DRS : Diagonal dominant Reduction for lattice-based Signature

Thomas PLANTARD,
Arnaud SIPASSEUTH, Cédric DUMONDELLE, Willy SUSILO

Institute of Cybersecurity and Cryptology
School of Computing and Information Technology
University of Wollongong
Australia

1 Background

Definition 1. *We call lattice a discrete subgroup of $\mathbb{R}^n$. We say a lattice is an integer lattice when it is a subgroup of $\mathbb{Z}^n$. A basis of the lattice is a basis as a $\mathbb{Z}-$module.*

In our work we only consider full-rank integer lattices (unless specified otherwise), i.e such that their basis can be represented by a $n \times n$ non-singular integer matrix. It is important to note that just like in classical linear algebra, a lattice has an infinity of basis. In fact, if B is a basis of $\mathcal{L}$, then so is UB for any unimodular matrix U (U can be seen as the set of linear operations over $\mathbb{Z}^n$ on the rows of B that do not affect the determinant).

Definition 2 (Minima)**.** *We note $\lambda_i(\mathcal{L})$ the $i-$th minimum of a lattice $\mathcal{L}$. It is the radius of the smallest zero-centered ball containing at least i linearly independant elements of $\mathcal{L}$.*

Definition 3 (Lattice gap)**.** *We note $\delta_i(\mathcal{L})$ the ratio $\frac{\lambda_{i+1}(\mathcal{L})}{\lambda_i(\mathcal{L})}$ and call that a lattice gap. When mentioned without index and called "the" gap, the index is implied to be $i = 1$.*

Definition 4. *We say a lattice is a diagonally dominant type lattice (of dimension n) if it admits a basis of the form $B = D + R$ where $D = d \times Id, d \in \mathbb{Z}$, and R is a "noise" matrix and $D + R$ is a diagonal dominant matrix as in [1], i.e*

$$\forall i \in [1, n], B[i][i] \geq \sum_{j=1, i \neq j}^{n} |B[i][j]|$$

In our scheme, we use a diagonal dominant lattice as our secret key, and will refer to it as our "reduction matrix" (as we use this basis to "reduce" our vectors).

Definition 5. *Let F be a subfield of $\mathbb{C}$, V a vector space over F^k, and p a positive integer or ∞. We call l_p norm over V the norm:*

- $\forall x \in V, \|x\|_p = \sqrt[p]{\sum_{i=1}^{k} |x_i|^p}$
- $\forall x \in V, \|x\|_\infty = \max_{i \in [1,k]} |x_i|$

The norm we use in our scheme is the maximum norm. We note that we also define the maximum matrix norm as the biggest value among the sums of the absolute values in a single column.

Definition 6 (**uSVP**$_\delta$: δ-unique Shortest Vector Problem)**.** *Given a basis of a lattice $\mathcal{L}$ with its lattice gap $\delta > 1$, solve SVP.*

Definition 7 (**BDD**$_\gamma$: γ-Bounded Distance Decoding)**.** *Given a basis B of a lattice $\mathcal{L}$, a point x and a approximation factor γ ensuring $d(x, \mathcal{L}) < \gamma\lambda_1(B)$ find the lattice vector $v \in \mathcal{L}$ closest to x.*

It has been proved that $\mathbf{BDD}_{1/(2\gamma)}$ reduces itself to $\mathbf{uSVP}_\gamma$ in polynomial time and the same goes from $\mathbf{uSVP}_\gamma$ to $\mathbf{BDD}_{1/\gamma}$ when γ is polynomially bounded by n [7], in cryptography the gap is polynomial the target point x must be polynomially bounded therefore solving one or the other is relatively the same in our case. To solve those problems, we usually use an embedding technique that extends a basis matrix by one column and one row vector that are full of zeroes except for one position where the value is set to 1 at the intersection of those newly added spaces, and then apply lattice reduction techniques on these. As far as our signature scheme is concerned, the $\mathbf{GDD}_\gamma$ is more relevant:

Definition 8 ($\mathbf{GDD}_\gamma$: γ-Guaranteed Distance Decoding)**.** *Given a basis B of a lattice $\mathcal{L}$, any point x and a approximation factor γ, find $v \in \mathcal{L}$ such that $\|x - v\| \leq \gamma$.*

2 Our scheme

The raw step-by-step idea for *Alice* to sign a file from *Bob* is the following:

- *Alice* sends a basis P (the public key) of a diagonal dominant lattice $\mathcal{L}(P)$ to *Bob*.
  This basis should have big coefficients and obfuscate the diagonal dominant structure.
- *Bob* sends a vector message m (that has big coefficients) to *Alice*.
  He challenges *Alice* to find $\|s\| < \gamma$ such that $m - s \in \mathcal{L}(P)$.
- *Alice* uses a diagonal dominant basis $D - M$ of $\mathcal{L}(P)$ to solve the $\mathbf{GDD}_\gamma$ on $\mathcal{L}(P)$ and m.
  She obtains a vector signature s (that has small coefficients) and give it to *Bob*.
- *Bob* checks if $(m - s) \in \mathcal{L}(P)$ and $\|s\| < \gamma$ for our $\mathbf{GDD}_\gamma$ problem.
  The signature is correct if and only if this is verified.

Our scheme is inspired by the scheme proposed by Plantard et al. [9], which was originally inspired by GGH itself [4].

In this section, we just give a quick overview of our algorithms and explain a bit the ideas behind them. How we choose to implement it in practice will be more detailed in the implementation section.

2.1 Setup: lattice, reduction matrix generation, and a random seed

The algorithm here is straightforward, we just have to generate a diagonal dominant matrix S of size $n * n$ with a low and bounded noise. To achieve this, we will just generate one first vector $S[1]$ in three simple steps.

1. We pick the diagonal coefficient d.
2. We pick four non-zero values b, N_b, N_1, Δ such that $d = b * N_b + N_1 + \Delta$.
3. $S[1][1] = d$ is our vector's first coefficient.
   Elsewhere, we put randomly N_b values b, N_1 values 1, and the rest is filled with zeroes.

And now, from $S[1]$ we generate our matrix S in two steps:

1. We generate each vector $S[i]$ by using a circular shift by one column $S[i+1]$:
   $S[i][1] \to S[i+1][2]$, $S[i][2] \to S[i+1][3]$, ... , $S[i][n] \to S[i+1][1]$.
2. We then multiply every non-zero and non-d coefficient by 1 or -1 (just changing the sign randomly).

Then, we have finished the required steps, and just created a diagonal dominant matrix $S = D - M$ where $D = d * Id$, $\|M\| = b * N_b + N_1 = d - \Delta$ and $\|M[1]\|_2 = \sqrt{N_B * B^2 + N_1}$. More details will be given later and how we decide to choose those "four non-zero values" (b, N_b, N_1, Δ), but those choices can be changed at the user's discretion as long as the previous equalities hold.

As part of the secret key, we also keep a seed value s that will be used as a seed for random generators, as we will explain in the next sections.

2.2 Setup: public key generation

Like most public-key lattice based cryptosystems, we construct our public key P such that a matrix U such that $P = US$ where U is unimodular and S is our secret matrix.

We want the coefficients of P to be bigger but relatively balanced, while having a fast method to generate it. Let us describe the following matrices:

$$A_+ = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \text{ and } A_- = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} \text{ and}$$

$$T^{n/2}_{\{+,-\}} = \left\{ i \in [1, n/2], x_i \in \{+,-\} : \begin{bmatrix} A_{x_1} & 0 & \dots & & 0 \\ 0 & A_{x_2} & \ddots & & \\ \vdots & 0 & \ddots & \ddots & \vdots \\ & & \ddots & A_{x_{(n/2)-1}} & 0 \\ 0 & \dots & & 0 & A_{x_{n/2}} \end{bmatrix} \right\}$$

where every $T \in T^{n/2}_{\{+,-\}}$ is an unimodular matrix composed of A_+ and A_- in its diagonal and 0 elsewhere. If δ is the maximum size of the coefficients in a matrix M, then after being multiplied by a transformation matrix $T \in T^{n/2}_{\{+,-\}}$, the maximum size of the coefficients in TM is at most 3δ, and we will use a matrix in $T^{n/2}_{\{+,-\}}$ everytime we wish to grow our coefficients. Using $T^{n/2}_{\{+,-\}}$, we define

$$U_i = T_i P_i \text{ where } P_i \in S_n \text{ is a permutation matrix and } T_i \in T^{n/2}_{\{+,-\}} \text{ randomly chosen.}$$

The point of using permutation matrices P_i is to ensure we use a different combination of rows at every growing step. Finally, we will use it to create

$$U = P_{R+1} \prod_{i=1}^{R} U_i$$

such that $R \geq 1$ is a certain number we choose ourselves and $P = U(D - M)$.

Note that, for every value of R, we obtain different U, and furthermore, since we are taking both T_i and P_i randomly, we note that every choice is dependent of the seed we put for our random generators (assuming they are deterministic like the ones provided by the NIST). Therefore we generate P in the following way:

1. Use our random secret seed s to set our random generators, and $P \leftarrow S$ where S is our secret matrix
2. Choose "randomly" $T_{s,1}$, a transformation matrix and $P_{s,1}$ a permutation matrix
3. Set $P \leftarrow T_{s,1} P_{s,1} P$
4. Repeat the last two steps that $R - 1$ more times and reapply one final permutation $P \leftarrow P_{s,R+1} P$

Finally, we obtain our final public-key matrix P.

2.3 Verification

To verify our signature, we need some additional information to decide whether s is a valid signature or not $((m - s) \in \mathcal{L}(P))$. We know that $(m - s) \in \mathcal{L}(P)$ if and only if there exists $k \in \mathbb{Z}^n$ such that $(m - s) = kP$.

Therefore, given k, m, s, P, just check whether or not we have $(m - s) = kP$. If the verification holds the signature is valid. Otherwise, it is invalid.

2.4 Signature

To generate the signature, we use exactly the same algorithm as the one used by Plantard et al. which in our case is basically reducing every big coefficient $|m[i]| > d$ of a vector message m by a value of $q * d$ such that $|m[i] - q * d| < d$, but re-adding some little noise $|qb|$ and $|q|$ in some other coefficients $m[j]$ with $j \neq i$, i.e

$$\text{applying } m \leftarrow m - qS[i] \text{ where } |m[i] - q * d| < d \text{ for every } |m[i]| > d \text{ until } \|m\|_\infty < d.$$

All proofwork can be found in the paper of the original reduction idea [9], however we propose here an alternative proved bound for the reduction to work, which is much easier to understand and to practically use: using the norm l_1, and supposing that our reduction matrix $d * Id - M$ is diagonal dominant.

Suppose we reduce a vector m by a q times a vector of our diagonal dominant basis S, which only happens when

$$\exists i \in [1, n],\ |m[i]| > d$$

Which means that we dropped $\|m\|_1$ by exactly $|qd|$ on one coefficient, but added $|q| \sum_{j=1}^{n} |M[i][j]|$ at most on $\|m\|_1$. Since the diagonal dominance gives $d > \sum_{j=1, j\neq i}^{n} |M[i][j]|$

$$q \text{ has the same sign as } m[i] \text{ and } d > 0 \text{ so } |m[i] - qd| = |m[i]| - |qd| < d < |m[i]|$$

$$\text{and } d < |m[i]|, \text{ thus } |m[i]| - |qd| < |m[i]|$$

$$\|m\|_1 - |qd| + |q| \textstyle\sum_{j=1, j\neq i}^{n} |M[i][j]|) < \|m\|_1$$

thus $\|m\|_1$ is lower than before : thus, we effectively reduce $\|m\|_1$ until $\|m\|_\infty < d$.

Once we have s, we still need the final k such that $kP = (m - s)$. To generate the final membership vector k, we basically first construct its values such that $k'(D - M) = k'S = m - s$, as we reduce m. At the first step, $s = m$ and $k = [0, ..., 0]$. However, everytime we use the vector $s = m - q * S[i]$, then $k[i] \leftarrow k[i] + q$ to keep the equality $k'S = m - s$ true.
Once the final s is constructed, we know $P = U(D - M)$ and $k \leftarrow k'U^{-1}$ and thus verify

$$kP = k'U^{-1}U(D - M) = k'(D - M) = (m - s)$$

As far as the computation of U^{-1}, it is fairly simple. Since

$$U = P_{R+1} \textstyle\prod_{i=1}^{R} T_i P_i = P_{R+1} T_R P_R ... T_1 P_1$$

we have

$$U^{-1} = (\textstyle\prod_{i=1}^{R} P_i^{-1} T_i^{-1}) P_{R+1}^{-1} = P_1^T T_1^{-1} ... P_R^T T_R^{-1} P_{R+1}^T$$

(note that order matters), and knowing

$$A_+^{-1} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \text{ and } A_-^{-1} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \text{ and}$$

$$T_{\{+,-\}}^{-(n/2)} = \left\{ i \in [1, n/2], x_i \in \{+, -\} : \begin{bmatrix} A_{x_1}^{-1} & 0 & \dots & & 0 \\ 0 & A_{x_2}^{-1} & \ddots & & \\ \vdots & 0 & \ddots & \ddots & \vdots \\ & & \ddots & A_{x_{(n/2)-1}}^{-1} & 0 \\ 0 & \dots & & 0 & A_{x_{n/2}}^{-1} \end{bmatrix} \right\}$$

U^{-1} is thus as easy to compute as U, and also give the exact same maximum coefficient growth 3^R, and we proceed very similarly to the generation of the public key, as follows:

1. Use our random secret seed s to set our random generators, and $k \leftarrow k'$ where $k'(D-M) = m - s$.
2. Choose "randomly" $T_{s,1}^{-1}$, the inverse of a transformation matrix and $P_{s,1}^{-1}$ the inverse of a permutation matrix
3. Set $k \leftarrow kP_{s,1}^{-1}T_{s,1}^{-1}$
4. Repeat the last two steps $R-1$ more times and reapply one final inverse permutation $k \leftarrow kP_{s,R+1}^{-1}$

Finally, we can give Bob the final couple (k, s) as the signature.

3 Security

The initial idea of reducing vectors using diagonal dominant lattices and the maximum norm was done as a countermeasure against the parallelepiped attack from [8] in Plantard et al's suggestion at PKC2008 [9] to fix GGHSign [4]. Their fix using the maximum norm have been unchallenged for almost a decade now and no critical security failure in their scheme have been discovered to the best of our knowledge. In the following subsection we will describe the state of the art method to attack it.

3.1 BDD-based attack

The security is based on what is known as the currently most efficient way to attack the scheme, a **BDD**-based attack as described below.

Input: Pk the public key of full rank n, d the diagonal coefficient, ϕ a $\mathbf{BDD}_\gamma$ solver
Output: $Sk = (D - M)$ the secret key
$Sk \leftarrow d * Id_n$;
// Loop on every position of the diagonal
foreach $\{i \in [1..n]\}$ **do**
  // Find r the difference between $(0, ...0, d, 0, ..., 0)$ and $\mathcal{L}(Pk)$
  $r \leftarrow \phi(\mathcal{L}(Pk), Sk[i])$;
  $Sk[i] \leftarrow Sk[i] + r$;
end
return Sk;

Algorithm 1: Diagonal Dominant Key recovery attack

Currently, the most efficient way to perform this attack will be:

i) to transform a **BDD** problem into a Unique Shortest Vector Problem (**uSVP**) (Kannan's Embedding Technique [6]), assuming $v = (0, ...0, d, 0, ..., 0)$

$$\begin{pmatrix} v & 1 \\ B & 0 \end{pmatrix},$$

ii) to solve this new **uSVP** using lattice reduction algorithm.

Using this method, we obtain a **uSVP** with a gap

$$\gamma \approx \frac{\Gamma\left(\frac{n+3}{2}\right)^{\frac{1}{n+1}} Det(\mathcal{L})^{\frac{1}{n+1}}}{\sqrt{\pi}\|M[1]\|_2} \approx \frac{\Gamma\left(\frac{n+3}{2}\right)^{\frac{1}{n+1}} d^{n\frac{1}{n+1}}}{\sqrt{\pi}\|M[1]\|_2}. \quad (1)$$

Lattice reduction methods are well studied and their strength are evaluated using the Hermite factor. Let $\mathcal{L}$ a $d-$dimensional lattice, the Hermite factor of a basis B of $\mathcal{L}$ is given by

$$\frac{\|B[1]\|_2}{det(\mathcal{L})^{\frac{1}{n}}}.$$

Consequently, lattice reduction algorithms strengths are given by the Hermite factor of their expected output basis.

In [3], it was estimated that lattice reduction methods solve USVP$_\gamma$ with γ a fraction of the Hermite factor. We will use a conservative bound of $\frac{1}{4}$ for the ratio of the USVP gap to the Hermite factor.

To guarantee security against the possibility of an attacker to eliminate the b coefficients from any vector of M by enumerating all the possible combinations, we pick N_b being at least the lowest value such that $2^{N_b}\binom{n}{N_b} \geq 2^\lambda$.

3.2 Expected Security Strength

Different papers are giving some relations between the Hermite factor and the security parameter λ [5, 10] often using BKZ simulation [2]. Aiming to be conservative, we are to assume a security of $2^{128}, 2^{192}, 2^{256}$ for a Hermite factor of $1.006^d, 1.005^d, 1.004^d$ respectively. we set $D = n$, and pick $\delta = 28$, $R = 24$ and $\Delta = 32$.

Dimension	n	N_B	B	N_1	Δ	R	δ	γ	2^λ
912	912	16	28	432	32	24	28	$< \frac{1}{4}(1.006)^{d+1}$	2^{128}
1160	1160	23	25	553	32	24	28	$< \frac{1}{4}(1.005)^{d+1}$	2^{192}
1518	1518	33	23	727	32	24	28	$< \frac{1}{4}(1.004)^{d+1}$	2^{256}

Table 1: Parameter Sets.

Table 1 parameters have been choosen to obtain a USVP gap (Equation 1) with $\gamma < \frac{\delta^{d+1}}{4}$ for $\delta = 1.006, 1.005, 1.004$.

4 Our implementation

While we gave the overall idea in the previous sections, in this section we specify some implementation choices. Nevertheless those choices are not intrinsic to the scheme and can be changed.
Below is an overview of the main point of our implementation:

4.1 Program parameters, and algorithm changes

The whole scheme is set by 7 parameters:

- n: the dimension
- s: a seed for random generators
- D: the diagonal coefficient, also the bound for the max norm of our reduced vectors
- δ: the bound for the max norm of our hashed messages vectors
- λ: the standard security parameter
- Δ: a parameter that defines an extra sparsity in our reduction matrix
- R: a "round" number, indicating the number of loop iterations used to generate the public key.

Note the introduction of a seed parameter s, that serves in both the public key generation and signature algorithm, and which interacts (directly or indirectly) with the following functions:

- **RdmSeed** : $s \rightarrow ()$ determines the output of the two next functions
- **RdmPmtn** : $M \rightarrow \sigma(M)$ randomly permutes the rows of the input matrix, or the values of an input vector
- **RdmSgn** : $() \rightarrow \{-1, 1\}$ output a random value, -1 or 1

Another important point is that rather than signing a vector message that is given to us, we sign a vector produced by the hashing of the received message. For now, we will refer to the hashed message vector as the message.

To maximize efficiency, we choose those last parameters such that all intermediate computations fit in 64-bits integers. One intermediate computation that might overflow is while checking the validity of messages-signatures couples. This is determined by the four parameters δ, Δ, D, R. Here, we choose to fix $D = n$, $\delta = 28$,$\Delta = 32$ and $R = 24$.

We will give all input sizes in the rest of the report in bits.

4.2 Algorithms

4.2.1 Secret Key Setup

From those initial parameters specified above, we compute other values for the secret reduction basis which coefficients' information is based on one initial secret vector:

- N_B the number of occurences per vector of the "big" noise $\{-B, B\}$, and is the lowest positive number such that $2^{N_B}\binom{n}{N_b} \geq 2^{\lambda}$
- B the value of the "big" noise, and is equal to $D/(2N_B)$
- N_1 the number of occurences per vector of the small noise $\{-1, 1\}$, and is equal to $D - (N_B B) - \Delta$

Input: - all initial parameters;
- another extra random seed x_2;
Output: - x, S the secret key;
// Initialization
$S \leftarrow 0$;
$t \leftarrow v \in \mathbb{Z}^n$;
// Algorithm start
$RdmSeed(x_2)$;
$t[1] \leftarrow D$;
// Put B values
for $i = 2$; $i < N_B$; $i = i + 1$ **do**
  $t[i] \leftarrow B$;
end
// Put ones
for $i = N_B + 1$; $i < N_B + N_1 + 1$; $i = i + 1$ **do**
  $t[i] \leftarrow 1$;
end
// Complete with zeroes
for $i = N_B + N_1 + 1$; $i < n$; $i = i + 1$ **do**
  $t[i] \leftarrow 0$;
end
$t \leftarrow$ **RdmPmtn**(t);
for $i = 1$; $i < n$; $i = i + 1$ **do**
  $S[i][i] \leftarrow t[0]$;
  for $j = 1$; $j < n$; $j = j + 1$ **do**
    $S[i][((i + j) \mod n) + 1] \leftarrow t[i][j] * RdnSgn()$;
  end
end
// Algorithm ends
return x, S;

Algorithm 2: Secret key generation

Please note that the matrix composed by the absolute values of the secret key is cyclic. This property allows us to keep only the first vector and the matrix representing the sign of the coefficients. Furthermore, to encore the first vector, we use the bits 0 for 0, 11 for a B, 10 for a 1 for the first vector. We ignore the first coefficient which is always d: this give us the bit size of the first vector as $(n-1) + N_b + N_1$. Then a sign for every non-zero element $(n(N_1 + N_b))$, and N_x bits the number of bits for the seed s used when generating P. Therefore the size of the secret key in bits is $(n-1) + (N_1 + N_b) + n(N_1 + N_b) + N_x = (n-1) + (n+1)(N_1 + N_b) + N_x$.

4.2.2 Public Key Setup

The public key setup is as described initially. We add an extra value corresponding $2^{63-\lceil \log_2(\|P\|_\infty) \rceil}$ this will help us to ensure that there will be no overflow during the verification process.

Input: - $R \in \mathbb{N}$ a number of rounds;
- $S = (D - M)$ the reduction matrix of dimension n;
- a random seed x;
Output: - P the public key, and p_2 a power of two;
// Initialization
$P \leftarrow S$;
// Algorithm start
$RdmSeed(x)$;
// Apply R rounds
for $i = 1$; $i < R$; $i = i + 1$ **do**
  $P \leftarrow$ **RdmPmtn**(P);
  for $j = 1$; $j < n - 1$; $j = j + 2$ **do**
    $t \leftarrow$ **RdmSgn()**;
    $P[j] = P[j] + t * P[j + 1]$;
    $P[j + 1] = P[j + 1] + t * P[j]$;
  end
end
$P \leftarrow$ **RdmPmtn**(P);
// Computes p_2
$p_2 \leftarrow \lceil \log_2 \|P\|_\infty \rceil$;
$p_2 \leftarrow 2^{63-p_2}$;
// Algorithm ends
return P, p_2;

Algorithm 3: Public key generation

The initial size of the coefficients of P (which is initially S) are inferior or equal to D. After R rounds, it is inferior to $3^R D$. Therefore to encode it, we will need $(n * n)(\log_2(3^R * D) + 1)$ bits (1 extra bit per coefficient due to the sign). We need to add 7 more bits to represent p_2 (by its power value), for a total of $(n * n)(\log_2(3^R * D) + 1) + 7$.

4.2.3 Signature

The signature algorithm is previously described and we will include the details here for completeness.

Input: - A vector $v \in \mathbb{Z}^n$;
- M the noise matrix;
- s a seed value;
Output: - w a reduced vector, with $v \equiv w\ [\mathcal{L}(D+M)]$, and k such that $kP = v - w$;
// Initialization
$w \leftarrow v$;
$i \leftarrow 0$;
$j \leftarrow 0$;
$k \leftarrow [0, ..., 0]$;
// Algorithm start
// Step 1 : Reduce until all coefficients are low enough
while $j \neq n$ **do**
  $j \leftarrow 0$;
  $q \leftarrow \frac{w_i}{d}$;
  $k_i \leftarrow k_i + q$;
  $w_i \leftarrow w_i - qd$;
  for $j = 0$ *to* $n-1$ **do**
    $l \leftarrow i + j \mod n$;
    $w_l \leftarrow w_l + qM_{i,j}$;
    if $|w_l| < d$ **then**
      $j \leftarrow j + 1$;
    end
  end
  $i \leftarrow i + 1 \bmod n$;
end
// Step 2 : use the seed value to modify k accordingly how P was $RdmSeed(x)$;
for $i = 1$; $i < R$; $i = i + 1$ **do**
  $k \leftarrow$ **RdmPmtn**(k);
  for $j = 1$; $j < n - 1$; $j = j + 2$ **do**
    $t \leftarrow$ **RdmSgn()**;
    $k[j+1] = k[j+1] - t * k[j]$;
    $k[j] = k[j] - t * k[j+1]$;
  end
end
$k \leftarrow$ **RdmPmtn**(k);
// Algorithm ends
return k, v, w;

Algorithm 4: Sign

The size of the message is $n(\log_2(\delta) + 1)$, and the size of the signature is the sum of the size of the reduced message vector $n(\log_2(D)+1)$ and the extra information vector k, which is $n(64)$ as explained below $(\log_2 \|k\| < 63)$ which leads to $n(\log_2(D) + 65)$ in signature size.

$$
\begin{aligned}
k'(D-M) &= v-w \\
\|k'\| &\leq \|v-w\|\|(D-M)^{-1}\| \\
&\leq \|v-w\|\|D^{-1}\frac{1}{1-\frac{M}{D}}\| \\
&\leq \|v-w\|\|D^{-1}\|\|\frac{1}{1-\frac{M}{D}}\| \\
&\leq \|v-w\|\|D^{-1}\|\|\|1+\frac{M}{D}+(\frac{M}{D})^2+...\| \\
&\leq \|v-w\|\|D^{-1}\|(\|1\|+\|\frac{M}{D}\|+\|\frac{M}{D}\|^2+...) \\
&\leq \|v-w\|\|D^{-1}\|\|\frac{1}{1-\|\frac{M}{D}\|}\| \\
&\leq \|v-w\|\|\frac{1}{D-\|M\|}\| \\
&\leq \|v-w\|\frac{1}{\Delta} \\
&\leq (\delta+1)\frac{1}{\Delta} = \frac{\delta+1}{\Delta}
\end{aligned}
$$

therefore :

$$
\begin{aligned}
k &= k'U^{-1} \\
\|k\| &\leq \|k'\|\|U^{-1}\| \\
\|k\| &\leq \|\frac{\delta+1}{\Delta}\|\|U^{-1}\| \\
\|k\| &\leq \frac{(\delta+1)3^R}{\Delta}
\end{aligned}
$$

and one can note that $\log_2(\frac{(\delta+1)3^R}{\Delta}) < 63$, from the parameters for δ, Δ, R we proposed earlier, which effectively gives us a 64-bits bound for k.

4.2.4 Verification

Given a hashed message vector v, the signature (k, w), the verification is reduced to the equality test $kP = (v - w)$. However, as the computation kP might overflow (the maximum size of k depends of δ, Δ, R, and P's ones from D, R). In the following verification algorithm we recursively cut k into two parts $k = r + p_2 q$ where p_2 is a power of 2 that is lower than $2^{63}/\|P\|$, which ensures rP is not overflowing.

Given $P, 2^k$ $t = v - w$ and $k = r + p_2 q$ with $\|r\| < p_2$, we have $kP - t = c$ with $c = 0$ if and only if $kP = v - w$. Therefore

$$qp_2P + rP - t = c \quad \rightarrow \quad qP = \frac{c+t-rP}{p_2}$$

and thus p_2 should divide $t - rP$ if $c = 0$: if not, that means $c \neq 0$ and the verification returns **FALSE**. Otherwise, we set $k' \leftarrow q$ and $t' \leftarrow t - rP$ and repeat

$$(qP - \frac{t-rP}{p_2} = \frac{c}{p_2}) \rightarrow (k'P - t' = c')$$

where c' becomes exactly the integer c/p_2 regardless of its value (if it didn't fail before). The verification stops when both $t' = 0$ and $k' = 0$. Note that both need to be 0 at the same time, if only one of them is 0 then the verification fails.

The verification, given k, v, w, P is then as follow:

Input: - A vector $v \in \mathbb{Z}^n$;
- P, p_2 the public key matrix and its associated power of 2;
- w the reduced form of v;
- k the extra information vector;
Output: - w a reduced vector, with $v \equiv w\ [\mathcal{L}(D+M)]$;
`// Algorithm start`
`// Test for max norm first`
if $\|w\|_\infty > D$ **then return** ***FALSE***;
`// Loop Initialization`
$q \leftarrow k$;
$t \leftarrow v - w$;
while $q \neq 0 \wedge t \neq 0$ **do**
  $r \leftarrow q \mod p_2$;
  $t \leftarrow rP - t$;
  `// Check correctness`
  if $t \neq 0 \mod y$ **then return** ***FALSE***;
  $t \leftarrow t/p_2$;
  $q \leftarrow (q - r)/p_2$;
  if $(t = 0) \veebar (q = 0)$ **then return** ***FALSE***;
end
`// Algorithm ends`
return ***TRUE***;

Algorithm 5: Verify

4.2.5 Potential speedups and modifications

The first one, would be to use the seed for the generation of the secret key that we reuse for the signature scheme. That way, we would have no need to store the sign data and recover it on the fly. This would transform a quadratic size memory part of the secret key to a constant size part. In experimentations however, this has increased the signing time significantly and therefore we have decided to not apply it.

The second one is to change the reduction order to a random one each time (i.e from $m[1], m[2]..., m[n]$ successively to $m[\rho(1)], m[\rho(2)]..., m[\rho(n)]$ where rho is a random permutation) : this would barely slow down the algorithm reduction but provide an extra layer of security against side-channel attacks. On top of that, experimentations showed that given a vector v, a valid answer w is not unique: therefore we can also choose to compute some extra steps at certain randomly chosen positions to blur the amount of computations actually done to solve $\mathbf{GDD}_\gamma$. However, one need to ensure that the process is deterministic assuming fixed parameters.

A third one would be to change the unimodular matrices we use for both the verification and the public key generation: we could use bigger blocks (i.e not $2 * 2$) that could be better balanced.

4.2.6 KAT files, speed tests and architecture

To build the KAT files, we use the Makefile provided by the NIST (as described in the example) along with the files $rng.c$, $rng.h$ and $PQCgenKAT\_sign.c$ provided by the NIST, and combine them with our own written files (all $.c$ and $.h$) in the same folder.

As far as the speed tests are concerned, we used the following options:

WARN_OPTS = -Wall -Wextra -Wno-format-overflow -Wno-sign-compare -Wno-unused-but-set-variable
-Wno-unused-but-set-variable -Wno-unused-parameter -pedantic -Wno-parentheses
CFLAGS = -std=c11 $(WARN_OPTS) -fno-verbose-asm -march=native -Ofast
-funroll-loops
LDFLAGS = -lssl -lcrypto

For the following speed tests, we only use the functions *crypto_sign_keypair*, *crypto_sign_open*, *crypto_sign* as defined by the NIST. A fraction of the time is thus spent in allocating structures and deallocating them. Setup are done with 10^3 keys per dimension and security parameter, and for each of those keys 10 signatures and verifications are done for a total of 10^4 signatures and 10^4. Times are displayed in seconds.

Dimension	512	768	1024	1280	1536	1792	1920	2048
Setup	48.680	100.553	188.313	319.1648	503.852	695.525	714.524	818.250
Signature	31.678	55.679	98.144	165.895	269.771	368.692	365.248	396.058
Verification	350.312	707.984	1203.507	1932.834	3072.41937	4234.666	4183.861	5019.465

Figure 1: Tests for $\lambda = 128$, $R = 24$, $\Delta = 32$, $\delta = 28$

Dimension	512	768	1024	1280	1536	1792	1920	2048
Setup	44.530	100.072	183.321	294.075	431.868	599.778	688.423	791.948
Signature	28.209	53.924	91.771	144.400	210.276	289.601	338.314	374.911
Verification	320.416	702.817	1137.230	1835.790	2645.116	3620.419	4138.517	4910.150

Figure 2: Tests for $\lambda = 192$, $R = 24$, $\Delta = 32$, $\delta = 28$

Dimension	512	768	1024	1280	1536	1792	1920	2048
Setup	44.593	104.411	210.255	309.667	451.837	614.799	726.933	832.339
Signature	27.475	55.251	108.2089	152.499	221.203	295.443	355.666	385.401
Verification	316.5690	730.978	1392.252	1906.629	2745.214	3694.047	4267.227	5058.264

Figure 3: Tests for $\lambda = 256$, $R = 24$, $\Delta = 32$, $\delta = 28$

The command "*lscpu*" gave us the following information about the processor we used to make those tests:

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 2
- Core(s) per socket: 4
- Socket(s): 2
- NUMA node(s): 2

- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 44
- Model name: Intel(R) Xeon(R) CPU X5647 @ 2.93GHz
- Stepping: 2
- CPU MHz: 2925.883
- BogoMIPS: 5851.76
- Virtualisation: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 12288K
- NUMA node0 CPU(s): 0,2,4,6,8,10,12,14
- NUMA node1 CPU(s): 1,3,5,7,9,11,13,15
- Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 popcnt aes lahf_lm tpr_shadow vnmi flexpriority ept vpid dtherm ida arat

The command "*cat /proc/meminfo*" gave us this information about the memory capacity we used to make those tests:

- MemTotal: 16414628 kB
- MemFree: 13267160 kB
- MemAvailable: 15789276 kB
- Buffers: 149600 kB
- Cached: 2426328 kB
- SwapCached: 0 kB
- Active: 553700 kB
- Inactive: 2045192 kB
- Active(anon): 27392 kB
- Inactive(anon): 8804 kB
- Active(file): 526308 kB
- Inactive(file): 2036388 kB
- Unevictable: 5312 kB

- Mlocked: 5312 kB
- SwapTotal: 16761852 kB
- SwapFree: 16761852 kB
- Dirty: 0 kB
- Writeback: 0 kB
- AnonPages: 28288 kB
- Mapped: 33676 kB
- Shmem: 9284 kB
- Slab: 463324 kB
- SReclaimable: 354528 kB
- SUnreclaim: 108796 kB
- KernelStack: 4416 kB
- PageTables: 3412 kB
- NFS_Unstable: 0 kB
- Bounce: 0 kB
- WritebackTmp: 0 kB
- CommitLimit: 24969164 kB
- Committed_AS: 463396 kB
- VmallocTotal: 34359738367 kB
- VmallocUsed: 0 kB
- VmallocChunk: 0 kB
- HardwareCorrupted: 0 kB
- AnonHugePages: 0 kB
- ShmemHugePages: 0 kB
- ShmemPmdMapped: 0 kB
- CmaTotal: 0 kB
- CmaFree: 0 kB
- HugePages_Total: 0
- HugePages_Free: 0
- HugePages_Rsvd: 0
- HugePages_Surp: 0
- Hugepagesize: 2048 kB
- DirectMap4k: 157156 kB

- DirectMap2M: 6121472 kB
- DirectMap1G: 10485760 kB

The command "*cat /etc/os-release*" gave us this information about the operating system we used to make those tests:

- NAME="Ubuntu"
- VERSION="17.10 (Artful Aardvark)"
- ID=ubuntu
- ID_LIKE=debian
- PRETTY_NAME="Ubuntu 17.10"
- VERSION_ID="17.10"
- HOME_URL="https://www.ubuntu.com/"
- SUPPORT_URL="https://help.ubuntu.com/"
- BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
- PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
- VERSION_CODENAME=artful
- UBUNTU_CODENAME=artful

4.2.7 NIST-approved primitives (random and hashes)

We used the random chars generators from *rng.c* and *rng.h* that were provided to us, along with the KAT files. Everytime we initialize our random functions, we use the NIST-provided generators to obtain a pool of random bits where we can extract as many bits as we want. Once we detect that the pool is depleted, we generate a fresh pool without changing the seed. Here's how we implemented our random generators:

- **RdmSgn** reads one bit b from our pool and returns $(2b-1)$.
- **RdmPmtn** fills a global array A of size n with random values such that $A[i] \leq i$.
  We generate $\rho \in S_n$ from A by computing the product $\rho = (1\ (1+A[n-1]))(2\ (2+A[n-2]))...(n\ A[1])$.

When hashing a random message to the message space, we used $SHAKE512$ to guarantee 256-bits collision resistance.

References

[1] Richard A Brualdi and Herbert J Ryser. *Combinatorial matrix theory*, volume 39. Cambridge University Press, 1991.

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

[3] Nicolas Gama and Phong Q Nguyen. Predicting lattice reduction. In *Advances in Cryptology–EUROCRYPT 2008*, pages 31–51. Springer, 2008.

[4] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *Advances in Cryptology - CRYPTO'97*, pages 112–131. Springer, 1997.

[5] Jeff Hoffstein, Jill Pipher, John M Schanck, Joseph H Silverman, William Whyte, and Zhenfei Zhang. Choosing parameters for ntruencrypt. In *Cryptographers' Track at the RSA Conference*, pages 3–18. Springer, 2017.

[6] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of operations research*, 12(3):415–440, 1987.

[7] Vadim Lyubashevsky and Daniele Micciancio. On bounded distance decoding, unique shortest vectors, and the minimum distance problem. In *Advances in Cryptology-CRYPTO 2009*, pages 577–594. Springer, 2009.

[8] Phong Q Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of ggh and ntru signatures. *Journal of Cryptology*, 22(2):139–160, 2009.

[9] Thomas Plantard, Willy Susilo, and Khin Than Win. A digital signature scheme based on cvp max). In *International Workshop on Public Key Cryptography*, pages 288–307. Springer, 2008.

[10] Joop van de Pol and Nigel P Smart. Estimating key sizes for high dimensional lattice-based systems. In *IMA International Conference on Cryptography and Coding*, pages 290–303. Springer, 2013.