete estimation for the cost of sieving in gates from [AGPS19]
- It accounts for the number of calls to the SVP oracle,

[12] https://www.latticechallenge.org/svp-challenge/

- It dismisses the dual attack as realistically more expensive than the primal one, noting that the analysis of [ADPS16] (also used here) by assuming that the sieve provides $2^{.208b}$ many vectors as short as the shortest one. First, most of those vectors are larger by a factor $\sqrt{4/3}$, secondly the trick of exploiting all those vectors is not compatible with the "dimension for free" trick of [Duc18].

The scripts for these refined estimates are provided in a git branch of the leaky-LWE-estimator of [DSDGR20],[13] and lead to the estimates given in Table 4. We refer to the Kyber Round 3 specification document for the details of this analysis. We point out that it is paired with a detailed discussion of the "known unknowns", providing a plausible confidence interval for these estimates. For the classical hardness of LWE at Levels 1 and 2, it is estimated that the true cost is no more than a 16 bits away from this estimate, in either direction.

We also note that a similar refined count of quantum gates seems essentially irrelevant: the work of [AGPS19] concluded that quantum speed-up of sieving are rather tenuous, while the quantum security target for each level is significantly lower that the classical target.

Table 4: Refined estimates for the MLWE hardness

	n	b	b'	$\log_2$(gates)	$\log_2$(memory in bits)
Dilithium Level 2	2049	433	394	158.6	97.8
Dilithium Level 3	2654	638	587	216.7	138.7
Dilithium Level 5	3540	883	818	285.4	187.4

n: Optimal lattice dimension for the attack
b: BKZ blocksize
b': Sieving dimension accounting for "dimensions for free"

C.5.2 Cost of MSIS

Ideally, one would also like to have similar refined estimates for the MSIS problem in the ℓ_∞-norm. However this requires further technical considerations. We strongly suspect that the refined cost of MSIS would be somewhat larger than of LWE, with the main reason being, as for the MLWE dual attack, the trick of exploiting all the vectors outputted by sieving is not compatible with the "dimension for free" trick of [Duc18]. We plan to provide such a refined analysis in the near future.

[13] https://github.com/lducas/leaky-LWE-Estimator/tree/NIST-round3