Sampling algorithm from a discrete normal distribution over lattices

Morgane Guerreau

January 2021

Introduction

Discrete Gaussian sampling is an algorithmic brick essential in lattice-based hash-and-sign signature schemes. As shown by Gentry, Peikert and Vaikuntanathan [GPV08], such a sampling algorithm over lattices allows to build a trapdoor function based on the problem "Short Integer Solution" (SIS). The trapdoor consists in sampling a solution SIS from a lattice's secret short basis.

The security of this trapdoor function is based on the worst-case difficulty of the "Shortest Vector Problem" (SVP) for an approximation factor determined by the standard deviation of the Gaussian distribution obtained in output of the sampling algorithm over the lattice. There is several sampling algorithm over lattices which differ by the standard deviation obtainable in output and their ability to benefit from the algebraic structure of some bases. The sampling algorithm ffSampling used in the Falcon signature scheme allows to achieve a small standard deviation and takes advantage of the algebraic structure of NTRU basis.

Contents

1 Theoretical explanations

1.1 Introduction to lattices

Let us begin with a few definitions and properties about lattices:

Definition 1.1. *Let $H = \mathbb{R}^m$. A lattice is a discrete subgroup of H. For a basis $B = \{b_1, ..., b_n\} \in H^n$, we note $L(B)$ and call lattice generated by B the set of vectors*

$$\left\{ \sum_{i=1}^{n} x_i b_i | x_i \in \mathbb{Z} \right\}$$

A lattice will be noted Λ or $L(B)$ when the basis B will be provided.

Definition 1.2. *The i-th successive minimum λ_i of a lattice Λ is defined as the minimum radius $r \in \mathbb{R}$ of a n-dimensional sphere B with center 0 that contains i linearly independent lattice vectors:*

$$\lambda_i(\Lambda) = min\{r | \dim(span(\Lambda \cap B_{r,0})) \geq i\}$$

Thus we have λ_1 the norm of the shortest non-zero vector in Λ.

Now let us define a few classical problems over lattices that are used to construct cryptosystems:

Definition 1.3 (SVP - Shortest Vector Problem)**.** *Given a n-dimensional lattice Λ, find a lattice vector v such that $||v|| = \lambda_1(\Lambda)$.*

Definition 1.4 (CVP - Closest Vector Problem)**.** *Given a n-dimensional lattice Λ and a point $c \in H$, find a lattice vector v such that $||c - v|| = dist(c, \lambda) = \min_{z \in \lambda} ||c - z||$.*

Those problems above cannot be solved efficiently on classical computers and there is currently no efficient algorithm on quantum computers neither. The following problem has been introduced by Ajtai in [Ajt96]:

Definition 1.5 ($\text{SIS}_{n,m,q,\beta}$ - Shortest Integer Solution)**.** *Let n and m,q = poly(n) be integers. Given a uniformly random matrix $A \in \mathbb{Z}_q^{n \times m}$, find a non-zero vector v such that $Az = 0 \mod q$ and $||z|| \leq \beta$.*

This problem has been proved to be as hard as classical lattices problem, in particular Ajtai showed in [Ajt96] that one-way functions based on the SIS problem were secure as long as SVP was hard. There exists ring variants of SIS where $\mathbb{Z}$ is replaced by the cyclotomic ring $\mathbb{Z}/(x^n + 1)$ with n a power of two and these have been also proved to be as hard as standard lattice problems.

1.2 Falcon Signature Scheme

Falcon is a lattice-based signature scheme and a candidate in the final round of the NIST Post-Quantum competition. It is based on a framework called GPV from its creators Gentry, Peikert and Vaikuntanathan [GPV08]. At a very high level, the signature scheme consists in mapping the message to a point of the space, and then finding a close vector in the lattice. It is easy to do with a "good" basis but difficult to do with a "bad" basis. The private key is thus the "good" basis while the public key is the "bad" basis. Since the two basis span the same lattice, one needs only the public key to verify that the signature is indeed a vector close to the message.

In order to achieve the best compactness and efficiency, Falcon signature scheme is using a specific class of lattices called NTRU lattices that were introduced by Hoffstein, Pipher and Silverman in [HPS98] and benefit from a ring structure which allows significant improvement in the computations speed. Besides the public key is relatively short because it can be reduced to (and stored as) a single polynomial.

In NTRU cryptosystem, the lattice is a discrete subgroup of a ring $R = \mathbb{Z}[x]/(x^n + 1)$ with n being a power of two. Thus elements of the lattice are polynomials that can be written as vectors. For example let $F \in R$, we have

$$F = \sum_{i=0}^{n-1} = F_i x^i = (F_0, F_1, ..., F_{n-1})$$

The secrets of a Falcon private key consist in four polynomials $f, g, F, G \in R$ verifying the NTRU equation $fG - gF = q \mod (x^n + 1)$ with $q \in \mathbb{N}^*$. The private key itself is the matrix $\begin{pmatrix} f & g \\ F & G \end{pmatrix}$ and the public key is the matrix $\begin{pmatrix} 1 & h \\ 0 & q \end{pmatrix}$ with $h = g \cdot f^{-1} \mod q$. Those two matrices span the same lattice Λ_q, but the private key is considered as a "good" basis because it contains four small polynomials, whereas the public key contains two large polynomials h and q. It is difficult to find two small polynomials f' and g' such that $h = g' \cdot (f')^{-1} \mod q$ and thus to recover the private key from the public key.

In fact, since the public basis and the secret basis needs to be orthogonal for the GPV to work, we have a public basis $A = \begin{pmatrix} 1 & h^* \end{pmatrix}$ and a secret basis $B = \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$. We can check that $B \times A = 0 \mod q$ and that the two matrices are indeed orthogonal. The lattice Λ_q is generated by A while B generates the lattice $\Lambda_q^\perp$, with $\Lambda_q^\perp$ being the lattice othogonal to Λ modulo q. Note that since q is a standard parameter set by Falcon, one needs only to store h as the public key.

To produce signatures, the GPV framework requires a trapdoor sampler. A

trapdoor sampler is an algorithm that takes in input a matrix A, a trapdoor T and a target c and finds a short vector s such that $s^t A = c \mod q$, which is in our case equivalent to finding a vector in $\Lambda_q^\perp$ close to our target. There exists several trapdoor samplers and Falcon documentation cites four of them, created by Klein [Kle00], Peikert [Pei10], Micciancio-Peikert [MP12] and Ducas-Prest [DP16]. The trapdoor sampler proposed by Ducas and Prest was the only one satisfying the three Falcon criteria: fastness of the algorithm, shortness of the output, and adaptability to NTRU lattices.

1.3 Fast Fourier Nearest Plane

In [DP16] Ducas and Prest propose a variation of the Babai's Nearest Plane algorithm. The nearest plane algorithm allows to solve the closest vector problem after the computation of the Gram-Schmidt orthogonalization of the lattice's basis but runs in quadratic time with respect to the dimension of the lattice, which can become very quickly prohibitive. The algorithm proposed in [DP16] runs in $\Theta(d \ log \ d)$, with d being the dimension of the lattice, which is much faster than the original nearest plane algorithm. Please note that in this whole section most of the definitions, properties and algorithms are taken either from [DP16] or [Pre15].

Lemma 1.6. *Let $H = \mathbb{R}^m$ and $B = \{b_1, ..., b_n\} \in H^n$ be a basis. For any $k \in [1, n]$, we note $V_k = Span(B_k)$. There is a unique basis $\tilde{B} = \tilde{b}_1, ..., \tilde{b}_n \in H^n$ verifying any of these equivalent properties:*
1. $\forall k \in [1, n], \tilde{b}_k = b_k - Proj(b_k, V_{k-1})$
2. $\forall k \in [1, n], \tilde{b}_k = b_k - \sum_{j=1}^{k-1} \frac{\langle b_k, \tilde{b}_j \rangle}{\langle \tilde{b}_j, \tilde{b}_j \rangle} \tilde{b}_j$
3. $\forall k \in [1, n], \tilde{b}_k \perp V_{k-1}$ *and* $(b_k - \tilde{b}_k) \in V_{k-1}$

Definition 1.7 (Gram-Schmidt Orthogonalization)**.** *Let $B = \{b_1, ..., b_n\} \in H^n$ be a basis. We call Gram-Schmidt orthogonalization (or GSO) of B and note $\tilde{B}$ the unique set $\{\tilde{b}_1, ..., \tilde{b}_n\} \in H^n$ verifying one of the equivalent properties of Lemma 1.6. When clear from context, we also note $\tilde{b}_i$ the i-th vector of $\tilde{B}$, which is also the orthogonalization of b_i with respect to the previous vectors $b_i, ..., b_{i-1}$.*

Proposition 1.8. *Les $B \in R^{n \times m}$ be a full-rank matrix. B can be uniquely decomposed as*

$$B = L \cdot \tilde{B}$$

where L is unit lower triangular, and the rows of $\tilde{B}$ are pairwise orthogonal.

Definition 1.9. *We say that a matrix $G \in R^{n \times n}$ is full-rank Gram (or FRG) if it is full-rank and there exists $m \geq n$ and $B \in R^{n \times m}$ such that $G = BB^*$.*

Since for a basis B and a matrix G we have the unique decompositions $B = L \cdot \tilde{B}$ and $G = LDL^*$, if G is FRG with $G = BB^*$, then $G = L \cdot (\tilde{B}\tilde{B}^*) \cdot L^*$ is a

LDL^* decomposition and by unicity of this decomposition we get the following equivalence:

$L \cdot \tilde{B}$ is the GSO of B $\iff$ $L \cdot (\tilde{B}\tilde{B}^*) \cdot L^*$ is the LDL^* decomposition of (BB^*)

Since we get that the L in the two decompositions is the same matrix, we can compute the LDL^* decomposition instead of the Gram-Schmidt decomposition to achieve the same result, which is faster.

Algorithm 1 $\text{LDL}^*_R(\text{G})$

$L, D \leftarrow 0^{n \times n}$
for i from 1 to n **do**
  $L_{ii} \leftarrow 1$
  $D_i \leftarrow -G_{ii} \sum_{j<i} L_{ij} L^*_{ij} D_j$
  for j from 1 to $i-1$ **do**
    $L_{ij} \leftarrow \frac{1}{d_j}(G_{ij} - \sum_{k<j} L_{ik} L^*_{jk} D_k)$
  end for
end for
return $((L_{ij}), \text{Diag}(D_i))$

Definition 1.10. *Let $B = \{b_1, ..., b_n\}$ be a real basis. We call fundamental parallelepiped generated by B and note P(B) the set $\sum_{i \leq j \leq n} [-\frac{1}{2}, \frac{1}{2}]\, b_j = [\frac{1}{2}, \frac{1}{2}]^n \cdot B$.*

The Babai's nearest plane algorithm takes a vector $t \in \mathbb{R}^n$, the matrix L such that $B = L \cdot \tilde{B}$ and $\tilde{B}$ is the GSO of B, and returns a vector $z \in \mathbb{Z}^n$ such that the difference between t and z lies in the fundamental parallelepiped spanned by $\tilde{B}$. In other words, we get a vector z such that $zB \in L(B)$ and $(t-z)B \in P(\tilde{B})$.

Algorithm 2 $\text{NearestPlane}_R(B, L, c)$

$t \leftarrow c \cdot B^{-1}$
for j from n to 1 **do**
  $\bar{t}_j \leftarrow t_j + \sum_{i>j} (t_i - z_i) L_{ij}$
  $z_j \leftarrow \lfloor \bar{t}_j \rceil$
end for
return $v \leftarrow z \cdot B$

As said earlier, the matrix L can be obtained with the LDL^* algorithm instead of the GSO one. In [DP16] the authors propose a fast algorithm to compute the LDL^* decomposition. They prove that the matrix L can be factored as a product of $\Theta(log\ d)$ sparse matrices, which can be stored in a binary tree.

First we need to define the coefficient vectors and circulant matrices.

Definition 1.11. *For any $d \in \mathbb{N}^*$, let R_d denote the ring $\mathbb{R}[x]/(x^d - 1)$, also known as circular convolution ring, or simply convolution ring.*

Definition 1.12. *We define the coefficient vector* $c : R_d^m \to \mathbb{R}^{dm}$ *and the circulant matrix* $C : R_d^m \to \mathbb{R}^{dn \times dm}$ *as follows. For any* $a = \sum_{i \in \mathbb{Z}_d} a_i x^i \in R_d$ *where each* $a_i \in \mathbb{R}$*:*
1. The coefficient vector of a is $c(a) = (a_0, ..., a_{d-1})$*.*
2. The circulant matrix of a is

$$C(a) = \begin{bmatrix} a_0 & a_1 & \dots & a_{d-1} \\ a_{d-1} & a_0 & \dots & a_{d-2} \\ \vdots & \vdots & \ddots & \vdots \\ a_1 & a_1 & \dots & a_0 \end{bmatrix} = \begin{bmatrix} c(a) \\ c(xa) \\ \vdots \\ c(x^{d-1}a) \end{bmatrix} \in \mathbb{R}^{d \times d}$$

3. c and C generalize to vectors and matrices in coefficient-wise manner.

Proposition 1.13. *The coefficients vector c and the circulant matrix C verify the following properties:*
1. C is a ring isomorphism onto its image. In particular C(a)C(b) = C(ab).
2. c(a)C(b) = c(ab).
3. $C(a)^* = C(a^*)$*.*

To allow factorization of L in sub-matrices, the authors [DP16] introduce also two linearization operators, denoted $V_{d/d'}$ and $M_{d/d'}$. The goal of those two operators is to break elements of the convolution ring R_d into elements of a smaller ring $R_{d'}$, with d' being a proper divisor of d.

Definition 1.14. *Let* $d \in \mathbb{N}^*$ *be a product of h primes. We note gpd(d) the greatest proper divisor of d. When clear from context, we also note h the number of prime divisors of d (counted with multiplicity),* $d_h = d$ *and for* $i \in [1, h]$, $d_{i-1} = d_i / gpd(d_i)$ *and* $k_i = d_i / d_{i-1}$*, so that* $1 = d_0 | d_1 | ... | d_h = d$ *and* $\prod_{j \leq i} k_j = d_i$*.*

The tower of divisors defined above is unique.

Definition 1.15. *Let d, d'*$\in \mathbb{N}^*$ *such that* $d | d'$*. We define the vectorization* $V_{d/d'} : R_d^{n \times m} \to R_d^{x \times m(d/d')}$ *inductively as follows:*
1. Let k = d/gpd(d). For d' = gpd(d) and a single element $a \in R_d$, $a = \sum_{0 \leq i \leq k_d} x^i a_i(x^k)$ *where* $a_i \in R_{d'}$ *for each i. Then*

$$V_{d/d'}(a) = (a_0, \dots, a_{k-1}) \in R_{d'}^k$$

In other words, $V_{d/d'}(a)$ *is the row vector whose coefficients are the* $(a_i)_{0 \leq i \leq k_d}$*.*
2. For a vector $v \in R_d^m$ *or a matrix* $B \in R_d^{n \times m}$, $V_{d/d'}(v) \in R_d^{(d/d')m}$ *and* $V_{d/d'}(B) \in R_d^{n \times (d/d')m}$ *are the componentwise applications of* $V_{d/d'}$*.*
3. For $d'' | d' | d$ *and any element* $a \in R_d$*,*

$$V_{d/d''}(a) = V_{d'/d''} \circ V_{d/d'}(a) \in R_{d''}^{d/d''}$$

When d is clear from context, we simply note $V_{d/d'} = V_{/d'}$*.*

As the vectorization $V_{d/d'}$ breaks elements of R_d into vectors of a smaller ring, the authors similarly define the matrixification $M_{d/d'}$ that breaks elements into matrices.

Definition 1.16. *Following the notations of Definition 1.15, we define the matrixification* $M_{d/d'} : R_d^{n\times m} \to R_d^{n(d/d')\times m(d/d')}$ *as follows:*
1. Let $k = d/gpd(d)$. *For* $d' = gpd(d)$ *and a single element* $a \in R_d$, $a = \sum_{0\le i\le k_d} x^i a_i(x^k)$ *where* $a_i \in R_{d'}$ *for each* i. *Then*

$$M_{d/d'}(a) = \begin{bmatrix} a_0 & a_1 & \dots & a_{k-1} \\ xa_{k-1} & a_0 & \dots & a_{k-2} \\ \vdots & \vdots & \ddots & \vdots \\ xa_1 & xa_2 & \dots & a_0 \end{bmatrix} = \begin{bmatrix} V_{d/d'}(a) \\ V_{d/d'}(x^k a) \\ \vdots \\ V_{d/d'}(x^{(d'-1)k} a) \end{bmatrix} \in R_{d'}^{nk\times mk}$$

In particular, if d *is prime, the* $M_{d/1}(a) \in \mathbb{R}^{d\times d}$ *is exactly the circulant matrix* $C(a)$.
2. For a vector $v \in R_d^m$ *or a matrix* $B \in R_d^{n\times m}$, $M_{d/d'}(v) \in R_d^{(d/d')\times(d/d')m}$ *and* $M_{d/d'}(B) \in R_d^{(d/d')n\times(d/d')m}$ *are the componentwise applications of* $M_{d/d'}$.
3. For $d''|d'|d$ *and any element* $a \in R_d$,

$$M_{d/d''}(a) = M_{d'/d''} \circ M_{d/d'}(a) \in R_{d''}^{(d/d'')\times(d/d'')}$$

When d *is clear from context, we simply note* $M_{d/d'} = M_{/d'}$.

The vectorization and matrifixification of a element $a \in R_d$ merely permute the rows and the columns of its circulant matrix $C(a)$. If a is a basis, the operation will change the lattice generated but as it preserves the scalar product it will leave the geometry of the lattice unaffected. The Figure 1 (taken from [Pre15]) illustrates the permutations of the rows and columns when the matrixification is applied to an element $a \in R_d$.

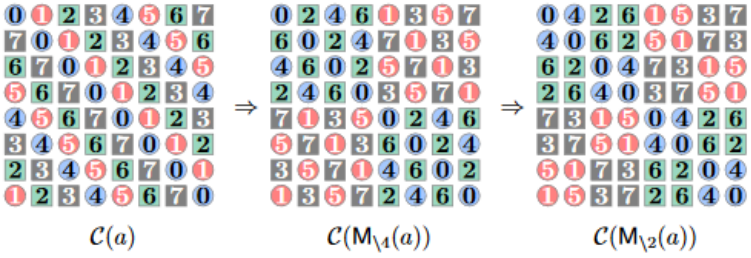

Figure 1: Matrixifications of $a = 0 + x + 2x^2 + \dots + 7x^2$

Now that we have all the definitions, we can state the main result presented in [DP16]:

Theorem 1.17. *Let $d \in \mathbb{N}$ and $1 = d_0|d_1|...|d_h = d$ be a tower of proper divisors of d. Let $b \in R_d^m$ such that $M_{d/1}(b)$ is full-rank. There exists a GSO of $M_{d/1}$ as follows:*

$$M_{d/i}(b) = \left(\prod_{i=0}^{h-1} M_{d_i/1}(L_i) \right) \cdot \tilde{B}_0$$

where $\tilde{B}_0 \in \mathbb{R}^{d \times dm}$ is orthogonal, and each $L_i \in R_{d_i}^{(d/d_i)\times(d/d_i)}$ is a block-diagonal matrix with unit lower triangular matrices of $R_{d_i}^{(d_{i+1}/d_i)\times(d_{i+1}/d_i)}$ as its d/d_{i+1} diagonal blocks.

This theorem allows to represent the L matrix of the GSO and LDL decomposition (which is the same in both cases) as a product of sparse matrices such that every factor L_i as fewer non-zero coefficients. The factorization of L can be stored in a L-tree, as showed in Figure 2 (taken from [Pre15].

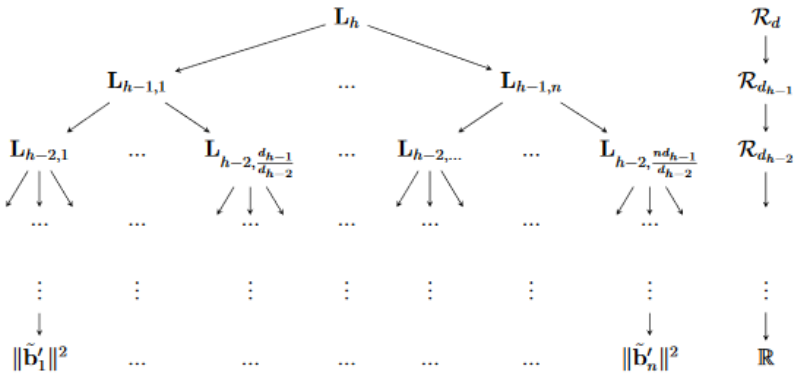

Figure 2: L-tree of matrices L_{ij} with $L = \prod_i M_{d_i/1}(diag_j(L_{ij}))$

The authors also propose an algorithm to compute the LDL decomposition tree from a full-rang Gram matrix $G \in R_d^{n\times n}$ called "Fast Fourier LDL" because all the polynomials are in FFT form.

Algorithm 3 ffLDL$_{R_d}(G)$

```
if d = 1 then
    return (G, [ ])
end if
(L, D) ← LDL_{R_d}(G)
for i from 1 to n do
    T_i ← ffLDL_{R_{gpd(d)}}(M_{d/gpd(d)}(D_ii))
end for
return (L, (T_i)_{1≤i≤n})
```

Finally, the authors use the precomputed L-tree to propose a Fast Fourier variant of the Babai's nearest plane algorithm. The algorithm ffNearestPlane takes in input a target vector $t \in R_d^n$, a precomputed L-tree T and output a vector z with the same properties as the classical Babai's nearest plane but $O(d)$ times faster.

Algorithm 4 ffNearestPlane$_{R_d}(t, T)$

if t is a 1-dimensional vector in $\mathbb{R}$ **then**
  return $\lfloor t \rceil$
end if
L $\leftarrow$ T.Node()
for j from n to 1 **do**
  $\bar{t}_j \leftarrow t_j + \sum_{i>j}(t_i - z_i)L_{ij}$
  $z_j \leftarrow V^{-1}_{d/\text{gpd(d)}} \left[\text{ffNearestPlane}_{R_{\text{gpd(d)}}}(V_{d/\text{gpd(d)}}(\bar{t}_j), \text{T.Child(j)})\right]$
end for
return z

The ffSampling algorithm described in the Falcon documentation and implemented for this project is merely the ffNearestPlane algorithm with different notations, and with the rounding of t replaced by a Gaussian sampling over the integers.

1.4 Why we need trapdoor samplers

Let us consider an older lattice-based signature scheme, the GGH signature scheme [GGH12]. GGH is based on the closest vector problem. Let $H(m)$ be the hash of the message to sign. To produce a signature of this message, GGH works with a lattice Λ spanned over $\mathbb{Z}_n$ and considers $H(m)$ as a point of $\mathbb{Z}_n$. With the Babai's Nearest Plane algorithm, it returns a closest vector s such that $||H(m) - s||$ is minimised.

Phong Q. Nguyen and Oded Regev showed in [NR06] that the secret key can be recovered in polynomial time. Indeed, all the $||H(m) - s||$ lie in the parallelepiped spanned by the secret basis for every couple of (message, signature). They called it the "hidden parallelepiped problem" as shown in Figure 3. They showed that recover the secret basis from the parallelepiped created by those points can be done in polynomial time using gradient descent, but it is out of the scope of this paper so we will not discuss it.

What we need to remember is that we cannot return exactly the closest vector, because with several signatures it becomes really simple to recover the secret basis. That is why we need a "perturbation" which is here a Gaussian sampler. Furthermore we need to choose very carefully the standard deviation σ of the sampler, which cannot be 0 because it would not prevent the previous attack,

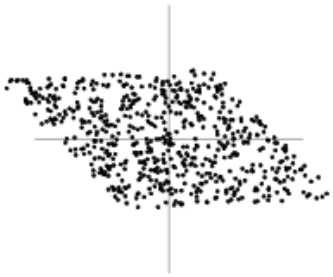

Figure 3: The hidden parallelepiped problem

and cannot be too high neither because then the outputed vector would not be short enough to meet the requirements of the cryptosystem.

In Figure 4 (taken from [Pre15]), we can see the distribution of the distances between the signatures and the corresponding messages for signature schemes like GGH or NTRUSign on the left, and for the GPV framework on the right. We can observe that it appears very intuitive to recover the short basis corresponding to the fundamental parallelepiped in the GGH, while in the GPV framework the parallelepiped (and thus the secret key) remains hidden in a cloud of points with no obvious geometric properties.

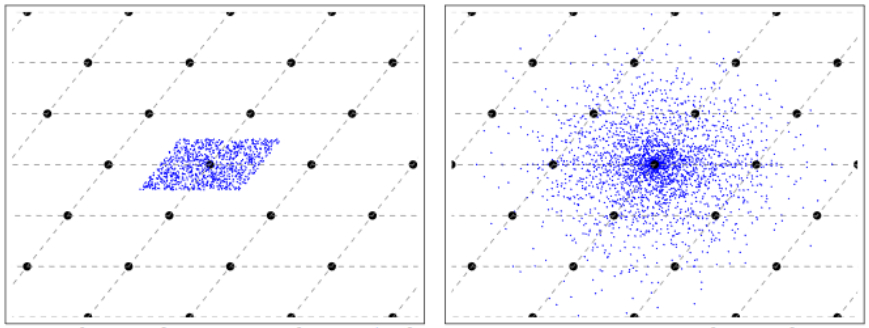

Figure 4: Distribution of the distances between signatures and messages

2 Description of the program architecture

2.1 Sampling over the integers

The `ffSampling` algorithm uses as a subroutine a gaussian sampler over the integers called `SamplerZ` that we needed to implement first. The function `SamplerZ` and its subroutines are in the file `samplerz.c`. Before the first call to this function we need to initialize the reverse cumulative distribution table RCDT stored in `tables.c`. It has to be done in a function because we cannot declare and initialize this array in a static way. Indeed, since the number contained in the RCDT are bigger than $2^{64} - 1$ they cannot be stored in `uint64_t` and thus we need to use the GMP library and the type `mpz_t`.

`SamplerZ` calls to a function `BaseSampler` which returns an integer $z \in \{1, 18\}$ according to the distribution table in RCDT. The integer obtained follows a distribitution very close to the half Gaussian distribution. The function `ApproxExp` returns an integral approximation of $2^{63} \cdot css \cdot exp(-x)$, with $x \in [0, ln(2)]$ and $css \in [0, 1]$. The result is lesser than 2^{64} and can be stored in a `uint64_t`. It is used by the function `BerExp` to return a single bit with probability $css \cdot exp(-x)$. Finally the function `SamplerZ` returns an integer according to a Gaussian distribution of parameters (mu, sigma). The quotient $sigma/sigmin$ at the beginning of the function is here to make the algorithm running in constant time.

Concerning the source of randomness, we use a function `random_bybtes` defined in the file `random.c`. This function has two definitions because its behaviour differs when the program is run in test mode (see section 2.3). In standard mode, this function reads bytes of randomness directly from `/dev/urandom` and store them in a buffer. The function opens and closes the file descriptor at each call but the file descriptor could be saved outside the function and opened only once to save system calls.

2.2 Sampling over lattices

The core of this project is the `ffSampling` algorithm that can be found in the file `ffsampling.c`. As explained in the section 1.3, this algorithm takes in input a target t and a L-tree T which has to be precomputed. So before implementing `ffSampling` we needed to implement all the functions required for the computation of the L-Tree, which will be called in this section and in the following a Falcon tree.

The Falcon tree is stored in a structure `t_tree` defined in the file `ffsampling.h`. It is a very basic binary tree with a value and two children. The value is itself a structure. In the previous section we were working only with integers, but from now we are working with polynomials so we need to define an adapted structure. In this project we have three different structures very similar for the

three types of polynomials we encounter. The three structures have a field `len` of type `int` and a field `coeffs` which is a pointer whose type differs depending of the structure.
- `t_pol`: basic polynomials. The type of its coefficient is `double`.
- `t_pol_fft`: polynomials in FFT representation. The type of its coefficient is `double complex`. We go from the `t_pol` type to the `t_pol_fft` type with the functions `fft` and `ifft` defined in the file `fft.c`.
- `t_pol_ntt`: polynomials in NTT representation. The type of its coefficient is `int`. We go from the `t_pol` type to the `t_pol_ntt` type with the functions `ntt` and `intt` defined in the file `ntt.c`.

Since `int` can be stored in `double` that can also be stored in `complex double`, we could have used a unique structure for all polynomials instead of three. However, it would have taken more space in memory than needed, it would have caused problem of floating-point precision with the integers and it would have made the code much more difficult to read as many operations would have required to distinguish the real part from the imaginary one. For all these reasons it has been chosen to create three different structures. Since the lengths of the polynomials are not fixed, it is necessary to dynamically allocate memory for the array of coefficients (and eventually to free it).

The file `pol_op.c` contains functions for elementary operations between polynomials. The `add_fft`, `sub_fft`, `adj_fft`, `mul_fft`, `div_fft` and `sqrt_fft` functions all takes in inputs operands as `t_pol_fft` and also another `t_pol_fft` to store the result, that needs to have been previously allocated. The functions `add_zq`, `sub_zq`, `mul_zq` and `div_zq` computes operations modulo q and takes in inputs standard polynomials. The polynomials are the transformed into NTT polynomials in order to perform the required operation. A standard polynomial is finally allocated and returned with the result.

The Falcon tree itself is stored in another structure called `t_sk` containing the secret key. The two other fields are the secret basis B which is a 2×2 array of `t_pol_fft` polynomials and a polynomial h which is in fact the public key. The secret key is generated in the function `gen_sk` in the file `keygen.c`. The basis B is computed from the four secret polynomials f, g, F, G. Then the Gram matrix $G = BB^*$ with the function `gram`. The Falcon tree is computed from the Gram matrix with the `ffLDL` function which itself calls the `LDL` function. Please note that since all those functions have been implemented with the very specific goal of computing the Falcon secret key from a 2×2 matrix as specified in the official documentation, they are not generic functions and only works for 2×2 matrices. Finally, the Falcon tree is normalized with respect to the standard deviation defined in Falcon parameters and the public key h is computed. The public key is not used in `ffSampling` but is needed for our pseudo-signature scheme (see section "Experimental results").

Now that we have computed our Falcon tree we can implement the algorithm `ffSampling`. As said earlier it takes in input a `t_tree` T containing a Falcon tree, an vector of two `t_pol_fft` t containing the target, and a structure `t_params` containing the parameters for the Falcon algorithm (more information about the parameters in the section "Experimental results"). This function returns a vector close to the target according to a Gaussian distribution. It is merely the algorithm `ffNearestPlane` described in 1.3 with the round-off part being replaced by a call to our function `SamplerZ`. We still need to generate the target t for testing the algorithm but it can be done randomly (or from a text message as shown later).

2.3 Testing

Unit tests have been provided whenever it was possible. To run the tests, the program should be compiled and run in TEST mode with the following command line: `make TEST=true && ./ffsampling`. If the program was previously compiled in standard mode, it is necessary to clean the object files before compilation with the command `make clean`. The main difference with the standard compilation is that `main.c` is replaced by `test.c`. `KAT.c` is also added to the source files. The file `test.h` is also added to the include files. Finally a define is set, allowing to switch from a truly random function `random_bytes` in `random.c` to a deterministic one, extracting "randomness" from a predefined flow of bytes.

test_samplerZ. This function uses a set of Known Answer Tests (KAT) provided by the Falcon documentation. The data are stored in a `samplerZ_KAT` structure in the `KAT.c` file. The parameters are as follows:
- a center `mu`
- a standard deviation `sigma`
- a minimum deviation `sigmin`
- a seed `randombytes` used for predictive randomness
- an expected output `z`

The function loops over the available KAT and calls `SamplerZ` with the given parameters. If the integer returned by `SamplerZ` differs from the expected output in the KAT then the function raises an exception and exit the program.

test_ftt. No KAT were provided for the implementation of the FFT algorithm since it is a very common algorithm. Since the FFT representation aims to compute a fast multiplication of two polynomials, the test function first applies the FFT transformation to two polynomials `f` and `g`, computes the multiplication of f and g in FFT representation, and then recover g with a division still in FFT representation. The function then applies the inverse FFT transformation and checks that the obtained polynomial is the same as the original `g`. The polynomials `f` and `g` are randomly generated and FFT is tested for degrees 2, 4, 8, 16, 32, 64 and 128.

test_ntt. This function is very similar to the `test_fft` function, excepts that it tests the NTT transformation instead of the FFT transformation. More precisely, it tests the modular multiplication of two polynomials, which itself uses the NTT transformation. Here again two polynomials `f` and `g` are randomly generated for degrees 2, 4, 8, 16, 32, 64 and 128. Then the modular multiplication $f \times g \mod q$ is computed and `g` is recovered by a modular division. The function then tests that the recovered `g` is the same as the original `g`.

3 Experimental results

In order to show an application of the algorithm ffSampling, we have implemented two additional functions: `pseudo_sign` and `pseudo_verify`. Those functions represent a very basic signature scheme over lattices, with no security pretension at all.

The function `pseudo_sign` takes in input a message, a private key and parameters. The message is simply a string that will be converted to a point of the vector space with the function `HashToPoint`. The private key has been generated in the function `gen_key` as explained in the previous section. The parameters are the ones defined in Falcon specification and they depend on the dimension of the lattice. According to Falcon documentation, they are computed as follows:

With n being the degree of the ring $\mathbb{Z}[x]/(x^n + 1)$ in which lies the lattice, Q_S being the maximal number of signing queries (defined by the NIST at 2^{64}), λ being the level of security (also defined by the NIST at either 128 or 256), q being a prime integer modulus set at 12289 to maximize the efficiency of NTT and $\epsilon \leq 1/\sqrt{Q_S \cdot \lambda}$ arbitrary, we have σ the standard deviation for the Gaussian distribution

$$\sigma = \frac{1}{\pi} \cdot \sqrt{\frac{log(4n(1 + 1/e))}{2}} \cdot 1.17 \cdot \sqrt{q}$$

We also have the maximal norm β used to accept or reject a signature (s_1, s_2) with the condition $||(s_1, s_2)||^2 \leq \lfloor \beta^2 \rfloor$:

$$\beta = 1.1 \cdot \sigma\sqrt{2n}$$

If the signature obtained in `pseud_sign` is larger than the given bound, the signature procedure is restarted. According to [Lyu12] (quoted in the Falcon documentation), this event happens with the following probability:

$$\mathbb{P}[||(s_1, s_2)||^2 \leq \lfloor \beta^2 \rfloor] \leq 1.1^{2n} \cdot e^{n(1-1.1^2)}$$

The values of the different parameters have been taken from the official Falcon implementation in python (https://github.com/tprest/falcon.py).

With those inputs, the function `pseudo_sign` first transforms the message into a polynomial in FFT representation and then uses the basis contained in the secret key to compute a preimage of the point. This preimage is not necessarily a short vector but is the starting point for the ffSampling algorithm, which uses the Falcon tree as auxiliary information to return us a random point on our lattice close to our message. More precisely, as described in 1.3, ffSampling gives us a vector $z \in \mathbb{Z}^n$ such that the difference between z and our message lies in the fundamental parallelepiped spanned by our basis. However, as stated above, there is a small chance that this distance is not short enough due to the use of the Gaussian sampler. Thus we check the norm and restart the signature procedure if needed, and we return the s_2 polynomial otherwise.

The function `pseudo_verify` is used to verify the signature generated by the function `pseudo_sign`. It takes in input the message as a string, a signature s_2 which is a point lying in our lattice and represented in the code as a polynomial in coefficients representation, a public key h which is a polynomial computed during the private key generation process in `gen_key`, and the Falcon parameters as described above. The function calls to `HashToPoint` to obtain the point p on the vector space corresponding to the message. The next step consists in recovering s_1 by computing $s_1 = p - s_2 \times h \mod q$. Then we normalize the coefficients around $q/2$, we compute the $||(s_1, s_2)||^2$ and we check it against $\lfloor \beta^2 \rfloor$. The function returns 1 upon success and 0 upon failure.

Those two functions are called in the `main` if the program is compiled in a standard way with the command `make` (as opposed to the TEST mode described in the 2.3 section). If the program was previously compiled in TEST mode then it is necessary to clean the object files before the new compilation with the command `make clean`. The program shall be launched in command line with exactly three arguments: an integer representing the degree n of the ring $\mathbb{Z}[x]/(x^n + 1)$, an option which is either `-s` for signing or `-v` for verifying, and the message to be signed or verified. If the program is executed in signing mode, the function `pseudo_sign` is called and the signature is stored in the file `message.sig`. If the program is executed in verifying mode, the signature is recovered from the file `message.sig` and the function `pseudo_verify` is called. If the program is executed with too few arguments then the usage is displayed:

```
Usage:  ./ffsampling [dim] [-s|-v] [message]
```

Currently the program only supports the following values for the dimension of the lattice: 2, 4, 8, 16, 32, 64 and 128. However it is not difficult to add the dimensions up to 1024 as it only requires to add the corresponding parameters and to provide suitable polynomials in the file `params.c` to generate the secret key. The NTRU polynomials can also be generated with the function described in Falcon documentation but not implemented in this project (they are currently taken from the test suite of the python implementation).

Please note that the function `HashToPoint` used to transform an arbitrary string into a point of the vector space is absolutely not secure and shall not be used for real cryptography. This function is inspired by the `djb2` algorithm which is really easy to implement and wildly used for database indexation but not for cryptographic purposes. Official Falcon implementation recommend the use of `SHAKE-256` but it would have been too heavy to implement for the scope of this project. However it is entirely possible to precompute the hash of the data to be signed before using our pseudo-signature scheme and thus eliminating the problem of the unsafe hash function.

Although a lot a memory is dynamically allocated as explained in section 2.2 due to the structure used for the polynomials, all the memory is properly freed progressively and before the end of the program as it can be seen with a memory checker such as valgrind:

```
➜  ffSampling git:(main) ✗ valgrind --leak-check=full --show-leak-kinds=all ./ffsampling 64 -s hello
==18087== Memcheck, a memory error detector
==18087== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==18087== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==18087== Command: ./ffsampling 64 -s hello
==18087==
message: hello
signature written to message.sig
==18087==
==18087== HEAP SUMMARY:
==18087==     in use at exit: 0 bytes in 0 blocks
==18087==   total heap usage: 3,842 allocs, 3,842 frees, 237,952 bytes allocated
==18087==
==18087== All heap blocks were freed -- no leaks are possible
==18087==
==18087== For counts of detected and suppressed errors, rerun with: -v
==18087== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

References

[Ajt96] Miklós Ajtai. "Generating hard instances of lattice problems". In: *ACM STOC* 28 (1996), pp. 99–108.

[DP16] Léo Ducas and Thomas Prest. "Fast fourier orthogonalization". In: *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016* (2016), pp. 191–198.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions". In: *ACM STOC* 40 (2008), pp. 197–206.

[GGH12] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. "Public-key cryptosystems from lattice reduction problems". In: *CRYPTO'97* 1294 (2012), pp. 112–131.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. "NTRU: A ring-based public key cryptosystem". In: *Algorithmic Number Theory, Third International Symposium, ANTS-III,Portland, Oregon, USA, June 21-25, 1998, Proceedings* 1423 (1998), pp. 267–288.

[Kle00] Philip N. Klein. "Finding the closest lattice vector when it's unusually close". In: *11th SODA* (200), pp. 937–941.

[Lyu12] Vadim Lyubashevsky. "Lattice signatures without trapdoors". In: *EUROCRYPT 2012* 7237 (2012), pp. 738–755.

[MP12] Daniele Micciancio and Chris Peike. "Trapdoors for lattices: Simpler, tighter, faster, smaller". In: *EUROCRYPT 2012* 7237 (2012), pp. 700–718.

[NR06] Phong Q. Nguyen and Oded Regev. "Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures". In: *EUROCRYPT 2006* 4004 (2006), pp. 271–288.

[Pei10] Chris Peikert. "An efficient and parallel Gaussian sampler for lattices". In: *CRYPTO 2010* 6223 (2010), pp. 80–97.

[Pre15] Thomas Prest. "Gaussian Sampling in Lattice-Based Cryptography". PhD thesis. École Normale Supérieure, 2015. URL: `https://tel.archives-ouvertes.fr/tel-01245066v2`.