

DRS: Diagonal dominant Reduction for lattice-based Signature

Version 2

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 B of the form a diagonal dominant matrix as in [1], i.e

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

We can also see a diagonally dominant matrix B as a sum $B = D + R$ where D is diagonal and $D_{i,i} > \|R_i\|_1$. 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:

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

l_1 and l_2 are commonly used and are often called taxicab norm and euclidean norm respectively. 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 **BDD $_{1/(2\gamma)}$** reduces itself to **uSVP $_{\gamma}$** in polynomial time and the same goes from **uSVP $_{\gamma}$** to **BDD $_{1/\gamma}$** when γ is polynomially bounded by n [8], 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 security is concerned, the **GDD $_{\gamma}$** is more relevant:

Definition 8 (**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 **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 **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. [10], which was originally inspired by GGH [4].

In this section, we just give a quick overview of our algorithms and explain 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 random seed

Note that this new setup for the secret key have been strongly changed following the attack from [14].

The new algorithm here is less straightforward but still the same as its core, we just have to generate a diagonal dominant matrix S of size $n * n$ with a low and bounded noise, given a chosen bound D and a value $\delta < D$. This is done by computing $S = D + M$ where M is chosen uniform among all possibilities verifying

$$\forall i \in [1, n], \|M_i\|_1 < D$$

To achieve this, we will just generate each vector $S_i = M_i + D_i$ with $i \in [1, n]$ in simple steps, but using a precomputation first:

1. We generate a table of numbers T such that $T[i]$ is the number of vectors of \mathbb{Z}^n with i zeroes.
2. We then generate a table of numbers T_S such that $T_S[i]$ is the number of vectors of \mathbb{Z}^n with i zeroes or less.

And now, from T_S we generate each vector S_i in simple steps:

1. We pick a random value $x \in [T_S[1] : T_S[n]]$, and select k $T_S[k - 1] < x \leq T_S[k]$.
2. We pick $n - k$ values x_1, \dots, x_{n-k} such that $0 < x_0 < \dots < x_{n-k} \leq D$.

3. We fill $S_i = [x_1, x_2 - x_1, x_3 - x_2, \dots, x_{n-k} - x_{n-k-1}, 0, \dots, 0]$.
4. We then multiply every element $S_{i,1}, \dots, S_{i,n-k}$ by 1 or -1 (just changing the sign randomly).
5. The last step consists in permuting randomly coefficients of S_i , and assigning $S_{i,i} \leftarrow S_{i,i} + D$.

As part of the secret key, we also keep the seed value s 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 M_{max} 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 $3\|M\|_\infty$, 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 security parameter 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 smaller noise $|qb|$ and $|q|$ in some other coefficients $m[j]$ with $j \neq i$, i.e

applying $m \leftarrow m - qS_i$ where $|m_i - q * d| < d$ for every $|m_i| > d$ until $\|m\|_\infty < d$.

All proofwork can be found in the paper of the original reduction idea [10], 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 where M has a zero diagonal.

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 $d > \sum_{j=1, j \neq i}^n |M_{i,j}|$

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

$$\text{and } d < |m_i|, \text{ thus } |m_i| - |qd| < |m_i|$$

$$\|m\|_1 - |qd| + |q| \sum_{j=1, j \neq i}^n |M_{i,j}| < \|m\|_1$$

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

This new secret key, however, do not have a constant element in its diagonal. Obviously, the matrix will still be diagonal dominant in any case. Let us denote d_i the diagonal coefficient $S_{i,i}$ of $S = D - M$.

If $d > d_i$ we can use the previous reasoning and reduce $|m_i|$ to $|m_i| < d_i < d$, but keep in mind we stop the reduction at $|m_i| < d$ to ensure we do not leak information about the noise distribution.

Now $d_i > d$ for some i : reducing to $|m_i| < d_i$ is guaranteed but not sufficient anymore as we can reach $d < |m_i| < d_i \leq d + \Delta < 2d$. Let us remind that $\Delta = d - \sum_{j=1}^n |M_{i,j}|$, where Δ is strictly positive as an initial condition of the DRS signature scheme (both on the original submission and this paper), $d_i = d + c$ where $c = |M_{i,i}|$.

Without loss of generality as we can flip signs, let us set $m_i = d + k < d_i = d + c$ with $k \geq 0$ the coefficient to reduce. Subtracting by S_i transforms

$$m_i \leftarrow (d + k) - d_i = (d + k) - (d + c) = k - c < 0$$

with $d > c > k \geq 0$. Therefore the reduction of $\|m\|_1$ without the noise is

$$\|m\|_1 \leftarrow \|m\|_1 - (d + k) + (c - k) = \|m\|_1 - (d - c) - 2k.$$

but the noise contribution on other coefficients, at worst, is $(d - \Delta) - c$ which added does

$$\|m\|_1 \leftarrow \|m\|_1 - (d - c) - 2k + (d - c - \Delta).$$

$$\|m\|_1 \leftarrow \|m\|_1 - 2k - \Delta = \|m\|_1 - (2k + \Delta).$$

where $2k + \Delta > 0$. Therefore the reduction is also ensured in the case $d_i > d$.

Once we have s the reduced form of m , 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 $k = [0, \dots, 0]$. 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

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

we have

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

(note that order matters), and knowing

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

U^{-1} is thus as easy to compute as U , and also give the exact same bound of 3^R for the growth of the maximum matrix norm after R rounds, 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 $k \leftarrow k P_{s,1}^{-1} T_{s,1}^{-1}$
4. Repeat the last two steps $R - 1$ more times and reapply one final inverse permutation $k \leftarrow k P_{s,R+1}^{-1}$

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 [9] in Plantard et al's suggestion at PKC2008 [10] to fix GGHSig [4]. Their original theoretical framework is still unchallenged however its instantiation in the form of the original DRS submission was severely weakened by Yu and Ducas' attack [14]. In the following subsection we will describe the state of the art method, in the best of our knowledge to attack this new version of DRS.

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 **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 $(0, \dots, 0, d, 0, \dots, 0)$ 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 $v = (0, \dots, 0, d, 0, \dots, 0)$

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

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

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

$$\gamma \approx \frac{\Gamma\left(\frac{n+3}{2}\right)^{\frac{1}{n+1}} \text{Det}(\mathcal{L})^{\frac{1}{n+1}}}{\sqrt{\pi} \|M_1\|_2} \approx \frac{\Gamma\left(\frac{n+3}{2}\right)^{\frac{1}{n+1}} d^{\frac{1}{n+1}}}{\sqrt{\pi} \|M_1\|_2}. \quad (1)$$

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_{γ} 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.

As we do not have a fixed euclidean norm for our secret vectors we have to rely on the approximates given to us by our new random method in sampling noise vectors M_i .

In our case, we know that for any vector $v \in \mathbb{Z}^n$ we have $\|v\|_2 \geq \frac{\|v\|_1}{\sqrt{n}}$, and our experiments (as seen below) allow us to use a higher bound

$$\|v\|_2 \gtrsim \sqrt{2} \frac{\|v\|_1}{\sqrt{n}}$$

Dimension	Δ	R	δ	γ	2^λ
1108	1	24	28	$< \frac{1}{4}(1.006)^{d+1}$	2^{128}
1372	1	24	28	$< \frac{1}{4}(1.005)^{d+1}$	2^{192}
1779	1	24	28	$< \frac{1}{4}(1.004)^{d+1}$	2^{256}

Table 1: Parameter Sets.

3.2 Expected Security Strength

Different papers are giving some relations between the Hermite factor and the security parameter λ [5, 13] 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 $D = n$, pick hashed messages $h(m)$ such that $\log_2(\|h(m)\|_\infty) = 28$, $R = 24$ and $\Delta = 1$.

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

Our experiments shows us that the distribution of zeroes among sampled noise vectors form a Gaussian and so does the euclidean norm of noise vectors when picking our random elements x, x_i uniformly.

Here we include below the distribution of 10^6 randomly generated noise vectors v with the x-axis representing $f(v) = \lfloor 100\sqrt{\frac{\|v\|_2^2}{D}} \rfloor$ where D is the signature bound.

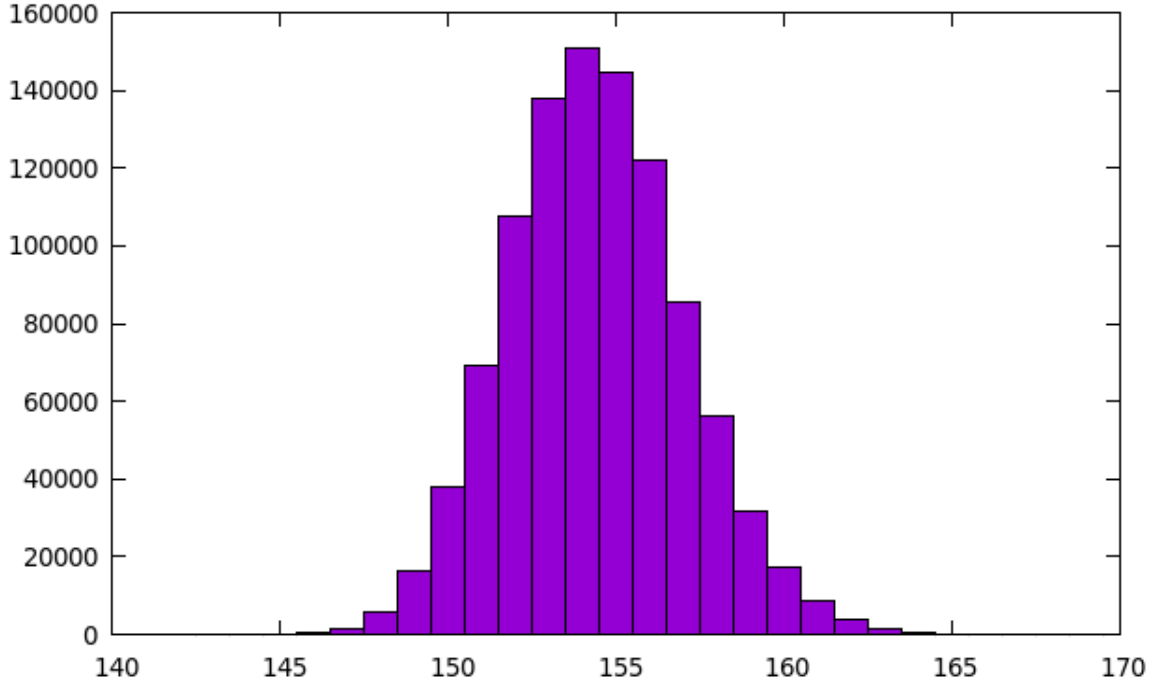


Figure 1: Distribution of $f(v)$ for dimension $n = 1108$ and bound $D = n - 1$ over 10^6 samples

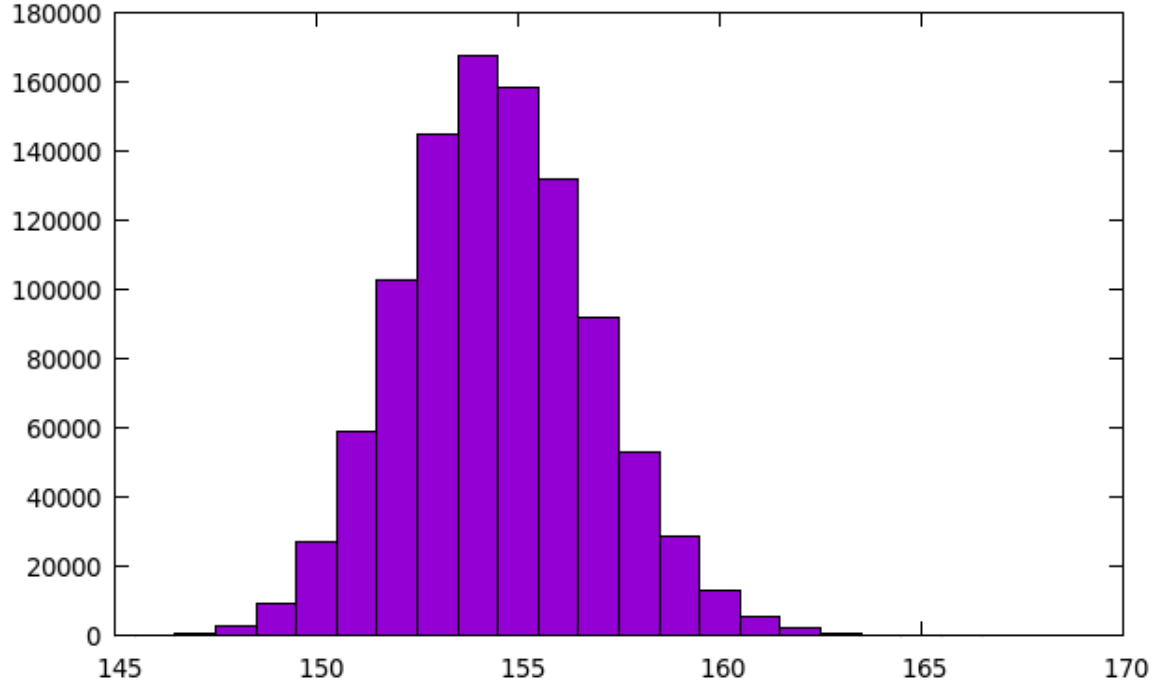


Figure 2: Distribution of $f(v)$ for dimension $n = 1372$ and bound $D = n - 1$ over 10^6 samples

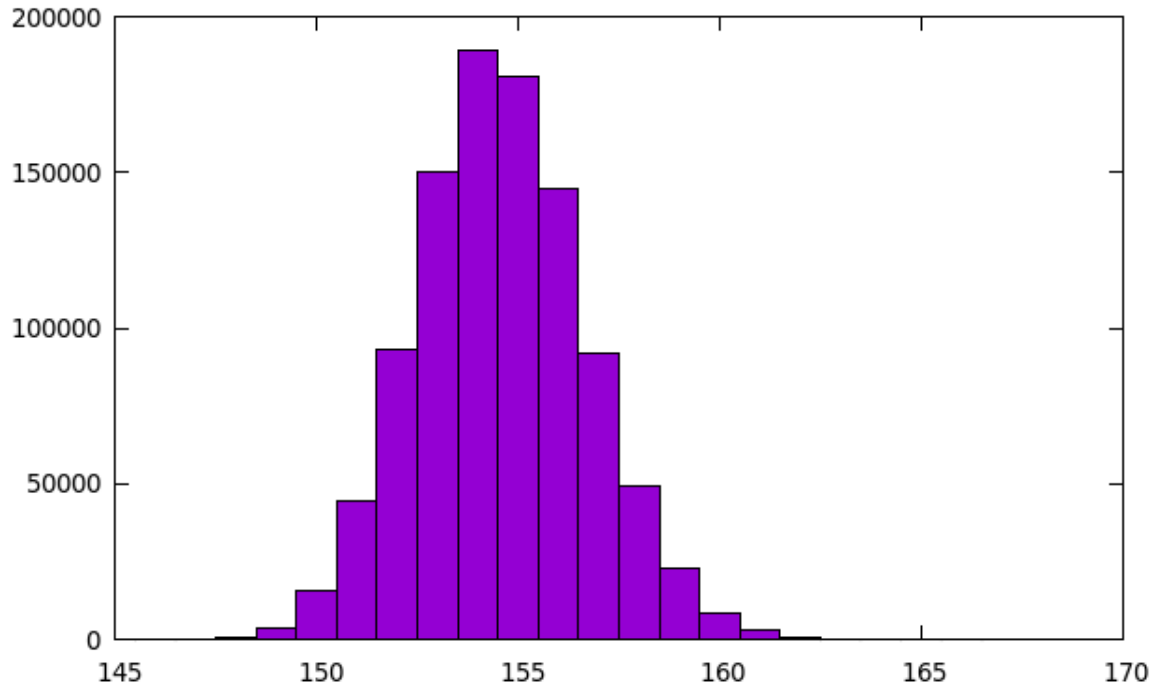


Figure 3: Distribution of $f(v)$ for dimension $n = 1779$ and bound $D = n - 1$ over 10^6 samples

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 6 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
- Δ : a parameter that defines an extra sparsity in our reduction matrix, which limits the taxicab norm of the noise vectors by $D - \Delta$. We usually set $\Delta = 1$ as it is the most secure choice.
- R : a "round" number, indicating the number of loop iterations used to generate the public key.

All those parameters can be found in the various *api.h* files available. Those include the precomputed tables T_S used for the generation of the secret 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 $D = n$, $\delta = 28$, $\Delta = 1$ 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

As previous readers of the original DRS report have noticed, the secret key setup was changed completely due to Yu and Ducas' attack [14]. Yu and Ducas' attack was exploiting the specific structure of the noise vector, which was composed of only two unique values (modulo signs). We are now using noise vectors taken randomly in the set of $v \in \mathbb{Z}^n$ such that $\|v\|_1 < D$.

The set of noise vectors we need to keep are all the vectors v that have a taxicab norm of $0 \leq \|v\|_1 \leq D-1$ and dimension n . Let us call that set V_n . The new noise matrix M we are aiming to build is a $n \times n$ matrix such that $M \in V_n^n$.

In that regard, we construct a table we will call T with D entries such that

$$T[i] = \#\text{vectors } v \in V_n \text{ with } i \text{ zeroes}.$$

This table is relatively easy to build and do not take much time, one can for example use the formulas derivated from [11] and [7]. From this table, we construct another table T_S such that $T_S[k] = \sum_{i=0}^k T[i]$. Below is the generation algorithm of the table T_S , which we will use as a precomputation for our new setup algorithm:

```

Input: - all initial parameters;
Output: -  $T_S$  the table sum;
// Initialization
 $m \leftarrow \min(\text{dimension}, \text{diagonal coef } D)$ ;
 $T \leftarrow \{1\}^{m+1}$ ;
 $T_S \leftarrow \{1\}^{m+1}$ ;
// Construct array  $T$ 
// Construct array  $T$  : loop over the norm
for  $j = 2 ; j \leq D ; j = j + 1$  do
    // Construct array  $T$  : loop over the number of non-zeroes elements in each
    // possibility
    for  $i = 2 ; i \leq m + 1 ; i = i + 1$  do
         $x \leftarrow 2^{i-1} \binom{n}{i-1} \binom{j-1}{i-2}$ ;
         $T[m + 1 - i] \leftarrow T[m + 1 - i] + x$ ;
    end
end
// Construct array  $T_S$  from  $T$ 
for  $i = 1 ; i \leq m ; i = i + 1$  do
     $T_S[i + 1] \leftarrow T_S[i + 1] + T[i]$ ;
end
 $T_S \leftarrow T_S$ ;
// Algorithm ends
return  $T_S$ ;

```

Algorithm 2: Secret key table precomputation

Let us denote the function $Z(x) \rightarrow y$ such that $T_S[y - 1] < x \leq T_S[y]$. Since T_S is trivially sorted in increasing order $Z(x)$ is nothing more than a dichotomy search inside an ordered table.

If we pick randomly x from $[0; T_S[D - 1]]$ from a generator with uniform distribution $g() \rightarrow x$ then we got $Zero() \rightarrow Z(g(x))$ a function that selects uniformly an amount of zeroes amount all vectors of the set V_n , i.e

$$Zero() \rightarrow \#\text{zeroes in a random } v \in V_n.$$

Now that we can generate uniformly the number of zeroes we have to determine the coefficients of the non-zero values randomly, while making sure the final noise vector is still part of V_n . A method to give such a vector with chosen taxicab norm is given in [12] as a correction of the Kraemer algorithm. As we do not want to choose the taxicab norm M directly but rather wants to have any random norm available, we add a slight modification: the method in [12] takes k non-zero elements x_1, \dots, x_k such that $x_i \leq x_{i+1}$ and forces the last coefficient to be equal to the taxicab norm chosen, i.e $x_k = M$. By removing the restriction and using $x_k \leq D$, giving the amount of non-zero values, we modify the method to be able to take over any vector values in V_n with the help of a function we will call

$$\text{KraemerBis}(z) \rightarrow \text{random } v \in V_n \text{ such that } v \text{ has } z \text{ zeroes}$$

which is described below in the following algorithm

```

Input: - all initial parameters;
- a number of zeroes  $z$ ;
Output: - a vector  $v$  with  $z$  zeroes and a random norm inferior or equal to  $D$ ;
// Algorithm start
 $v \in \mathbb{N}^n$ ;
Pick randomly  $n - z + 1$  elements such that  $0 \leq x_0 < x_1 < \dots < x_{n-z} \leq D$ ;
for  $i = 1 ; i \leq n - z ; i = i + 1$  do
     $v[i] \leftarrow x_i - x_{i-1}$ ;
end
for  $i = n - z + 1 ; i \leq n ; i = i + 1$  do
     $v[i] \leftarrow 0$ ;
end
// Algorithm ends
return  $v$ ;

```

Algorithm 3: KraemerBis

With both those new parts, the new setup algorithm we construct is the following:

```

Input: - all initial parameters;
- another extra random seed  $x_2$ ;
Output: -  $x, S$  the secret key;
// Initialization
 $S \leftarrow D \times Id_n$ ;
 $t \in \mathbb{Z}^n$ ;
// Algorithm start
InitiateRdmSeed( $x_2$ );
for  $i = 1 ; i \leq n ; i = i + 1$  do
    // Get the number of zeroes and create a new vector
     $Z \leftarrow \text{Zero}()$ ;
     $t \leftarrow \text{KraemerBis}(Z)$ ;
    // Randomly switch coefficient signs
    for  $j = 1 ; j \leq n - Z ; j = j + 1$  do
         $t[j] \leftarrow t[j] \times \text{RdmSgn}()$ 
    end
    // RdmPmtn permutes everything
     $t \leftarrow \text{RdmPmtn}(t)$ ;
     $S_i \leftarrow S_i + t$ ;
end
// Algorithm ends
return  $x, S$ ;

```

Algorithm 4: New secret key generation (square matrix of dimension n)

We note that in our new secret key, the structure is less present and we cannot compress the way we did in the initial iteration of DRS.

The secret key is a square matrix where every element is within $[-2D, 2D]$, however as we only store the noise, we consider elements in $[-D + 1, D - 1]$, 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 $n^2 \lceil \log_2(2D) + 1 \rceil + N_x$.

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: -  $S$  the reduction matrix of dimension  $n$ , obtained previously;
- a random seed  $x$ ;
Output: -  $P$  the public key, and  $p_2$  a power of two;
// Initialization
 $P \leftarrow S$ ;
// Algorithm start
InitiateRdmSeed( $x$ );
// Apply  $R$  rounds
for  $i = 1 ; i < R ; i = i + 1$  do
     $P \leftarrow \text{RdmPmtn}(P)$ ;
    for  $j = 1 ; j \leq n - 1 ; j = j + 2$  do
         $t \leftarrow \text{RdmSgn}()$ ;
         $P[j] = P[j] + t * P[j + 1]$ ;
         $P[j + 1] = P[j + 1] + t * P[j]$ ;
    end
end
 $P \leftarrow \text{RdmPmtn}(P)$ ;
// 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 5: 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 $n^2 \lceil \log_2(3^R * D) + 1 \rceil$ bits (1 extra bit per coefficient due to the sign). Unlike the previous DRS iteration, we do not need to add 7 more bits to represent p_2 (by its power value) as we can compute it when reading the key's data, for a total of $n^2 \lceil \log_2(3^R D) + 1 \rceil$.

4.2.3 Signature

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

```

Input: - A vector  $v \in \mathbb{Z}^n$ ;
-  $S$  the secret key matrix, with diagonal coefficient  $d$ ;
-  $s$  a seed value;
Output: -  $w$  with  $v \equiv w [\mathcal{L}(S)]$ ,  $\|w\|_\infty < d$  and  $k$  with  $kP = v - w$ ;
// Initialization
 $w \leftarrow v$ ;
 $i \leftarrow 0$ ;
 $k \leftarrow [0, \dots, 0]$ ;
// Algorithm start
// Reduce until all coefficients are low enough
while  $\|w\|_\infty < d$  do
     $q \leftarrow \frac{w_i}{d}$ ;
     $k_i \leftarrow k_i + q$ ;
     $w \leftarrow w - qS_i$ ;
     $i \leftarrow i + 1 \bmod n$ ;
end
// Use the seed to modify  $k$  such that  $kP = v - w$ 
// The seed defines the output of RdmPmtn and RdmSgn
InitiateRdmSeed( $x$ );
for  $i = 1 ; i \leq R ; i = i + 1$  do
     $k \leftarrow \text{RdmPmtn}(k)$ ;
    for  $j = 1 ; j \leq n - 1 ; j = j + 2$  do
         $t \leftarrow \text{RdmSgn}()$ ;
         $k[j + 1] = k[j + 1] - t * k[j]$ ;
         $k[j] = k[j] - t * k[j + 1]$ ;
    end
end
 $k \leftarrow \text{RdmPmtn}(k)$ ;
// Algorithm ends
return  $k, v, w$ ;

```

Algorithm 6: Sign : coefficient reduction first, validity vector then

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

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

therefore :

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

and one can note that $\log_2\left(\frac{(\delta+1)3^R}{\Delta}\right) < 68$ with $\Delta = 1$, from the parameters for δ, Δ, R we proposed earlier, which effectively gives us a 68-bits bound for k . However in practice, 3^R is a heavy overestimation and can be easily replaced in average by 2.5^R and thus give us in practice 60 bits, which is below 64.

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 $k = r + p_2q$ 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 $k = r + p_2q$ with $\|r\| < p_2$, we have $kP - t = c$ with $c = 0$ if and only if $kP = v - w$. Therefore

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

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
     $r \leftarrow q \bmod p_2$ ;
     $t \leftarrow rP - t$ ;
    // Check correctness
    if  $t \neq 0 \bmod y$  then return FALSE;
     $t \leftarrow t/p_2$ ;
     $q \leftarrow (q - r)/p_2$ ;
    if  $(t = 0) \vee (q = 0)$  then return FALSE;
end
// Algorithm ends
return TRUE;

```

Algorithm 7: 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, \dots, m_n successively to $m_{\rho(1)}, m_{\rho(2)}, \dots, m_{\rho(n)}$ where ρ 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}_γ . 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 -Ofast
            -funroll-loops
LDFLAGS = -lssl -lcrypto

```

We do not only use the option `-march = native`, as we want to distinguish the time with `AVX512` and without it. It does seem that the compiler do use the `AVX512` instructions for us when we add the following options: `AVX512FLAGS = -mavx512f -mavx512dq -mavx512cd -mavx512bw -mavx512vl`.

For the following speed tests, we only use the functions `crypto_sign_keypair`, `crypto_sign_open`, `crypto_sign` as defined by the NIST. Setup are done with 10^4 keys per dimension and security parameter, and for each of those keys 1 signature and verification is done for a total of 10^4 signatures and 10^4 . Times are displayed in seconds. We set in black results without `-march = native` and `AVX512FLAGS`, and in **red** results with `AVX512FLAGS` but without `-march = native`, and **blue** for results with `-march = native`.

We also colored in light blue the columns relevant to the algorithms we described. There are other algorithms which take a significant amount of time out of the total:

- *Init* initialize the memory before all operations can be applied. Using Intel intrinsics for memory allocation (i.e `_mm_malloc`) will greatly improve performance but as we let gcc apply the AVX instructions for us we did not bother.
- *Write* functions writes our data structures into a portable character array. Those were a time sink in the first iteration of DRS, and still are even though they have been improved for the most part.
- *Read* functions read characters arrays to port them into our data structures. Like *Write*, those were also time killers.
- *hash* is basically the non-optimised compact version of *SHAKE512* available on the KECCAK website, although we did modify the parameters to reach a 256-bit security (the original code had *SHAKE256*).

It is worth mentioning that if those were ever dropped algorithms would seem more efficient than the table total results show. For example, one can easily imagine a case where one key pair would be read only once, and used to sign and verify thousands of messages and signatures: in that case, *Init*, *Write* and *Read* algorithms for keys could be used only once per 10^4 messages/signatures and not 10^4 times as presented here.

<i>Dimension</i>	<i>TOTAL</i>	<i>Init</i>	<i>SecretKey</i>	<i>PublicKey</i>	<i>WriteSecretKey</i>	<i>WritePublicKey</i>
1108	787.80	29.42	599.57	110.02	15.84	20.24
1108	794.97	30.00	614.61	100.76	16.37	20.53
1108	784.41	31.47	606.96	100.31	16.28	16.34
1372	1215.23	56.51	890.24	184.73	25.10	31.60
1372	1222.69	57.10	911.21	169.49	25.95	31.72
1372	1198.09	57.54	889.05	169.80	28.77	25.58
1779	1979.86	104.91	1344.84	386.56	42.02	54.10
1779	1984.83	107.06	1377.22	355.31	43.79	54.14
1779	1970.66	106.75	1374.24	355.12	45.17	44.08

Figure 4: Time for setup, 10^4 keys

We note that the setup of the secret key is not gaining at all from vectorization, which is not surprising as it has no purely linear operations in a loop. Mostly permutations.

<i>Dimension</i>	<i>TOTAL</i>	<i>Init</i>	<i>Hash</i>	<i>Sign</i>	<i>WriteSignature</i>	<i>ReadSecretKey</i>
1108	42.09	2.59	5.64	21.80	0.045	11.97
1108	36.08	2.50	5.88	15.80	0.043	11.81
1108	35.18	2.57	5.90	15.73	0.040	10.90
1372	63.79	7.86	6.91	31.49	0.05	17.44
1372	55.43	7.95	7.20	21.98	0.05	18.21
1372	55.93	7.97	7.20	21.70	0.05	18.97
1779	104.96	18.95	8.70	48.45	0.067	28.74
1779	92.12	18.97	9.03	32.57	0.07	31.39
1779	89.31	18.93	9.04	32.39	0.07	28.80

Figure 5: Time for signature, 10^4 signatures (1 per key)

<i>Dimension</i>	<i>TOTAL</i>	<i>Init</i>	<i>Hash</i>	<i>Verification</i>	<i>ReadSignature</i>	<i>ReadPublicKey</i>
1108	62.44	16.45	5.62	12.90	0.03	23.63
1108	58.54	16.83	5.76	8.09	0.03	24.22
1108	55.51	16.78	5.78	8.15	0.03	21.08
1372	101.55	29.58	6.89	22.37	0.05	37.14
1372	95.45	30.31	7.08	13.67	0.04	38.60
1372	89.73	30.29	7.07	13.67	0.04	32.81
1779	181.86	58.31	8.66	42.74	0.06	63.29
1779	172.48	59.37	8.90	26.88	0.065	68.37
1779	157.27	58.79	8.91	26.23	0.06	55.58

Figure 6: Time for verification, 10^4 verification (1 per key)

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): 48
- On-line CPU(s) list: 0-47
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 4
- NUMA node(s): 4
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 85
- Model name: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
- Stepping: 4

- CPU MHz: 1201.687
- BogoMIPS: 6800.00
- Virtualisation: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 1024K
- L3 cache: 19712K
- NUMA node0 CPU(s): 0,4,8,12,16,20,24,28,32,36,40,44
- NUMA node1 CPU(s): 1,5,9,13,17,21,25,29,33,37,41,45
- NUMA node2 CPU(s): 2,6,10,14,18,22,26,30,34,38,42,46
- NUMA node3 CPU(s): 3,7,11,15,19,23,27,31,35,39,43,47
- 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 art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_single pti intel_ppin ssbd mba ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts pku ospke flush_lld

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

- MemTotal: 65644480 kB
- MemFree: 64727784 kB
- MemAvailable: 64594512 kB
- Buffers: 41804 kB
- Cached: 285372 kB
- SwapCached: 0 kB
- Active: 212228 kB
- Inactive: 184900 kB
- Active(anon): 70388 kB
- Inactive(anon): 2828 kB
- Active(file): 141840 kB
- Inactive(file): 182072 kB
- Unevictable: 0 kB

- Mlocked: 0 kB
- SwapTotal: 8388604 kB
- SwapFree: 8388604 kB
- Dirty: 28 kB
- Writeback: 0 kB
- AnonPages: 70192 kB
- Mapped: 67208 kB
- Shmem: 3236 kB
- Slab: 253820 kB
- SReclaimable: 86456 kB
- SUNreclaim: 167364 kB
- KernelStack: 9776 kB
- PageTables: 4084 kB
- NFS_Unstable: 0 kB
- Bounce: 0 kB
- WritebackTmp: 0 kB
- CommitLimit: 41210844 kB
- Committed_AS: 659616 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: 200512 kB

- DirectMap2M: 6817792 kB
- DirectMap1G: 61865984 kB

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

- NAME="Ubuntu"
- VERSION="18.04.1 LTS (Bionic Beaver)"
- ID=ubuntu
- ID_LIKE=debian
- PRETTY_NAME="Ubuntu 17.10"
- VERSION_ID="18.04"
- 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=bionic
- UBUNTU_CODENAME=bionic

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 with a shift and/or a mask from the pool to transform it into either 1 or -1 .
- **RdmPmtn** apply $\rho = (1\ a_{n-1})(2\ a_{n-2})...(n\ a_0) \in S_n$ where a_i is a random value in $[n - i, n]$.

When hashing a random message to the message space, we used *SHAKE512* to guarantee 256-bits collision resistance. The code is taken from the git directory from the creators of KECCAK.

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] Donald Erwin Knuth, Ronald L Graham, Oren Patashnik, et al. Concrete mathematics. *Adison Wesley*, 1989.
- [8] 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.
- [9] Phong Q Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of ggh and ntru signatures. *Journal of Cryptology*, 22(2):139–160, 2009.
- [10] 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.
- [11] Joan Serra-Sagristà. Enumeration of lattice points in l1 norm. *Information processing letters*, 76(1-2):39–44, 2000.
- [12] Noah A Smith and Roy W Tromble. Sampling uniformly from the unit simplex. 2004.
- [13] 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.
- [14] Yang Yu and Léo Ducas. Learning strikes again: the case of the drs signature scheme. Cryptology ePrint Archive, Report 2018/294, 2018.