CRYSTALS-Dilithium

Algorithm Specifications and Supporting Documentation

Léo Ducas[1], Eike Kiltz[2], Tancrède Lepoint[3], Vadim Lyubashevsky[4], Peter Schwabe[5], Gregor Seiler[6] and Damien Stehlé[7]

[1] CWI, Netherlands
[2] Ruhr Universität Bochum, Germany
[3] SRI International, USA
[4] IBM Research – Zurich, Switzerland
[5] Radboud University, Netherlands
[6] IBM Research – Zurich, Switzerland
[7] ENS de Lyon, France

November 30, 2017

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 in [Lyu09, GLP12, BG14]. 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 that require virtually as much space as time. 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 almost trivial 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] 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 essentially 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.

<u>Gen</u>
$\mathbf{A} \leftarrow R_q^{k\times\ell}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
return $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$

<u>$\mathsf{Sign}(sk, M)$</u>
$\mathbf{z} := \perp$
while $\mathbf{z} = \perp$ do
  $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$
  $\mathbf{w}_1 := \mathsf{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$
  $c \in B_{60} := \mathsf{H}(M \parallel \mathbf{w}_1)$
  $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathsf{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$, then $\mathbf{z} := \perp$
return $\sigma = (\mathbf{z}, c)$

<u>$\mathsf{Verify}(pk, M, \sigma = (\mathbf{z}, c))$</u>
$\mathbf{w}'_1 := \mathsf{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$
if return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![c = \mathsf{H}(M \parallel \mathbf{w}'_1)]\!]$

Figure 1: Template for our signature scheme.

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 60 ± 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 $> 2^{256}$. 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 60 ± 1's and the maximum coefficient in $\mathbf{s}_i$ is η, it's easy to see that $\beta \leq 60\eta$. 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 (in our instantiations, this number is between 4 and 7).

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 easily fixed) inefficiency is that the public key consists of a matrix of $k \cdot \ell$ polynomials, which could have a rather large representation. The obvious fix is to have $\mathbf{A}$ generated from some seed ρ using SHAKE-128, and this is a standard technique. The novelty of Dilithium over the previous schemes is that we also shrink the size of the public key by a factor of 2.5 at the expense of increasing the signature by around 150 bytes. For the recommended security level, the scheme has 2.7KB signatures and 1.5KB public keys.

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 make 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. [KLS17] 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. 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 of such polynomials. In our recommended parameter setting, $\mathbf{A}$ is a 5×4 matrix and therefore consists of 20 polynomials. Thus the multiplication $\mathbf{Av}$ involves 20 polynomial multiplications. As in most 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 NTT, we needed 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 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 a different approach than other fast implementations in that we use integer 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.

1.3 Comparisons to Other Post-Quantum Signature Schemes

We now give a brief comparison between our signature scheme and other post-quantum signature schemes that we are aware of.

1.3.1 Lattice Schemes

Schemes with Smaller Signature Sizes. The lattice-based digital signatures (with security reductions) that have the smallest signature sizes are [DDLL13, DLP14], which are based on the NTRU assumption. If one adjusts the parameters of [DDLL13] and [DLP14] so that the security is comparable to that of Dilithium[1], the signature sizes will be approximately 1.5KB and 1KB respectively (compared to 2.7KB in Dilithium), while the public key sizes will remain approximately the same as in Dilithium.

The main down-side of [DDLL13, DLP14] is that they intrinsically require the use of (discrete) Gaussian sampling in order to be efficient, which creates several serious real-world issues. The first is that it is not straight-forward to construct a constant-time discrete Gaussian sampler. The second is that mistakes in sampling (discrete) Gaussians are extremely hard to detect in testing; yet even small deviations from the right distribution can lead to complete signing key recovery by the adversary. For this reason, we believe that a universally-used scheme should be very simple to implement in a secure fashion, and we have thus eschewed using anything other than uniform sampling in Dilithium.

Schemes with (Quantum) Security Reductions from (Ring / Module)-LWE Only. The security of Dilithium is, in the quantum random oracle model (QROM), tightly based on the hardness of the standard MLWE and MSIS problems, as well as a "hybrid" SelfTargetMSIS problem that was defined in [KLS17]. For the latter problem, there is a reduction MSIS $\leq$ SelfTargetMSIS in the random oracle model via the forking lemma, but not in the quantum random oracle model.

One could construct a version of Dilithium whose security is based entirely on the hardness of Ring-LWE in the QROM, but this scheme would result in a 5X increase in the public key size and a 2X increase in the signature size [KLS17, Table 1]. Furthermore, we would not be able to work over a the ring that supports NTT, which would make signing and verification slower. As we explain in Section 5, the SelfTargetMSIS problem is in fact the lattice version of a problem upon which *tight* security proofs of today's signatures (e.g. Schnorr) using the Fiat-Shamir transform are based. We therefore believe that this assumption is a perfectly sound one to make and avoiding it is not worth the significant cost in speed and output size.

[1] The security in [DDLL13, DLP14] was not set as conservatively as for Dilithium – in particular, sieving attacks were not considered.

1.3.2 Other Post-Quantum Signatures.

Of all the non-lattice post-quantum schemes that we are aware of, Dilithium has the smallest combination of public key and signature size. There may be scenarios, however, that call for one of these values to be very small, while not caring too much about the other. If one would like to minimize the public key size, then hash-based signatures (e.g. [BHH+15]) are a good option because the public key is less than a hundred bytes. The signature length of such signatures, on the other hand, is between 30-40KB, and signing time is around 50X slower than Dilithium. If, on the other hand, one would like to minimize the signature size, then multivariate schemes (see the various comparisons in [CLP+17]) may be of interest. The signature sizes in these schemes are less than one hundred bytes and signing time is noticeably faster. The public keys, on the other hand, are often larger than 100KB.

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 q.[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 $\|w\|_\infty \leq \eta$.

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

[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.

$\mathbb{Z}_q[X]/(X - r^i) \cong \mathbb{Z}_q$. 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 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 4.5.

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 4.2.

Hashing to a Ball. Let B_h denote the set of elements of R that have h coefficients that are either -1 or 1 and the rest are 0. We have $|B_h| = 2^h \cdot \binom{n}{h}$. For our signature scheme, we will need a cryptographic hash function that hashes onto B_{60} (which has more than 2^{256} elements). The algorithm we will use to create a random element in B_{60} 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]

SampleInBall
Initialize $\mathbf{c} = c_0 c_1 \ldots c_{255} = 00\ldots0$
for $i := 196$ to 255
    $j \leftarrow \{0, 1, \ldots, i\}$
    $s \leftarrow \{0, 1\}$
    $c_i := c_j$
    $c_j := (-1)^s$
return $\mathbf{c}$

Figure 2: Create a random 256-element array with 60 ± 1's and 196 $0's$

[3]Normally, the algorithm should begin at $i = 0$, but since there are 196 0's, the first 195 iterations would just be setting components of $\mathbf{c}$ to 0.

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 $\hat{\mathbf{A}} \in \mathbb{Z}_q^{256}$, 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, 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 y. The function ExpandMask, used for deterministically generating the randomness of the signature scheme, maps $K \parallel \mu \parallel \kappa$ to $y \in S_{\gamma_1-1}^{l}$.

Collision resistant hash. The function CRH used in our signature scheme is a collision resistant hash function mapping to $\{0,1\}^{384}$.

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 crucial 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 $\mathbf{h}, \mathbf{h}'$*

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

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

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

$\underline{\mathsf{Decompose}_q(r, \alpha)}$
$r := r \bmod^{+} q$
$r_0 := r \bmod^{\pm} \alpha$
$\textbf{if } r - r_0 = q - 1$
$\quad \textbf{then } r_1 := 0;\ r_0 := r_0 - 1$
$\textbf{else } r_1 := (r - r_0)/\alpha$
$\textbf{return } (r_1, r_0)$

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

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

Figure 3: Supporting algorithms for Dilithium.

be vectors of bits. Then the $\mathsf{HighBits}_q$, $\mathsf{MakeHint}_q$, *and* $\mathsf{UseHint}_q$ *algorithms satisfy the following properties:*

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*

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

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 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 reduced by a factor of around 2.5 at the expense of an additional hundred 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 $\lceil \log q \rceil - d$ bits. In our instantiation, $q \approx 2^{23}$ and $d = 14$, which means that instead of 23 bits in each public key coefficient, there are instead 9.

The main problem with 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 $\mathbf{Az} - c\mathbf{t}_1 \cdot 2^d = \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0$. But since the product $c\mathbf{t}_0$ consists of only small numbers, and we only care about the high order bits, we really only need to know the carries that each coefficient of $c\mathbf{t}_0$ causes. These are the carries that the signer sends 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 (this is the extra bytes in the signature), 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 $\hat{A}$) 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 ρ and K, and the random seed used to create $\mathbf{s}_1, \mathbf{s}_2$ in the key generation algorithm. All the other parts of the secret key can be recreated from these. 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.

Another possible change is to remove the strict deterministic nature of the digital signature. One may want to consider this option due to the recent side-channel attacks that exploit determinism [SBB+17, PSS+17]. An easy way in which to give an option of using randomized signatures is to allow the appending of some system randomness to the input of ExpandMask when generating $\mathbf{y}$. As we mentioned earlier, the security proof for Dilithium is "tight" according to [KLS17] for deterministic signatures and the bound gets gradually looser the more different signatures are seen per message. We therefore still recommend using deterministic signatures except in environments that may be vulnerable to the aforementioned side-channel attacks.

3.2 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

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

Furthermore, because the signer also checks in Line 19 that $\mathbf{r}_1 = \mathbf{w}_1$, this is equivalent to

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

Therefore, the $\mathbf{w}_1$ computed by the verifier is the same as that of the signer, and the verification procedure will always accept.

<u>Gen</u>
01 $\rho \leftarrow \{0,1\}^{256}$
02 $K \leftarrow \{0,1\}^{256}$
03 $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
04 $\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ $\quad\triangleright$ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$
05 $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ $\quad\triangleright$ Compute $\mathbf{A}\mathbf{s}_1$ as $\mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{s}_1))$
06 $(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t}, d)$
07 $tr \in \{0,1\}^{384} := \mathsf{CRH}(\rho \parallel \mathbf{t}_1)$
08 **return** $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

<u>$\mathsf{Sign}(sk, M)$</u>
09 $\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ $\quad\triangleright$ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$
10 $\mu \in \{0,1\}^{384} := \mathsf{CRH}(tr \parallel M)$
11 $\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$
12 **while** $(\mathbf{z}, \mathbf{h}) = \perp$ do $\quad\triangleright$ Pre-compute $\hat{\mathbf{s}}_1 := \mathtt{NTT}(\mathbf{s}_1)$, $\hat{\mathbf{s}}_2 := \mathtt{NTT}(\mathbf{s}_2)$, and $\hat{\mathbf{t}}_0 := \mathtt{NTT}(\mathbf{t}_0)$
13 $\quad\mathbf{y} \in S_{\gamma_1-1}^\ell := \mathsf{ExpandMask}(K \parallel \mu \parallel \kappa)$
14 $\quad\mathbf{w} := \mathbf{A}\mathbf{y}$ $\quad\triangleright$ $\mathbf{w} := \mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{y}))$
15 $\quad\mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$
16 $\quad c \in B_{60} := \mathsf{H}(\mu \parallel \mathbf{w}_1)$ $\quad\triangleright$ Store c in NTT representation as $\hat{c} = \mathtt{NTT}(c)$
17 $\quad\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ $\quad\triangleright$ Compute $c\mathbf{s}_1$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_1)$
18 $\quad(\mathbf{r}_1, \mathbf{r}_0) := \mathsf{Decompose}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$ $\quad\triangleright$ Compute $c\mathbf{s}_2$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_2)$
19 $\quad$**if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$, then $(\mathbf{z}, \mathbf{h}) := \perp$
20 $\quad$**else**
21 $\quad\quad\mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$ $\quad\triangleright$ Compute $c\mathbf{t}_0$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{t}}_0)$
22 $\quad\quad$**if** $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω, then $(\mathbf{z}, \mathbf{h}) := \perp$
23 $\quad\kappa := \kappa + 1$
24 **return** $\sigma = (\mathbf{z}, \mathbf{h}, c)$

<u>$\mathsf{Verify}(pk, M, \sigma = (\mathbf{z}, \mathbf{h}, c))$</u>
25 $\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ $\quad\triangleright$ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$
26 $\mu \in \{0,1\}^{384} := \mathsf{CRH}(\mathsf{CRH}(\rho \parallel \mathbf{t}_1) \parallel M)$
27 $\mathbf{w}_1' := \mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$ $\triangleright$ Compute as $\mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{z}) - \mathtt{NTT}(c) \cdot \mathtt{NTT}(\mathbf{t}_1 \cdot 2^d))$
28 **return** $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![c = \mathsf{H}(\mu \parallel \mathbf{w}_1')]\!]$ **and** $[\![\#$ of 1's in $\mathbf{h}$ is $\leq \omega]\!]$

Figure 4: The signature scheme Dilithium.

3.3 Number of Iterations

We now want to compute the probability that Step 19 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

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

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 $2\gamma_2$,

Table 1: Parameters for Dilithium. The formulas for the sizes of the public key and signature are given in Section 4.3. The explanations for the NIST security levels is in Section 5.3.

	I weak	II medium	III recommended	IV very high
q	8380417	8380417	8380417	8380417
d	14	14	14	14
weight of c	60	60	60	60
$\gamma_1 = (q-1)/16$	523776	523776	523776	523776
$\gamma_2 = \gamma_1/2$	261888	261888	261888	261888
(k, ℓ)	$(3, 2)$	$(4, 3)$	$(5, 4)$	$(6, 5)$
η	7	6	5	3
β	375	325	275	175
ω	64	80	96	120
pk size (bytes)	896	1184	1472	1760
sig size (bytes)	1487	2044	2701	3366
Exp. reps (from Eq. (5))	4.3	5.9	6.6	4.3
BKZ block-size b to break SIS	235	355	475	605
Best Known Classical bit-cost	68	103	138	176
Best Known Quantum bit-cost	62	94	125	160
BKZ block-size b to break LWE	200	340	485	595
Best Known Classical bit-cost	58	100	141	174
Best Known Quantum bit-cost	53	91	128	158
NIST Security Level	-	1	2	3
Gen cycles (Haswell)	$169,972$	$269,844$	$382,756$	$512,116$
Sign cycles (Haswell)	$765,442$	$1,285,476$	$1,817,902$	$1,677,782$
Verify cycles (Haswell)	$196,048$	$296,920$	$395,936$	$548,558$
Gen cycles (AVX2, Haswell)	$104,128$	$156,432$	$225,432$	$292,404$
Sign cycles (AVX2, Haswell)	$338,922$	$493,332$	$673,144$	$711,018$
Verify cycles (AVX2, Haswell)	$105,584$	$150,228$	$207,164$	$288,398$

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 $1/2$).

As we already mentioned, if $\|c\mathbf{s}_2\|_\infty \le \beta$, then $\|\mathbf{r}_0\|_\infty < \gamma_2 - \beta$ implies that $\mathbf{r}_1 = \mathbf{w}_1$. Thus the last check should succeed with overwhelming probability when the previous check passed. Therefore, the probability that Step 19 passes is

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

It is more difficult to formally compute the probability that Step 22 results in a restart. The parameters were set such that heuristically $(\mathbf{z}, \mathbf{h}) = \perp$ with probability less than 1%. Therefore the vast majority of the loop repetitions will be caused by Step 19.

4 Implementation Details

4.1 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.

We start with the vector $\mathbf{w}_1$ that, together with μ, is hashed to a ball. 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 the coefficients lie in $\{0, \ldots, 15\}$ and can be represented by 4 bits each. This allows $\mathbf{w}_1$ to be packed in a string of $k \cdot 256 \cdot 4/8 = k \cdot 128$ bytes. 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.

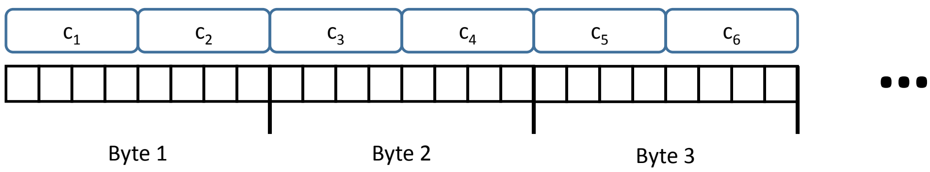

Figure 5: Bit-packing $\mathbf{w}_1$. The k polynomials comprising $\mathbf{w}_1$ are $w_{1,1}, \ldots, w_{1,k}$ and we let $c_1, \ldots, c_{256}$ be the coefficients of $w_{1,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $w_{1,2}$, etc.

Next we turn to the vector $\mathbf{t}_1$, which is the power-of-two rounding of $\mathbf{t}$. Note that $q - 1 = 2^{23} - 2^{13} = (2^9 - 1)2^{14} + 2^{13}$ which shows that the coefficients of the k polynomials of $\mathbf{t}_1$ lie in $\{0, \ldots, 2^9 - 1\}$ and can be represented by 9 bits each. These 9 bits per coefficient, in little-endian byte-order, are bit-packed. In total $\mathbf{t}_1$ needs $k \cdot 256 \cdot 9/8 = 288k$ bytes. See Figure 6 for an explanation of the exact bit packing.

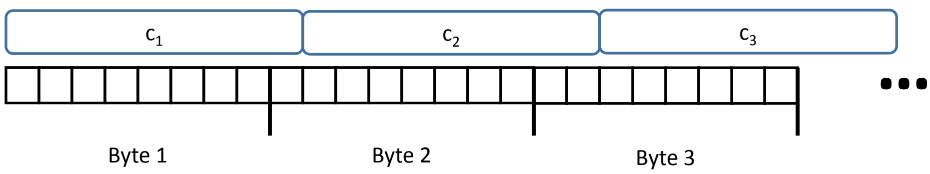

Figure 6: Bit-packing $\mathbf{t}_1$. The k polynomials comprising $\mathbf{t}_1$ are $t_{1,1}, \ldots, t_{1,k}$ and we let $c_1, \ldots, c_{256}$ be the coefficients of $t_{1,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $t_{1,2}$, etc.

The coefficients of the polynomials of $\mathbf{t}_0$ can be written in the form $q + 2^{13} - v$ with $v \in \{0, \ldots, 2^{14} - 1\}$. These v in little endian byte-order are bit-packed. This results in $256 \cdot 14/8$ bytes per polynomial and $k \cdot 256 \cdot 14/8 = 448k$ bytes for $\mathbf{t}_0$. See Figure 7 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 $256 \cdot 4/8 = 128$ bytes for the weak, medium and recommended security levels, and $256 \cdot 3/8 = 96$ bytes for the very high security level. The bit-packing is

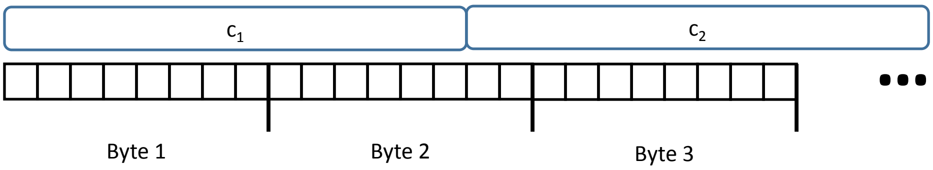

Figure 7: Bit-packing $\mathbf{t}_0$. The k polynomials comprising $\mathbf{t}_0$ are $t_{0,1}, \ldots, t_{0,k}$ and we let $c_1, \ldots, c_{256}$ be the coefficients of $t_{0,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $t_{0,2}$, etc.

done similarly to the case of $\mathbf{w}_1$, $\mathbf{t}_1$ and $\mathbf{t}_0$. See Figure 8 for an explanation of the exact bit packing.

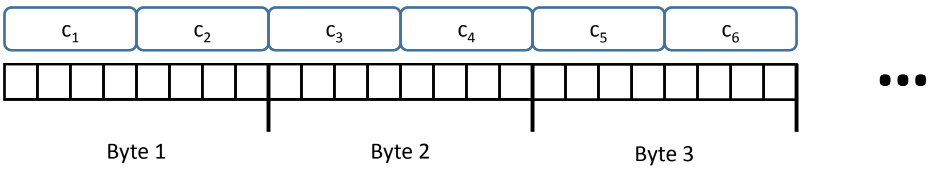

Figure 8: Bit-packing $\mathbf{s}_i$. The ℓ polynomials comprising $\mathbf{s}_1$ are $s_{1,1}, \ldots, s_{1,\ell}$ and we let $\eta - c_1, \ldots, \eta - c_{256}$ be the coefficients of $s_{1,1}$ (with the lower powers first), $\eta - c_{257}, \ldots, \eta - c_{512}$ be the coefficients of $s_{1,2}$, 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 four bits per coefficient (i.e. when $4 \leq \eta \leq 15$). When $\eta < 4$, one would only use three bits per coefficient and pack in the obvious manner.

Finally, $\mathbf{z}$ contains polynomials whose coefficients are equivalent modulo q to $\gamma_1 - 1 - c$ with $c \in \{0, \ldots, 2\gamma_1 - 2\}$ and these values c are bit packed. Since $\lceil \log 2\gamma_1 - 1 \rceil = 20$, 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 9 for an explanation of the exact bit packing.

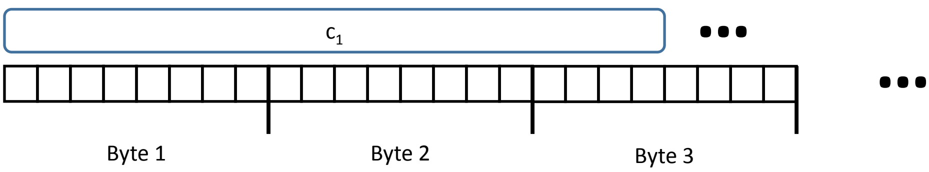

Figure 9: Bit-packing $\mathbf{z}$. The ℓ polynomials comprising $\mathbf{z}$ are $z_1, \ldots, z_\ell$ and we let $\gamma_1 - 1 - c_1, \ldots, \gamma_1 - 1 - c_{256}$ be the coefficients of z_1 (with the lower powers first), $\gamma_1 - 1 - c_{257}, \ldots, \gamma_1 - 1 - c_{512}$ be the coefficients of z_2, etc.

4.2 Hashing

Hashing to a Ball. We now precisely specify the operation of the function $\mathrm{H} : \mu \parallel \mathbf{w}_1 \mapsto c \in B_{60}$ described in Fig. 2 as it is used in our signature scheme. H absorbs the 48 bytes of μ immediately followed by the $128k$ bytes for the bit-packed representation of $\mathbf{w}_1$ 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 60 bits in the first 8 bytes of this random stream are interpreted as 60 random sign bits $s_i \in \{0, 1\}$, $i = 0, \ldots, 59$. The remaining 4 bits are discarded. Then H uses Algorithm 2 to compute c. In each iteration of the for loop it uses rejection sampling on elements from $\{0, \ldots, 255\}$ until it gets 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 coefficient $\hat{a}_{i,j} \in R_q$ of $\hat{\mathbf{A}}$ separately. For the coefficient $\hat{a}_{i,j}$ it absorbs the 32 bytes of ρ immediately followed by one byte representing $0 \leq 2^4 \cdot j + i < 255$ into SHAKE-128. Next it uses consecutive blocks of 3 bytes of the variable-length output string in order to obtain a sequence of integers between 0 and $2^{23} - 1$. This is done by setting the highest bit of the third byte in each block to zero and interpreting the blocks in little endian byte order. So for example the three bytes b_0, b_1 and b_2 from SHAKE-128 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 $a_{i,j}(r_0), a_{i,j}(-r_0), \ldots, a_{i,j}(r_{127})a_{i,j}(-r_{127})$ of $\hat{a}_{i,j}$ uniformly from the set $\{0, \ldots, q-1\}$ in the order of our NTT domain representation.

Sampling the vectors y. The function ExpandMask maps $K \parallel \mu \parallel \kappa$ to $y \in S_{\gamma_1 - 1}^l$, where $\kappa \geq 0$, and works as follows. It computes each of the l coefficients of y, which are polynomials in $S_{\gamma_1 - 1}$, independently. For the i-th polynomial, $0 \leq i < l$, it absorbs the 48 bytes of μ concatenated with the 32 bytes of K and two bytes representing $\kappa + i$ in little endian byte order into SHAKE-256. Then each block of 5 consecutive output bytes is used to get two 20 bit integers between 0 and $2^{20} - 1$. For this the first two bytes of each output block together with a third byte having as lower 4 bits the lower 4 bits of the third output byte and 4 high zero bits is interpreted in little endian order. Then the high 4 bits of the third output byte followed by the 16 bits of the fourth and and fifth byte are interpreted as the second 20 bit integer. As an example assume we have received the five bytes $b_0, \ldots, b_4$ from SHAKE-128. Then ExpandMask computes the two integers $0 \leq b_2' \cdot 2^{16} + b_1 \cdot 2^8 + b_0 \leq 2^{20} - 1$ and $0 \leq b_4 \cdot 2^{12} + b_3 \cdot 2^4 + b_2'' \leq 2^{20} - 1$ where b_2' is the AND of b_2 and 15 and $b_2' = \lfloor b_2/16 \rfloor$. On the resulting sequence of 20 bit integers rejection sampling is performed to get 256 values $v_j \in \{0, \ldots, 2\gamma_1 - 2\}$. From these the polynomial coefficients are computed in increasing order as $q + \gamma_1 - 1 - v_j$.

Collision resistant hash. The function CRH in Figure 4 is a collision resistant hash function. For this purpose 384 bits of the output of SHAKE-256 are used. CRH is called with two different sets of inputs. First it is called with $\rho \parallel \mathbf{t}_1$. The function then absorbs the 32 bytes of ρ followed by the $k \cdot 256 \cdot 9/8$ bytes for the bit-packed representation of $\mathbf{t}_1$ into SHAKE-256 and takes the first 48 bytes of the first output block of SHAKE-256 as the output hash. The second input is $\mu \parallel M$. Here the concatenation of the hash μ and the message string are absorbed into SHAKE-256 and the first 48 output bytes are used as the resulting hash.

4.3 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 $32 + 288k$ 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 $64 + 48 + 32((k + l) \cdot \lceil \log 2\eta + 1 \rceil + 14k)$ bytes. For the weak, medium and high security level this is equal to $112 + 576k + 128l$ bytes. With the very high security parameters one needs $112 + 544k + 96l = 3856$ bytes.

Signature. The signature byte string is the concatenation of a bit packed representation of $\mathbf{z}$ and encodings of h and c in this order. We describe the encoding of h, which needs $\omega + k$ bytes. Together all the polynomials in the vector $\mathbf{h}$ have at most ω non-zero coefficients. It is 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 $0 \leq j \leq \omega$. In the encoding of the challenge c, the first 256 bits are 0 or 1 when the corresponding coefficient of c is zero or non-zero, respectively. The next 60 bits are 0 or 1 if the corresponding non-zero coefficient is 1 or -1, respectively. Note that there are precisely 60 non-zero coefficients. The 4 bits up to the next byte boundary are zero.

Therefore, a signature requires $640l + \omega + k + 40$ bytes.

4.4 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 $\mathsf{CRH}(\mu \parallel \mathbf{w}_1)$ also in the case of rejected $\mathbf{y}$, but this does not reveal any information about the secret key when CRH is modeled as a random oracle and $\mathbf{w}_1$ has high min-entropy.

4.5 Reference implementation

Our reference NTT is a natural iterative implementation for 32 bit unsigned 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 $r' \equiv r2^{32} \pmod q$. 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 and ExpandMask initially squeeze a number of output blocks of SHAKE-256 and SHAKE-128 that gives 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 needed at least for one polynomial. They suffice with probability greater than $1-2^{-132}$. ExpandMask initially retrieves 5 blocks from SHAKE-256 that have 136 bytes. This is the minimum number of blocks and suffices with probability greater than $1-2^{-81}$.

As mentioned in the introduction our reference implementation is protected against timing attacks. For this reason the centralized remainders in the rounding functions given in Figure 3 are not computed with branchings. Instead we use the following well-known trick to compute the centralized remainder $r' = r \bmod^{\pm} \alpha$ where $0 \leq r < q$. Subtracting $\alpha/2+1$ from r yields a negative result if and only if $r \leq \alpha/2$. Therefore, shifting this result arithmetically to the right by 31 bits gives -1, i.e. the integer with all bits equal to 1, if $r \leq \alpha/2$ and 0 otherwise. Then the logical AND of the shifted value and α is added to r and $\alpha/2-1$ subtracted. This results in $r-\alpha$ if $r > \alpha$ and r if $r \leq \alpha/2$, i.e. the centralized remainder.

We make heavy use of lazy reduction in our implementation. In the NTT we do not reduce the results of additions and subtractions at all. For rounding and norm checking it is important to map to standard representatives. This freezing of the coefficients is achieved in constant-time by conditionally subtracting q with another instance of the arithmetic right shift trick.

4.6 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 unsigned 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. 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` in the second last level and a combination of `vpshufd` and `vpblendd` in the last level. The multiplications with the constant roots of unity are performed using the `vpmuludq` 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.

4.7 Computational Efficiency

We have performed timing experiments with our reference implementation on a Haswell 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 10000 operations each. Signing was performed with a message size of 32 bytes. The computer we have used is equipped with an Intel Core i7-4770K CPU running at the constant clock frequency of 3500 Mhz. Hyperthreading and Turbo Boost are switched off. The system runs Debian stable with Linux Kernel version 3.16.0 and the code was compiled with gcc 6.3.0.

5 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 because 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 hardness 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

[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.

hardness of MSIS and the hash function H. In the classical ROM, there is a (non-tight) reduction from MSIS to SelfTargetMSIS.

5.1 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

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

The SelfTargetMSIS Problem. Suppose that $\mathsf{H} : \{0,1\}^* \to B_{60}$ is a cryptographic hash function. To an algorithm A we associate the advantage function

$$\mathrm{Adv}^{\mathsf{SelfTargetMSIS}}_{\mathsf{H},m,k,\gamma}(\mathsf{A}) :=$$

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

5.2 Signature Scheme Security

The concrete security of Dilithium was analyzed in [KLS17], 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

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

for D a uniform distribution over S_η, and

$$\zeta = \max\{\gamma_1 - \beta, 2\gamma_2 + 1 + 2^{d-1}\cdot 60\} \leq 4\gamma_2, \tag{7}$$

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

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.

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

5.2.1 UF-CMA Security Sketch

It was shown in [KLS17] 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 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 [KLS17][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

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

Lemma 1 implies that one can rewrite

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

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

$$\|\mathbf{u}'\|_\infty \leq \|c\mathbf{t}_0\|_\infty + \|\mathbf{u}\|_\infty \leq \|c\|_1 \cdot \|\mathbf{t}_0\|_\infty + \|\mathbf{u}\|_\infty \leq 60 \cdot 2^{d-1} + 2\gamma_2 + 1 < 4\gamma_2.$$

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 $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$, $\|c\|_\infty = 1$, $\|\mathbf{u}'\|_\infty < 4\gamma_2$, and $M \in \{0,1\}^*$ such that

$$\mathsf{H}\left([\ \mathbf{A} \mid \mathbf{t} \mid \mathbf{I}_k\ ] \cdot \begin{bmatrix} \mathbf{z} \\ c \\ \mathbf{u}' \end{bmatrix} \ \|\ M \right) = c. \tag{11}$$

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

[6] It was also shown in [KLS17] 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.

[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.

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. (11).

The intuition for the security of the problem in Eq. (11) (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 M "strategically" should not help – so the problem would be equally hard if the M 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 $\mathsf{H}(\ \mathbf{w} \parallel M\ ) = c$, and then finding $\mathbf{z}, \mathbf{u}'$ such that $\mathbf{Az} + \mathbf{u}' = \mathbf{w} + c\mathbf{t}$.[8] The hardness of finding such $\mathbf{z}, \mathbf{u}'$ with ℓ_∞-norms less than $4\gamma_2$ such that

$$\mathbf{Az} + \mathbf{u}' = \mathbf{t}' \tag{12}$$

for some $\mathbf{t}'$ is the problem whose concrete security we will be analyzing. Note that this is conservative because in Eq. (11) $\|\mathbf{z}\|_\infty < \gamma_1 - \beta \approx 2\gamma_2$. Furthermore, only ω coefficients of $\mathbf{u}'$ can be larger than $2\gamma_2$.

5.2.2 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 M 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 $\mathbf{Az} + \mathbf{u} = \mathbf{0}$.

5.3 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 follow the general methodology from [ADPS16, BCD+16] to analyze the security of our signature scheme, with minor adaptations. This methodology is significantly more conservative than prior ones used in lattice-based cryptography. In particular, we assume the adversary can run the asymptotically best algorithms known, with no overhead compared to the asymptotic run-times. In particular, we assume the adversary can cheaply handle huge amounts of (possibly quantum) memory. This conservatism is in line with the goal of long-term post-quantum security. We note that despite this security analysis methodology, our schemes remain competitive in practice.

[8]This is indeed the (non-tight) proof sketch in the classical random oracle model.

The security parameters in Table 1 are based on this conservative methodology. Since we are making so many approximations (in favor of the adversary), it may seem a little strange that our numbers are so "precisely" stated. The purpose of such precision is to make it possible to compare between our scheme and other lattice-based ones based on the same conservative analysis. On the other hand, because our security levels understates the actual security of the schemes and the best cryptanalytic algorithms are extremely memory-intensive, we believe that our schemes still satisfy their stated "NIST Security Level" designation despite the security numbers appearing to be below the required levels.

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 5.1 with the ring R_q being replaced by $\mathbb{Z}_q$.

5.4 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 1. 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.[9] 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.

References

[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`. 16, 17, 21, 28

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part I*, volume 6755 of *LNCS*, pages 403–415. Springer, Heidelberg, July 2011. 29

[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. 18

[BCD+16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 1006–1018. ACM Press, October 2016. 21

[9] 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.

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

[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. 2, 20, 26

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Marc Fischlin and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015. http://cryptojedi.org/papers/#sphincs. 6

[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. 2

[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. 29

[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 Robert Krauthgamer, editor, *27th SODA*, pages 893–902. ACM-SIAM, January 2016. 30

[CDPR16] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 559–585. Springer, Heidelberg, May 2016. 30

[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. 30

[CGS14] Peter Campbell, Michael Groves, and Dan Shepherd. Soliloquy: A cautionary tale. In *ETSI 2nd Quantum-Safe Crypto Workshop*, pages 1–9, 2014. 30

[CLP+17] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang, and Chen-Mou Cheng. Implementing 128-bit secure mpkc signatures. Cryptology ePrint Archive, Report 2017/636, 2017. https://eprint.iacr.org/2017/636. 6

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20. Springer, Heidelberg, December 2011. 27

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *CRYPTO (1)*, pages 40–56, 2013. 2, 5

[DLP14] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. In *ASIACRYPT*, pages 22–41, 2014. 2, 5

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoit Gerard, and Mehdi Tibouchi. Side-channel attacks on bliss lattice-based signatures – exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. Cryptology ePrint Archive, Report 2017/505, 2017. https://eprint.iacr.org/2017/505 To appear in CCS 2017. 2

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *CHES*, pages 530–547, 2012. 2

[GQ88] Louis C. Guillou and Jean-Jacques Quisquater. A "paradoxical" indentity-based signature scheme resulting from zero-knowledge. In *CRYPTO*, pages 216–231, 1988. 18, 21

[HPS11] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 447–464. Springer, Heidelberg, August 2011. 27

[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. 30

[KLS17] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. Cryptology ePrint Archive, Report 2017/916, 2017. `https://eprint.iacr.org/2017/916`. 4, 5, 10, 19, 20, 26

[Knu97] Donald Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3 edition, 1997. page 145. 7

[Laa15] Thijs Laarhoven. *Search problems in cryptography.* PhD thesis, Eindhoven University of Technology, 2015. 27

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In *ASIACRYPT*, pages 598–616, 2009. 2, 20

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In *EUROCRYPT*, pages 738–755, 2012. 20, 26

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To bliss-b or not to be - attacking strongswan's implementation of post-quantum signatures. Cryptology ePrint Archive, Report 2017/490, 2017. `https://eprint.iacr.org/2017/490`. To appear in CCS 2017. 2

[PSS+17] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. Cryptology ePrint Archive, Report 2017/1014, 2017. `https://eprint.iacr.org/2017/1014`. 10

[SBB+17] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking ed25519 in wolfssl. Cryptology ePrint Archive, Report 2017/985, 2017. `https://eprint.iacr.org/2017/985`. 10

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO*, pages 239–252, 1989. 18, 21

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994. 26

A Proofs for Rounding Algorithm Properties

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

Lemma 3. *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 4. *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 5. *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.

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.

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 17 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 $\|c\mathbf{s}_i\|_\infty \leq \beta$ (with overwhelming probability), 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 $S^{\ell}_{\gamma_1-\beta-1} \times B_{60}$.

The simulation of the signature follows [Lyu12, BG14]. The simulator picks a uniformly random $(\mathbf{z}, c)$ in $S^{\ell}_{\gamma_1-\beta-1} \times B_{60}$, 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 $\|c\mathbf{s}_2\|_\infty \leq \beta$, we will have

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

Since our β was chosen such that the probability (over the choice of $c, \mathbf{s}_2$) that $\|c\mathbf{s}_2\|_\infty < \beta$ is $> 1 - 2^{-128}$, the simulator does not need to perform the check that $\mathbf{r}_1 = \mathbf{w}_1$ and can always assume that it passes.

We can then program

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

Unless we have already set the value of $\mathsf{H}(\mu \parallel \mathbf{w}_1)$ to something else, the resulting pair $(\mathbf{z}, c)$ has the same distribution as in a genuine signature of μ. It was shown in [KLS17] that the probability, over the random choice of $\mathbf{A}$ and $\mathbf{y}$, that we already set the value of $\mathsf{H}(\mu \parallel \mathbf{w}_1)$ is less than 2^{-255}.

All the other steps (after Step 19) 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 [CN11]. 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 [BDGL16] 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.

In our case, the input ℓ_i's are of the following form (cf. Fig. 10). 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 $\mathit{slope}(b)$ 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 $\mathit{slope}(b)$ (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

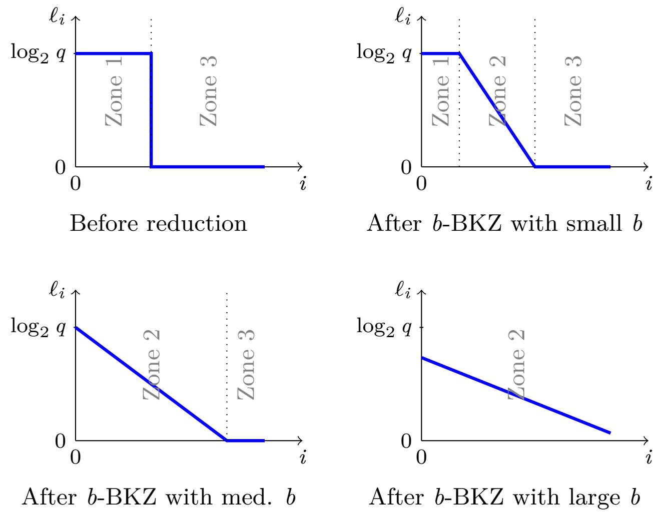

Figure 10: Evolution of Gram-Schmidt length in log-scale under BKZ reduction for various blocksizes. The area under the curves remains constant, and the slope in Zone 2 decrease with the blocksize. Note that Zone 3 may disappear before Zone 1, depending on the shape of the input basis.

lattice $\Lambda = \{\mathbf{x} \in \mathbb{Z}^d : \mathbf{M}\mathbf{x} = \mathbf{0} \bmod \mathbf{q}\}$ 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 5.2.1, the best known attack against the SelfTargetMSIS problem involves either breaking the security of H or solving the problem in Eq. (12). The latter amounts to solving the $\mathsf{MSIS}_{k,\ell+1,\zeta}$ problem for the matrix $[\ \mathbf{A} \mid \mathbf{t}'\ ]$. [10]

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

[10] Note that a solution to Eq. (12) would require the coefficient in from of $\mathbf{t}'$ to be ± 1, while we're allowing any small polynomial. Furthermore, as discussed after Eq. (12), 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.

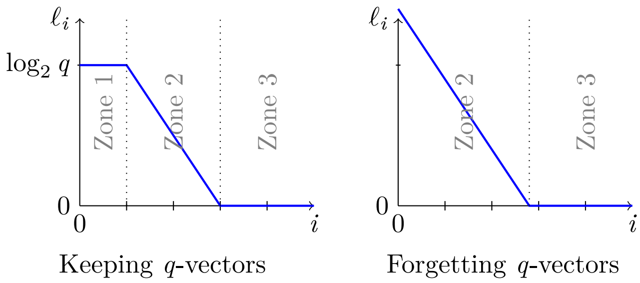

Figure 11: Effect of forgetting q-vectors by randomization, under the same BKZ-blocksize b.

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, 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 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. 11.

It turns out that this strategy always improves over the previous strategy for the parameter ranges considered in Table 1. 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.