University of Technology Darmstadt
Department of Computer Science
Cryptography and Computeralgebra

Diploma Thesis

March 2006

Current Attacks on NTRU

Richard Lindner

University of Technology Darmstadt
Department of Mathematics

Supervised by Prof. Dr. Johannes Buchmann,
Dr. Christoph Ludwig,
Dr. Ulrich Vollmer

Acknowledgements

I should thank many people who have helped me on the way of writing and then righting what you read now. I start with my family, whose love and support have been the very foundation of all my studies. I thank Professor Buchmann, who interested me in the field of lattice cryptography and arranged this thesis. I thank my first supervisor Christoph Ludwig, for many talks about lattice-reduction and programming even from afar. And I especially thank my most thorough supervisor and guide through the worlds of computer science and mathematics, Ulrich Vollmer. As a nice counterpoint, I thank Stefanie Stuckenholz my sweet love, for ensuring that I never spent too much time working. On the same note, I thank my longest and bestest friend Frank Karinda, for giving me the ambition to try for a doctoral thesis next. I thank Rafaël Dahmen for every single tomato-wheat-talk. And last but in no way least, I thank all my other proof-readers Ben, Burgi, and Nicole, who in their different fields of expertise are the most sympathetic geniuses I know. And I thank Ben once more for countless hours of help during the climax of this work. May none of you ever lack for a beverage.

Warranty

I hereby warrant that the content of this thesis is the direct result of my own work and that any use made in it of published or unpublished material is fully and correctly referenced.

Date: ..... Signature: .....

Contents

Introduction

A lot of the worlds most valuable information travels through the Internet today. Be it via emails, chat or video conferences, online banking, eBay, Amazon, or through other channels.

The way in which we protect this data is by usage of public key cryptography systems. For now, the systems of choice are RSA, Diffie Hellman, DSA, and MQF. However, in the future quantum computers may be able to break all these systems in polynomial time, making them insecure. The closely related digital signature schemes face the same problem.

Therefore, new public-key systems and signature schemes are under extreme scrutiny, because the most secure one of them may one day replace the systems we use today. An overview of the digital signature schemes which are post-quantum secure is given in [7]. In this work, we have analysed one particularly promising, quantum-age secure digital signature scheme — NTRUSIGN.

NTRUSIGN is lattice based, so in order to understand how it works, we will need a basic introduction to lattices. The most prominent attack on NTRUSIGN is employing lattice reduction algorithms, so we introduce and explain both these fields in section 1.

As a follow up, in section 2 we will introduce the public-key cryptosystem NTRUENCRYPT, which was invented long before NTRUSIGN. Then we explain NTRUSIGN itself, with a special emphasis on the way the private lattice basis is found. We present a new and efficient way to do this, differing from the one NTRU suggests in [20].

Having explained NTRUSIGN in detail, we go on to analyse all major attacks that have been made on it. For two of these four attacks, we uncover small fallacies that NTRU made and we propose solutions. For a third, we derive a new algorithm to calculate the probability for an attack to succeed. All this is done in section 3.

Using all these newly gained insights, we can illustrate an improved version of the parameter-generation algorithm from [21] in section 4.

And finally, in section 5 we present the surprising results of our tests using lattice reduction techniques. We imitated NTRU's own analysis from [23] for the new lattices they suggested in [20]. We found that 10% of the NTRUSIGN instances generated in good faith with the algorithm proposed by NTRU in [21] (seen in section 4) have a far lower bit-security than we are led to believe.

1 Lattices

Lattices are discrete subgroups of $\mathbb{R}^n$. They were first studied in the 19th century by number theorists. In 1981 Arjen K. Lenstra, Hendrik W. Lenstra Jr., and László Lovász developed their famous lattice basis reduction algorithm. This algorithm soon became a basic tool in discrete linear optimisation as well as cryptography. It was used to break many cryptographic schemes and well known systems in special cases [11, 1, 30]. On the other hand Miklós Ajtai proved in 1997 the NP-hardness of certain problems in a lattice [2], which was then used in security proofs for many cryptographic systems [3, 16, 22]. A good overview of the current uses of lattices in cryptography is given by Nguyen and Stern in [43, 44].

In this section we will start by defining lattices and see the problems that are hard to solve in a given lattice. Using these problems we will derive the NTRUSIGN scheme in section 2.4.

Definition 1.1 (Point Lattice)**.** A *point lattice* is a discrete additive subgroup of $\mathbb{R}^n$. A linearly independent generating system $B = [\mathbf{b}_1, \ldots, \mathbf{b}_m] \in \mathbb{R}^{n \times m}$ of a point lattice L with $\mathbf{b}_i \in \mathbb{R}^n$ for $i = 1, \ldots, m$ is called a *basis* of L. The number of basis-vectors is called the *rank* or *dimension* of L.

We will use the shorter term *lattice* for a point lattices throughout this work. Be aware that this is not to be confused with the same term applied to ordered sets treated in lattice theory.

Every lattice has a basis and the rank of a lattice is independent of the basis. For a lattice L with the basis $B = [\mathbf{b}_1, \ldots, \mathbf{b}_m]$ every point can be written as an integral linear combination of basis vectors

$$L = \left\{ \sum_{i=1}^{m} x_i \mathbf{b}_i \mid x_1, \ldots, x_m \in \mathbb{Z} \right\}$$

Lemma 1.2. *Two bases $B, B' \in \mathbb{R}^{n \times m}$ span the same lattice if and only if there is an unimodular matrix transforming one into the other. Meaning there exists a matrix $U \in \mathbb{Z}^{m \times m}$ with $\det(U) = \pm 1$ such that $B' = BU$.*

Definition 1.3 (Determinant)**.** The *determinant* of a lattice L with basis B is defined as $\det(L) := \sqrt{|\det(B^T B)|}$.

The determinant of a lattice is independent of the basis by lemma 1.2.

1.1 Convolution Modular Lattices

Convolution modular lattices (CML) are a special class of lattices. Their big advantage is that a CML of dimension $2N$ can be characterised by a single

vector of dimension N and an integer q, referring to the modulus which is being used. This makes them especially interesting in cryptography because instead of transferring the whole basis ($2Nn$ numbers), we just transfer the characterising vector and the modulus ($N+1$ numbers). In order to understand them we start with the convolution product for vectors.

Definition 1.4 (Convolution product)**.** For 2 vectors $\mathbf{u} = (u_0, \ldots, u_{N-1})^T$ and $\mathbf{v} = (v_0, \ldots, v_{N-1})^T$ in $\mathbb{Z}^N$, we define their *convolution* to be

$$\mathbf{u} * \mathbf{v} := \mathbf{w} \quad \text{where} \quad w_k := \sum_{i+j=k \mod N} u_i v_j$$

The convolution product makes the lattice $\mathbb{Z}^N$ into a ring $(\mathbb{Z}^N, +, *)$. This ring is isomorphic to the ring of integer polynomials with degree less than N, where all polynomials are reduced modulo $(X^N - 1)$.

$$(\mathbb{Z}^N, +, *) \cong (\mathbb{Z}[X] \;/\; (X^N - 1), +, \cdot)$$

This means that we can calculate the convolution product easily using polynomials instead of vectors.

Definition 1.5 (Convolution modular lattice)**.** Let $\mathbf{c} \in \mathbb{Z}^N$ be a fixed vector and $q > 1$ an integer, then the *convolution modular lattice* associated with $\mathbf{c}$ and q is defined by

$$L(\mathbf{c}, q) := \left\{ (\mathbf{u}, \mathbf{v})^T \in \mathbb{Z}^{2N} \;\middle|\; \mathbf{u} * \mathbf{c} \equiv \mathbf{v} \mod q \right\}.$$

We can see that $L(\mathbf{c}, q)$ is a lattice, since it is an additive subgroup and hence a sublattice of $\mathbb{Z}^{2N}$. We can also see that $L(\mathbf{c}, q)$ has dimension $2N$, because it lies between two other lattices of this dimension.

$$q\mathbb{Z}^{2N} \;\subseteq\; L(\mathbf{c}, q) \;\subseteq\; \mathbb{Z}^{2N}$$

We can also explicitly write out the basis of $L(\mathbf{c}, q)$. For any vector $\mathbf{c} = (c_0, \ldots, c_{N-1})^T$, a *rotation* ρ of $\mathbf{c}$ is a cyclic shifting of the indices

$$\rho(\mathbf{c}) := (c_{N-1}, c_0, \ldots, c_{N-2})^T$$

The *circulant matrix* of $\mathbf{c}$ is the one containing all the rotations of $\mathbf{c}$ as columns

$$\mathcal{C}(\mathbf{c}) = \begin{pmatrix} \mathbf{c} & \rho(\mathbf{c}) & \cdots & \rho^{N-1}(\mathbf{c}) \end{pmatrix} = \begin{pmatrix} c_0 & c_{N-1} & \cdots & c_1 \\ c_1 & c_0 & \cdots & c_2 \\ \vdots & \vdots & \ddots & \vdots \\ c_{N-1} & c_{N-2} & \cdots & c_0 \end{pmatrix}$$

and the explicit basis of $L(\mathbf{c}, q)$ can be written

$$\begin{pmatrix} I_N & 0 \\ \mathcal{C}(\mathbf{c}) & q\, I_N \end{pmatrix},$$

where I_N is the unit matrix.

It is important to note that all four $N \times N$ submatrices $I_N, 0, \mathcal{C}(\mathbf{c})$ and $q\, I_N$ are closed under rotations. This means that for any vector $(\mathbf{u}, \mathbf{v})^T$ in the lattice, each k-rotation $(\rho^k(\mathbf{u}), \rho^k(\mathbf{v}))^T$ is still in the lattice. For this reason, CMLs are sometimes called *circulant modular lattices*. This basis also tells us that the determinant of a convolution modular lattice $L(\mathbf{c}, q)$ is always

$$\det(L(\mathbf{c}, q)) = q^N.$$

In cryptographic applications, we usually need to construct a lattice that contains an especially short vector in some norm. In the case of CMLs, this can be realised for each vector $(\mathbf{a}, \mathbf{b})^T \in \mathbb{Z}^{2N}$ where $\mathbf{a}$ has a convolution inverse $\mathbf{a}_q^{-1}$ modulo q, i.e. $\mathbf{a}_q^{-1} * \mathbf{a} \equiv 1 \mod q$. The lattice is then $L(\mathbf{a}_q^{-1} * \mathbf{b} \mod q, q)$.

This condition is of course equivalent to the polynomial associated with $\mathbf{a}$ being invertible in $(\mathbb{Z}/q\mathbb{Z})[X]/(X^N-1)$. Let us consider N to be prime. If we pick a random binary polynomial $\mathbf{a} \in \{0,1\}[X]/(X^N-1)$, the $\gcd(\mathbf{a}, X^N-1)$ will be 1 when $\mathbf{a}$ is not congruent $(X-1)$ or $(1+X+\cdots+X^{N-1})$ modulo X^N-1. Since all coefficients of $\mathbf{a}$ are invertible in $\mathbb{Z}/q\mathbb{Z}$, $\mathbf{a}$ will often be invertible in $(\mathbb{Z}/q\mathbb{Z})[X]/(X^N-1)$.

In summation, when $\mathbf{a}$ is invertible modulo q we can form $\mathbf{c} = \mathbf{a}_q^{-1} * \mathbf{b} \mod q$ and the lattice $L(\mathbf{c}, q)$ will then contain $(\mathbf{a}, \mathbf{b})^T$ by definition.

Should this short vector $(\mathbf{a}, \mathbf{b})^T$ be unbalanced, i.e. one component has a much bigger norm than the other one, we may insert a balancing constant λ into the basis of $L(\mathbf{c}, q)$ in the following fashion:

$$L_{\mathsf{lb}} = \begin{pmatrix} \lambda\, I_N & 0 \\ \mathcal{C}(\mathbf{c}) & q\, I_N \end{pmatrix} \qquad \det(L_{\mathsf{lb}}) = (\lambda q)^N.$$

Now, L_{lb} is a different CML, in which $(\lambda\mathbf{a}, \mathbf{b})^T$ is a short vector. We use $\lambda = \| \mathbf{b} \| / \| \mathbf{a} \|$ to adapt the norm of the left coordinate to the norm of the right one, thus balancing this vector. For $\lambda \neq 0$, this differently balanced lattice L_{lb} has the same dimension as $L(\mathbf{c}, q)$, but it has a different determinant, as we have seen.

We may also balance the shortest vector the other way round, putting a factor before the right coordinate in order to adapt its norm to the left. To get a lattice containing $(\mathbf{a}, \lambda\mathbf{b})^T$ we use:

$$L_{\mathsf{rb}} = \begin{pmatrix} I_N & 0 \\ \lambda\, \mathcal{C}(\mathbf{c}) & \lambda q\, I_N \end{pmatrix} \qquad \det(L_{\mathsf{rb}}) = (\lambda q)^N.$$

1.2 Hard Problems in Lattices

Two typical problems you might want to solve in a lattice are the *shortest vector problem* (SVP) and the *closest vector problem* (CVP). In a given lattice L, SVP is the problem of finding the shortest non-zero vector in L, and CVP is the problem of finding the vector in L which is closest to a given target vector $\mathbf{t}$, which is somewhere in the vector space containing L. These two problems are so closely related that being able to solve one of them effectively would suffice [17].

In 1981, Emde Boas showed that both SVP and CVP with the max-norm are NP-hard [12]. And with the Euclidean norm, SVP is at least NP-hard for probabilistic transformations due to Ajtai [2]. In practice however, there is the LLL-algorithm [39], which was already good at approximating SVP when it was invented in 1981. Since then, LLL has been improved by many, most notably Schnorr [46, 47, 49, 48].

Approximating the correct solution to SVP and CVP gives rise to further interesting lattice problems. These approximation versions are called $\mathcal{N}$-APPR-SVP, where we try to find a vector $\mathbf{v} \neq \mathbf{0}$ in a lattice L such that $\| \mathbf{v} \| < \mathcal{N}$. And analogously $\mathcal{N}$-APPR-CVP, where we try to find $\mathbf{v} \in L$ such that $\| \mathbf{t} - \mathbf{v} \| < \mathcal{N}$ for a given target vector $\mathbf{t}$. Signature forgery in NTRUSIGN is for example an APPR-CVP. And key-recovery in NTRUENCRYPT or NTRUSIGN is an example of APPR-SVP.

We will use the following standard notation for the length of the shortest non-zero vector in a lattice

$$\lambda_1(L) := \min \left\{ \| \mathbf{v} \| \;\middle|\; \mathbf{v} \in L,\ \mathbf{v} \neq \mathbf{0} \right\}$$

and similarly for the smallest distance of a lattice vector to a given target vector $\mathbf{t}$

$$\lambda_1(\mathbf{t}, L) := \min \left\{ \| \mathbf{t} - \mathbf{v} \| \;\middle|\; \mathbf{v} \in L \right\}.$$

Also note that for the notion of closeness used in these problems, we do not need the positive definiteness of a norm, so a semi-norm suffices.

1.3 Solving SVP and CVP

We start with a survey of exact algorithms which really do find the *shortest* vector in a given lattice, but run in exponential-time of the lattice dimension. We then continue by looking at the more practical, polynomial-time algorithms, which only *approximate* a solution. Throughout this section, L will be a general lattice of dimension n in which SVP and CVP need to be solved.

1.3.1 Exact Algorithms

Rectangular Cube Search. Ravi Kannan showed in 1983 that an exact solution to CVP can be obtained in time $2^{\mathcal{O}(n \log(n))}$ [33, 34]. His algorithm finds the closest vector in L to a given point x by recursively searching inside an n-dimensional rectangular parallelepiped (cube), which is centred at x, and has its edges along the Gram-Schmidt vectors of a given basis of L (these Gram-Schmidt vectors are explained in 1.3.2).

So Kannan's solution to CVP belongs to the category of recursive cube search (RCS) algorithms. He proves the given super-exponential time bound using Minkowski's convex body theorem. Kannan's RCS remains, in theory, the best exact CVP algorithm (see also [19] for an improved constant).

Sieve. For the exact SVP, Kannan's algorithm was beaten in 2001 by the randomised and merely exponential Sieve algorithm. This algorithm with running time $2^{\mathcal{O}(n)}$ was discovered by Ajtai *et al* [4]. It works in the following way.

Given a basis of the lattice L, it randomly chooses $2^{\mathcal{O}(n)}$ lattice points $z_1, \ldots, z_N$, all inside a sufficiently large parallelepiped. Then these lattice points are perturbed by small quantities to get vectors $x_1, \ldots, x_N$, but the original z_i are also kept. With each z_i and x_i an approximation a_i is introduced, which starts at 0.

The algorithm iterates, by putting all $(x_i - a_i)$ into the actual sieve, which is a sub-algorithm that searches a small subset of "representatives" and a large subset of "survivors". For a given bound R on the length of all input vectors, the sieve guarantees that all survivors are in an $R/2$ ball around a known representative. For an illustration see figure (1).

Note that this implies, for each survivor $(x_i - a_i)$ and its representative $(x_j - a_j)$, that their distance to each other is about half the distance between x_i and its approximation. Using this knowledge, each approximation a_i is improved, by adding to it the real lattice point $(z_j - a_j)$ corresponding to its representative $(x_j - a_j)$. This yields smaller "difference" lattice points $w_i = z_i - a_i$ with each iteration.

In fact, if $w_i = 0$, then (z_i, x_i, a_i) can be discarded, since the algorithm searches for small non-zero lattice points only. The perturbation guarantees that this will not happen too often. When all w_i are sufficiently small, the algorithm stops and the authors show that with a probability exponentially close to 1, the smallest vector is one of these w_i.

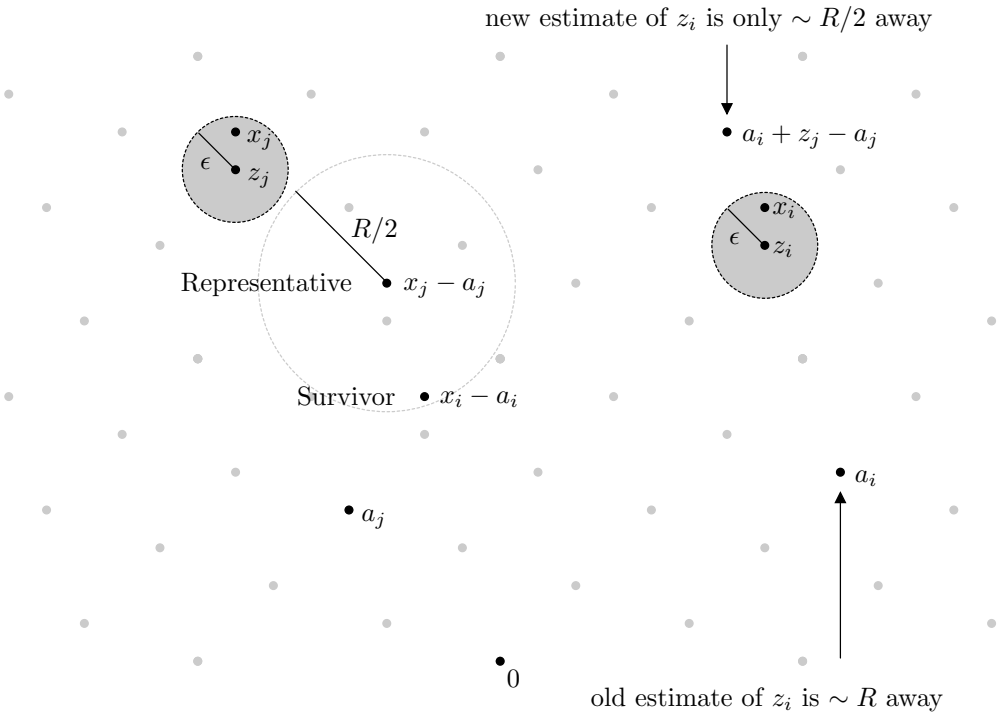

Figure 1: Iterative part of the sieve algorithm.

1.3.2 Lattice Reduction Algorithms

The exact algorithms we have seen up to this point are of theoretical importance only, they are not used in practice, unlike the algorithms we will see now.

The goal of lattice reduction is to find a lattice basis of *small* vectors. Such a basis can then easily be used to approximate both SVP and CVP. So the process of lattice reduction is that we start with a lattice basis which is not special in any way, and we end up with a new basis for the same lattice, which is made up of many short vectors.

Gram-Schmidt. The first step that is used in lattice reduction is the Gram-Schmidt algorithm which decomposes a lattice basis.

Definition 1.6 (Gram-Schmidt)**.** Let $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ be a basis of the lattice L. The decomposition $B = \hat{B}R$ is called *Gram-Schmidt decomposition* if $\hat{B} = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n]$ has pairwise orthogonal columns and $R = (r_{i,j}) \in \mathbb{R}^{n \times n}$ is a unit upper triangular matrix, i.e. $r_{i,j} = 0$ for $i < j$ and $r_{i,j} = 1$ for $i = j$.

Note that if $B \in \mathbb{R}^{d \times n}$ and $d \geq n$ and B has full rank n, then this decomposition is *unique*. This will be the case for all further work, because we will decompose a basis, which has full rank by definition. The Gram-Schmidt decomposition is a special case of the QR-decomposition, which is similar but R is required to be merely an upper triangular, and not a *unit* upper triangular matrix.

Proposition 1.7. *The Gram-Schmidt decomposition with the above notation can be efficiently computed using the following recursion.*

$$\hat{\boldsymbol{b}}_1 = \boldsymbol{b}_1 \qquad \hat{\boldsymbol{b}}_j = \boldsymbol{b}_j - \sum_{i=1}^{j-1} r_{i,j}\hat{\boldsymbol{b}}_i \qquad \text{for } 1 < j \leq n$$

$$r_{i,i} = 1 \qquad r_{i,j} = \frac{\langle \hat{\boldsymbol{b}}_i, \boldsymbol{b}_j \rangle}{\left\| \hat{\boldsymbol{b}}_i \right\|^2} \qquad \text{for } 1 \leq i < j \leq n$$

For any Gram-Schmidt decomposition the following hold

$$\det(L) = \prod_{j=1}^{n} \left\| \hat{\boldsymbol{b}}_j \right\| \quad \text{and} \quad \lambda_1(L) \geq \min\{ \left\| \hat{\boldsymbol{b}}_1 \right\|, \ldots, \left\| \hat{\boldsymbol{b}}_n \right\| \} \tag{1}$$

We see that Gram-Schmidt decomposes the original basis into two matrices that are easier to work with, but it does not start with the actual shortening of the basis vectors and just gives us a lower bound for $\lambda_1(L)$ (see (1)). In practice, the Gram-Schmidt decomposition is usually done with different algorithms based on Householder transformations [24] or fast Givens rotations [18], because these leave less room for numerical errors.

LLL. The real work starts when we use the decomposed basis and perform the famous LLL algorithm by Lenstra, Lenstra, and Lovász [39] to accomplish a δ-LLL-reduction.

Definition 1.8 (δ-LLL)**.** Let $\delta \in]\frac{1}{4}, 1]$, and let $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ be a lattice basis with Gram-Schmidt decomposition $B = \hat{B}R$. Then B is called δ-LLL-reduced if and only if these conditions hold

$$|r_{i,j}| \leq \tfrac{1}{2} \qquad \text{for } 1 \leq i < j \leq n \text{ and}$$

$$\left\| \hat{\mathbf{b}}_{j+1} \right\|^2 \geq (\delta - r_{j,j+1}^2) \cdot \left\| \hat{\mathbf{b}}_j \right\|^2 \qquad \text{for } 1 \leq j < n$$

Proposition 1.9. *In a δ-LLL-reduced basis $B = \hat{B}R$ the following hold for $\alpha := (\delta - \frac{1}{4})^{-1}$ and $1 \leq i \leq j \leq n$.*

$$\| \boldsymbol{b}_1 \| \leq \alpha^{\frac{n-1}{4}} \det(L)^{\frac{1}{n}} \qquad \prod_{i=1}^{n} \| \boldsymbol{b}_i \| \leq \alpha^{\frac{n(n-1)}{4}} \det(L)$$

$$\| \boldsymbol{b}_i \| \leq \alpha^{\frac{j-1}{2}} \left\| \hat{\boldsymbol{b}}_j \right\| \qquad \alpha^{\frac{1-i}{2}} \lambda_i(L) \leq \| \boldsymbol{b}_i \| \leq \alpha^{\frac{n-1}{2}} \lambda_i(L) \tag{2}$$

An LLL-reduced basis contains an approximate solution for SVP (see (2)). The LLL-algorithm computes these reductions in $\mathcal{O}(dn^4)$ arithmetic steps. The original algorithm by Lenstra, Lenstra and Lovász worked on rational numbers having bit length $\mathcal{O}(n^2)$, and improvements of LLL due to Schnorr and Euchner [49] operate on floating point numbers of bit length $\mathcal{O}(n)$, which increases performance drastically.

KZ. Next to LLL there is a different approach to lattice basis reduction namely the Korkine-Zolotarev reduction. Korkine-Zolotarev reduction is defined using smaller-dimensional, orthogonal projections of the original lattice L.

Definition 1.10 (Orthogonal Projection)**.** Let $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ be a basis of a lattice L with a Gram-Schmidt decomposition $\hat{B}R$, where $\hat{B} = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n]$. Then we define

$$P_j := [\tfrac{\hat{\mathbf{b}}_1}{\| \hat{\mathbf{b}}_1 \|}, \ldots, \tfrac{\hat{\mathbf{b}}_{j-1}}{\| \hat{\mathbf{b}}_{j-1} \|}] \qquad \text{for all } 1 \leq j \leq n$$

We use P_j to define the projection π_j of L onto $W_j := \{\mathbf{b}_1, \ldots, \mathbf{b}_{j-1}\}^{\perp}$ by

$$\pi_j : \mathbb{R}^d \to \mathbb{R}^d : \mathbf{v} \mapsto P_j P_j^T \mathbf{v}$$

The image of L in W_j is denoted $L_j(B) := \pi_j(L)$.

Note that the projection π_j depends on the basis B used to describe L, also $\pi_1 = id_{\mathbb{R}^d}$.

Definition 1.11 (Korkine-Zolotarev)**.** Let $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ be a lattice basis with Gram-Schmidt decomposition $B = \hat{B}R$. Then B is called Korkine-Zolotarev-reduced if and only if these conditions are satisfied

$$|r_{i,j}| \leq \tfrac{1}{2} \qquad \text{for } 1 \leq i < j \leq n \text{ and}$$

$$\left\| \hat{\mathbf{b}}_j \right\| = \lambda_1(L_j(B)) \qquad \text{for } 1 \leq j < n.$$

Clearly the first vector of a Korkine-Zolotarev-reduced basis is a solution for the SVP in the lattice spanned by B. However, the other basis vectors are also small by the following proposition.

Proposition 1.12. *Let $B = [\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n] \in \mathbb{R}^{d \times n}$ be a Korkine-Zolotarev reduced basis, then*

$$\tfrac{4}{3+j}\lambda_j(L)^2 \leq \| \boldsymbol{b}_j \|^2 \leq \tfrac{3+j}{4}\lambda_j(L)^2 \qquad \text{for } 1 \leq j \leq n.$$

Several algorithms can compute Korkine-Zolotarev reductions [33, 34, 38]. However they all have an exponential runtime in the lattice dimension n, whereas LLL has polynomial runtime in n.

BKZ. Seeing the differences in runtime and reduction payoff between LLL and KZ, Schnorr proposed a parameterised form of reducedness, with LLL at one and KZ at the other end [46, 47]. This notion of reduction is called β-Block Korkine-Zolotarev or β-BKZ reduction.

Definition 1.13 (β-BKZ)**.** Let $\beta \in \{2, \ldots, n\}$, and let $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ be a lattice basis. Then B is called β-Block Korkine-Zolotarev reduced if and only if these conditions hold

$$B_j := [\pi_j(\mathbf{b}_j), \ldots, \pi_j(\mathbf{b}_{\min\{j+\beta-1,n\}})] \quad \text{is KZ-reduced} \quad \text{for } 1 \leq j \leq n.$$

After Schnorr developed this notion of reducedness he continued to analyse the approximation factor guaranteed by β-BKZ. This would help in order to show the payoff in reduction for higher-polynomial or even exponential runtime.

Definition 1.14 (Hermite constant)**.** The *Hermite constant* γ_n of dimension n is defined as

$$\gamma_n := \sup \left\{ \tfrac{\lambda_1(L)^2}{\det(L)^{2/n}} \mid L \text{ is a } n\text{-dimensional lattice} \right\}$$

Proposition 1.15. *Let $B = [\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n] \in \mathbb{R}^{d \times n}$ be a β-BKZ reduced basis. Then B is also 1-LLL reduced and for $1 \leq j \leq n$:*

$$\| \boldsymbol{b}_j \|^2 \;\leq\; (\gamma_\beta)^{\frac{2(n-1)}{\beta-1}} \tfrac{3+j}{4}\lambda_j(L)^2 \;\leq\; \left(\tfrac{2}{3}\beta\right)^{\frac{2(n-1)}{\beta-1}} \tfrac{3+j}{4}\lambda_j(L)^2$$

In [46], Schnorr proposes a (semi) $2k$-BKZ reduction algorithm, that computes an almost $2k$-BKZ reduced basis in time $\mathcal{O}(n^3 k^{k+o(k)} + n^4)$. So his algorithm is polynomial in the lattice dimension n, just like LLL but it is exponential in the reduction parameter k.

In practice, yet another combination of the two basic reductions ideas is used, namely the (δ, β)-BKZ reduction, which has also been proposed by Schnorr and Euchner in [49].

Definition 1.16 ((δ,β)-BKZ)**.** Let $\delta \in]\frac{1}{4}, 1[$, and $\beta \in \{2,\ldots,n\}$, and let $B \in \mathbb{R}^{d\times n}$ be a lattice basis with Gram-Schmidt decomposition $B = \hat{B}R$. Then B is called (δ,β)-BKZ reduced if and only if

$$\begin{aligned} |r_{i,j}| &\leq \tfrac{1}{2} && \text{for } 1 \leq i < j \leq n \text{ and} \\ \delta \left\| \hat{\mathbf{b}}_j \right\|^2 &= \lambda_1(\pi_j(L(\mathbf{b}_j,\ldots,\mathbf{b}_{\min\{j+\beta-1,n\}}))) && \text{for } 1 \leq j < n. \end{aligned}$$

There is no proven time bound for (δ,β)-BKZ, but in practice (for $\beta = 2k$) its worst case behaviour seems similar to $2k$-BKZ.

RSR. In [48] Schnorr proposes a new lattice reduction method that can be used together with LLL and BKZ. It uses the property of LLL and BKZ that given a set that is slightly bigger than the lattice basis the algorithm will find the linear dependencies and still end up with a reduced basis. Even better if we add a few very short vectors to the basis given to the reduction algorithm (LLL or BKZ), we end up with a much better result.

The main idea of RSR is to construct a candidate set of vectors which have a high probability of being short and then selecting some vectors from there to add to the lattice basis prior to the real reduction done by LLL or BKZ. The proof done by Schnorr that this set does indeed contain short vectors rests upon two assumptions — the Geometric Series Assumption (GSA) and the Randomness Assumption (RA).

We will explore their motivation and state these two assumptions. Remember that LLL enforces the following $(\delta - \frac{1}{4}) \left\| \hat{\mathbf{b}}_{j-1} \right\|^2 \leq \left\| \hat{\mathbf{b}}_j \right\|^2$. From this we deduce

$$(\delta - \tfrac{1}{4})^{j-1} \| \mathbf{b}_1 \|^2 \leq \left\| \hat{\mathbf{b}}_j \right\|^2 \qquad \text{for } 1 < j \leq n$$

Which states that $\left\| \hat{\mathbf{b}}_j \right\|^2$ is bound below by $\| \mathbf{b}_1 \|^2$ times the geometric sequence of $(\delta - \frac{1}{4})$ (which is smaller than 1 for all interesting choices of δ). The same holds for BKZ. Now in practice one even observes that for most lattice bases B the Gram-Schmidt vectors of the output basis do in fact resemble such a geometric series, so for some $q_B \in [0,1]$, $(*)$ $\left\| \hat{\mathbf{b}}_j \right\| \approx q_B^{j-1} \| \mathbf{b}_1 \|^2$. Of course $(\delta - \frac{1}{4}) \leq q_B$ by the lower bound we showed before. GSA says that q_B exists for all lattices basis B and that $(*)$ is an equality.

Consider the vectors $\mathbf{w} = (w_1,\ldots,w_{n-u},\ldots,w_n) \in [0,1]^n$ whose coordinates are bounded in the following way

$$\begin{aligned} -\tfrac{1}{2} < w_j &\leq \tfrac{1}{2} && \text{for } 1 \leq j < n-u \\ -1 < w_j &\leq 1 && \text{for } n-u \leq j < n \\ w_j &= 1 && \text{for } j = n \end{aligned}$$

The candidate set used by RSR is

$$S_{u,B} := \left\{ \mathbf{v} \in L(B) \;\middle|\; \mathbf{v} = \sum_{j=1}^{n} w_j \hat{\mathbf{b}}_j \quad w_j \text{ are bounded as above} \right\}$$

For $B \in \mathbb{Z}^{d \times n}$ we can show that $|S_{u,B}| = 2^u$. Within RSR there is a sub-algorithm calls SA, which randomly selects samples from $\{0, \ldots, 2^u - 1\}$ and turns them into elements of $S_{u,B}$ like a bijective function. RA states this specific sub-algorithm preserves the randomness, i.e. that for a uniform random $x \in_R \{0, \ldots, 2^u - 1\}$ we get a $\mathbf{v} \in S_{u,B}$ whose w_j are uniformly random in their constraints

$$\begin{aligned} w_j &\in_R]-\tfrac{1}{2}, \tfrac{1}{2}] && \text{for } 1 \leq j < n - u \\ w_j &\in_R]-1, 1] && \text{for } n - u \leq j < n \end{aligned}$$

Under these two assumptions it was shown by Schnorr, that RSR computes a basis where $\| \mathbf{b}_1 \| \leq (\frac{k}{6})^{\frac{n}{2k}} \lambda_1(L)$ in expected $\mathcal{O}(n^3 (\frac{k}{6})^{\frac{k}{4}} + n^4)$ time. Where k is the RSR parameter subject to $24 \leq k$ and $3k + k \ln(k) \leq n$.

SR. Building upon RSR, Ludwig has proposed a more practical generalisation called Sampling Reduction SR [8] which is completely independent of GSA at the cost of the runtime bound given by Schnorr. It is easy however to show that SR always terminates.

As we have learned within the RSR-algorithm is a sub algorithm called SA, which for a given u takes a random element in $\{0, \ldots, 2^u - 1\}$ (u pseudo-random coin tosses) and turns it into a candidate from $S_{u,B}$ in a randomness preserving fashion (here we use RA). Ludwig does just about the same with a sub-algorithm called SAMPLE. The difference starts when these small samples are selected. RSR samples a fixed number of times using a formula to show that the chance of having found a properly small vector is sufficiently high. This formula uses GSA.

SR uses a second sub-algorithm called BESTBOUND which tries to estimate the number of samples which are needed for the search space $V_t := \{\mathbf{v}_1, \ldots \mathbf{v}_{2^{-t}}\}$ in order to guarantee a high success probability. Meaning we want a fifty-fifty chance of having found a vector which is $1/\sqrt{\gamma}$ shorter than $\mathbf{b}_1$, which is the shortest vector we know this far.

$$\mathrm{P}(\min_{\mathbf{v} \in V_t} \| \mathbf{v} \|^2 \leq \gamma \| \mathbf{b}_1 \|^2) \geq 1/2$$

SR does this in a fashion, that gives a bound at least as sharp as the formula from RSR if GSA is satisfied.

Summary. Finally we will end this section with an overview containing a good selection of the known basis reduction algorithms with their running time and SVP approximation factor.

We assume for now that the lattice L we observe has dimension n and is given by a basis of n vectors $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{n \times n}$. For practical purposes set a bound on the length of these basis vectors. We require that $\| \mathbf{b}_i \|_2 < 2^{\mathcal{O}(n)}$ for all $i \in \{1, \ldots, n\}$. The SVP approximation factor means the function $f(n)$, which relates the first vector in the reduced basis to the smallest vector λ_1 in the lattice L.

$$\| \mathbf{b}_1 \| \leq f(n)\lambda_1(L)$$

A few remarks should be made concerning the following list. The constant μ is closely connected with the arithmetic implementation which is used. Choose μ such that a multiplication of two n-bit numbers takes time n^μ to get the correct running time.

The exact SVP algorithm discovered in 2001 by Ajtai *et al*, has little practical purpose, since the $\mathcal{O}$-constant in its runtime bound $2^{\mathcal{O}(n)}$ is around 30.

The three LLL variants all have the same approximation factor which is

$$f(n) = \left(\frac{4}{4\delta - 1}\right)^{\frac{n-1}{2}} \rightarrow \left(\frac{4}{3}\right)^{\frac{n-1}{2}} \qquad \text{for } \delta \rightarrow 1.$$

Since $1/4 < \delta < 1$ has to be satisfied, we choose $\delta = 0.99$ and thus get a slight difference to the approximation factor presented. However we need to remember, that LLL reductions work much better in practice than in theory, so the given bound will usually be too large.

The (semi) $2k - BKZ$ reduction algorithm by Schnorr does not quite compute a $2k$-reduced basis, that is why the approximation function is especially only an approximation to the real result.

The RSR result, which is quite powerful rests upon two assumptions as we have learned, namely the Geometric Series Assumption (GSA) and the Randomness Assumption (RA). It also requires that $24 \leq k$ and $3k + k\ln(k) \leq n$, but this is minor. Of the two other assumptions GSA in particular is not satisfied in most cases, so Ludwig's generalisations [8] called Sampling Reduction (SR) tries to get away from GSA and rests solely upon RA.

Ludwig's Sampling Reduction algorithm is not included in the list, since there are no known runtime bounds. However the reduction he achieves are similar or surpass RSR for settings where GSA is fulfilled. The main achievement of Ludwig is, that SR often achieves RSR-like reductions even when GSA is not granted.

Koy's Primal-Dual Reduction is in fact unpublished, though the citation holds some slides describing the technique.

Reduction algorithm	SVP-$f(n)$	Runtime Bound
Exact algorithms:		
Minkowski Convex Body, Kannan (1983), [33]	1	$2^{\mathcal{O}(n \log(n))}$
Sieve, Ajtai, Kumar & Sivakumar (2001), [4]	1	$2^{\mathcal{O}(n)}$
LLL-Variants:		
LLL, Lenstra, Lenstra & Lovász (1982), [39]	$\left(\frac{4}{3}\right)^{\frac{n-1}{2}}$	$\mathcal{O}(n^{5+2\mu})$
Floating-Point LLL, Schnorr & Euchner (1994), [49]	$\left(\frac{4}{3}\right)^{\frac{n-1}{2}}$	$\mathcal{O}(n^{5+\mu})$
Strong-Segment LLL, Koy & Schnorr (2002), [37]	$\left(\frac{4}{3}\right)^{\frac{n-1}{2}}$	$\mathcal{O}(n^{3+\mu}\log(n))$
BKZ-Variants:		
(semi) $2k$-BKZ, Schnorr (1987), [46]	$\left(\frac{4}{3}k\right)^{\frac{n-1}{2k-1}}$	$\mathcal{O}(n^3k^{k+o(k)}+n^4)$
RSR, Schnorr (2003), [48]	$\left(\frac{k}{6}\right)^{\frac{n}{2k}}$	$\mathcal{O}(n^3(\frac{k}{6})^{\frac{k}{4}}+n^4)$
Primal-Dual Reduction, Koy (2004), [36]	$\left(\frac{k}{6}\right)^{\frac{n}{k}}$	$\mathcal{O}(n^3k^{\frac{k}{2}+o(k)}+n^4)$

Table 1: Basis reduction overview

2 NTRU Systems

First of all, the abbreviation NTRU stands for "Number Theorists R Us". It was meant as a joke by the three inventors, Jeffrey Hoffstein, Jill Pipher and Joseph H. Silverman who are all mathematicians at Brown University in Providence, Rhode Island, America. Business people did not like this playful phrase however, so it was changed to "Number Theory Research Unit".

In this section we will explain both NTRUEncrypt, a system for public key cryptography, and NTRUSign, a digital signature scheme.

We start by introducing the public parameter-sets upon which both systems operate. Then we take a closer look first at NTRUEncrypt and finally at NTRUSign. We will see how both systems can be used.

2.1 Parameters

We will start with the two public parameters that both systems have in common

$N \in \mathbb{N}$	*Degree Parameter*	N should be prime [14].
$q \in \mathbb{N}$	*Large Modulus*	q should be a prime-power.

The parameter N is used to define the basic superset in which all further work takes place. It is the set $\mathcal{R}$ of all polynomials which have a degree smaller than N. We define it with the following quotient:

$$\mathcal{R} := \mathbb{Z}[X] \,/\, (X^N - 1).$$

The ring $\mathcal{R}$ is not finite yet, but this where q comes into play. We restrict the coefficients of our polynomials to come from $\mathbb{Z}/q\mathbb{Z}$ and get:

$$\mathcal{R}_q := (\mathbb{Z}/q\mathbb{Z})\,[X] \,/\, (X^N - 1).$$

If we use the convolution product '$*$' (see definition 1.4) to turn $(\mathbb{Z}^N, +)$ into a $\mathbb{Z}$-algebra $(\mathbb{Z}^N, +, *)$, this algebra is isomorphic to $(\mathcal{R}, +, \cdot)$ where '$\cdot$' is the normal polynomial multiplication in $\mathcal{R}$. The isomorphism is given by the map Φ which identifies the i-th coefficient with the i-th coordinate of the vector.

$$\Phi \;:\; \mathcal{R} \longrightarrow \mathbb{Z}^N \;:\; f_0 + \cdots + f_{N-1}\, X^{N-1} \mapsto \begin{pmatrix} f_0 & \cdots & f_{N-1} \end{pmatrix}^T$$

Analogously, we can show that $\mathcal{R}_q \cong (\mathbb{Z}/q\mathbb{Z})^N$. Therefore we need not make a distinction between vectors and polynomials when we work with these rings. From now on, we will write elements in bold if we want to emphasise that they are in $\mathcal{R}$ or $\mathcal{R}_q$, e.g. $\mathbf{x} \in \mathcal{R}$.

Our notation for some special elements of $\mathcal{R}$: The zero polynomial we denote by $\mathbf{0}$, the one polynomial is $\mathbf{e}$ ($*$ neutral element), the polynomial with *all* coefficients equal to one we call $\mathbf{1}$, and the polynomial which is equal to q is $q\mathbf{e} = \mathbf{q}$. Also note that when we use the $*$ multiplication, it *always* includes a reduction modulo $(X^N - 1)$.

Note that q should be a prime-power, because otherwise the ring $\mathcal{R}_q$ can be decomposed using the Chinese remainder theorem. This is done using $d \in \mathbb{Z}$ such that $\gcd(q/d, d) = 1$.

$$\mathcal{R}_q \cong \mathcal{R}_{q/d} \times \mathcal{R}_d$$

If, on the other hand, q is a prime-power, say m^k for some $k > 1$, we may not decompose $\mathcal{R}_q$, but we still get a normal subgroup. Namely for each $l \in \mathbb{Z}$ that divides k, we have $\mathcal{R}_{m^l} \triangleleft \mathcal{R}_q$. There is no known way to exploit this however, so the parameter q will usually be a large power of 2.

We now look at another parameter exclusive to NTRUEncrypt.

$p \in \mathbb{Z}[X]$ *Small Modulus* p must be coprime to q.

For efficiency reasons, p is usually a small prime number like 3 and not a polynomial.

2.2 Norms

We use the basic parameters N and q to define the following semi-norms, which we will use throughout this work.

Definition 2.1 (Centred Euclidean Norm)**.** In $\mathbb{R}^N$ we define the *mean value* of a vector $\mathbf{v} = (v_1 \cdots v_N)^T$ to be $\mu(\mathbf{v}) := \frac{1}{N}(v_1 + v_2 + \cdots + v_N)$, and the *centred Euclidean norm* $\|\cdot\|_\mu$ is defined as the Euclidean norm of the vector minus its mean.

$$\| \mathbf{v} \|_\mu := \| \mathbf{v} - \mu(\mathbf{v})\mathbf{1} \|_2$$

This semi-norm is closely related to the standard deviation: We have that $\| \mathbf{v} \|_\mu = \sqrt{N}\sigma(\mathbf{v})$. Notice that for any vector $\mathbf{v} \in \mathbb{R}^N$, the vector $(\mathbf{v} - \mu(\mathbf{v})\mathbf{1})$ is the projection of $\mathbf{v}$ into the space perpendicular to $\mathbf{1}$. This explains the following formula, by which we calculate this semi-norm:

$$\begin{aligned} \| \mathbf{v} \|_2^2 - \mu(\mathbf{v})^2 N &= \| \mathbf{v} \|_2^2 - \| \mu(\mathbf{v})\mathbf{1} \|_2^2 \\ &= \| \mathbf{v} - \mu(\mathbf{v})\mathbf{1} + \mu(\mathbf{v})\mathbf{1} \|_2^2 - \| \mu(\mathbf{v})\mathbf{1} \|_2^2 \\ &= \| \mathbf{v} - \mu(\mathbf{v})\mathbf{1} \|_2^2 + \| \mu(\mathbf{v})\mathbf{1} \|_2^2 - \| \mu(\mathbf{v})\mathbf{1} \|_2^2 \\ &= \| \mathbf{v} - \mu(\mathbf{v})\mathbf{1} \|_2^2 = \| \mathbf{v} \|_\mu^2 \end{aligned}$$

The following interesting property holds for $\|\cdot\|_\mu$.

Lemma 2.2. *The centred Euclidean norm is almost multiplicative for random vectors in* $(\mathbb{Z}^N, *)$.

$$\| \mathbf{v} * \mathbf{w} \|_\mu \approx \| \mathbf{v} \|_\mu \| \mathbf{w} \|_\mu$$

A proof of this is given in [11].

We will use the centred Euclidean norm on $\mathbb{Z}^N \cong \mathcal{R}$ to make a semi-norm on $\mathcal{R}_q \cong (\mathbb{Z}/q\mathbb{Z})^N$. By taking the μ-norm of the best representative, we define for all $\mathbf{v} \in \mathcal{R}_q$:

$$\| \mathbf{v} \|_q := \min_{\mathbf{f} \in \mathcal{R}} \left\{ \| \mathbf{f} \|_\mu \;\middle|\; \mathbf{f} \equiv \mathbf{v} \mod q \right\}.$$

This q-norm retains the multiplication property of the μ-norm. We continue by once more taking this semi-norm to define the L-semi-norm for the elements of any lattices $L(\mathbf{h}, q) \subseteq \mathcal{R}_q^2$. We set for all $(\mathbf{v}, \mathbf{w})^T \in \mathcal{R}_q^2$:

$$\left\| \begin{pmatrix} \mathbf{v} \\ \mathbf{w} \end{pmatrix} \right\|_L := \sqrt{\| \mathbf{v} \|_q^2 + \| \mathbf{w} \|_q^2}.$$

Note that $\| (\mathbf{v}, \mathbf{w})^T \|_L$ is *not* the same as $\| (\mathbf{v}, \mathbf{w})^T \|_q$, because in the first case, the mean is subtracted in each N-component separately, and in the second one the mean of the whole vector is subtracted from every entry.

Sometimes we will want to scale the second component a little with a scaling constant β using the following β-semi-norm. We define again for all $(\mathbf{v}, \mathbf{w})^T \in \mathcal{R}_q^2$:

$$\left\| \begin{pmatrix} \mathbf{v} \\ \mathbf{w} \end{pmatrix} \right\|_\beta := \left\| \begin{pmatrix} \mathbf{v} \\ \beta\mathbf{w} \end{pmatrix} \right\|_L = \sqrt{\| \mathbf{v} \|_q^2 + \beta^2 \| \mathbf{w} \|_q^2}.$$

2.3 NTRUEncrypt

NTRUEncrypt is a public key cryptosystem. It was first presented during the rump session of Crypto '96 by three mathematicians from Brown University, Rhode Island. They were Jeffrey Hoffstein, Jill Pipher and Joseph H. Silverman. These three joined forces with Daniel Lieman, also a Brown mathematician, and together they founded "NTRU Cryptosystems," also in 1996, to market their cryptography ideas. The full paper on NTRUEncrypt was presented at the Algorithmic Number Theory Symposium (ANTS III) in 1998 [22].

NTRUEncrypt's features are a high en- and decryption speed, as well as low memory requirements. The security of NTRUEncrypt is derived from a polynomial mixing system over the finite rings $\mathcal{R}_q$ and $\mathcal{R}_p$. We will now see how NTRUEncrypt can be used in the following scenario, where Bob wants to receive a secret message from Alice.

Setting up. We choose N, q and p as described above. Then we choose $d_f, d_g, d_r \in \mathbb{N}$ and using

$$\mathcal{B}(d) := \left\{ \mathbf{f} \in \mathcal{R} \;\middle|\; \mathbf{f} \text{ is } \textit{binary} \text{ with } d \text{ ones} \right\},$$

we define:

$\mathcal{D}_m$	*Plaintext Space*	to be $\mathcal{R}_p$
$\mathcal{D}_f, \mathcal{D}_g$	*Key Spaces*	to be $\mathcal{B}(d_f)$ and $\mathcal{B}(d_g)$
$\mathcal{D}_r$	*Blinding Space*	to be $\mathcal{B}(d_r)$

and with these spaces we are all set.

Key creation. Bob creates first his private and then his public key. He starts by randomly choosing $\mathbf{f} \in \mathcal{D}_f$ and $\mathbf{g} \in \mathcal{D}_g$. Bob has to check whether $\mathbf{f}$ is invertible in $\mathcal{R}_{pq}$, or equivalently whether $\mathbf{f}$ is invertible in both $\mathcal{R}_p$ and $\mathcal{R}_q$. Bob may proceed if the inverses of $\mathbf{f}$ exist in both rings. Otherwise he has to select a new $\mathbf{f}$ randomly and try again. Once Bob finds an $\mathbf{f}$ which has both inverses, he uses the pair $(\mathbf{f}, \mathbf{g})$ as his private key.

The inverse of $\mathbf{f}$ in $\mathcal{R}_p$ will be denoted by $\mathbf{f}_p^{-1}$, and $\mathbf{f}_q^{-1}$ is the corresponding inverse in $\mathcal{R}_q$. Bob continues by computing his public key $\mathbf{h}$ defined by

$$\mathbf{h} := \mathbf{f}_q^{-1} * \mathbf{g} \mod q.$$

Encryption. Before encryption can start, the message must be converted into a polynomial $\mathbf{m}$ from $\mathcal{D}_m$ in a well known way. Now Alice wants to send her message to Bob and she starts by selecting a random blinding polynomial $\mathbf{r} \in \mathcal{D}_r$ which is used to scramble the message. Then she uses Bobs public key to compute

$$\mathbf{c} = p\,\mathbf{r} * \mathbf{h} + \mathbf{m} \mod q$$

which is the encrypted message she sends to Bob.

Decryption. In order to decrypt the message $\mathbf{c}$, which Bob just received, he calculates

$$\mathbf{a} = \mathbf{f} * \mathbf{c} \mod q$$

and chooses the coefficients to be within $]-q/2, q/2]$. This choice of coefficients is called *centring*. Then he takes this special representative of $\mathbf{a}$ and uses it to calculate

$$\mathbf{f}_p^{-1} * \mathbf{a} \mod p$$

In most cases this will yield the original message $\mathbf{m}$. The probability of a failure during this decryption can be pushed below 2^{-104} (as shown in [26]) and recently NTRU claims that the parameter sets provided in [28] avoid decryption failures altogether, by ensuring that the coefficients of $\mathbf{a} = p\,\mathbf{r} * \mathbf{g} + \mathbf{f} * \mathbf{m}$ lie in an interval of length q.

Why decryption works. Bob decrypts the message calculating $\mathbf{a}$:

$$\begin{aligned}\mathbf{a} &= \mathbf{f} * \mathbf{c} \mod q \\ &= p\,\mathbf{r} * \mathbf{g} + \mathbf{f} * \mathbf{m} \mod q.\end{aligned}$$

Then he centres $\mathbf{a}$ (here we need that the coefficients of $\mathbf{a}$ all lie in an interval of length q), reduces it modulo p, and multiplies with $\mathbf{f}_p^{-1}$, yielding

$$\begin{aligned}\mathbf{f}_p^{-1} * \mathbf{a} &= \mathbf{f}_p^{-1} * p\,\mathbf{r} * \mathbf{g} + \mathbf{f}_p^{-1} * \mathbf{f} * \mathbf{m} \mod p \\ &= \mathbf{m}.\end{aligned}$$

which was the original message.

Padding. NTRUEncrypt should not be used without padding. There is a simple chosen ciphertext attack (explained in [31]) which exploits the lack of padding. To counter this attack, one may use NAEP, an efficient NTRUEncrypt padding scheme (explained in [27]). We will not introduce it here, since our focus is on NTRUSign.

2.4 NTRUSign

NTRUSign was first presented at the rump session of AsiaCrypt 2001. Again, Jeffrey Hoffstein, Jill Pipher and Joseph H. Silverman had worked on the preprint, but they had support this time from Nick Howgrave-Graham and William Whyte. Nick Howgrave-Graham worked for two years at the IBM, Yorktown Heights Research Lab in New York and William Whyte was Senior Cryptographer with Baltimore Technologies in Dublin, Ireland before they both co-invented NTRUSign and joined NTRU Cryptosystems. The full paper on NTRUSign was published at the RSA Conference 2003 [20].

NTRUSign is the first efficient realisation of a proposal made in 1997 by Goldreich, Goldwasser and Halevi [16]. They suggested that digital signatures could be made by demonstrating the ability to solve an approximate closest vector problem ($\mathcal{N}$-appr-CVP) in a given lattice.

This parameter $\mathcal{N}$ is a public parameter in an NTRUSign setting. So let us now find out which lattices we are going to use.

2.4.1 NTRU Lattice

We remember the public parameters $N, q \in \mathbb{N}$ as well as convolution modular lattices from section 1.1. Now, for a polynomial $\mathbf{h}$ we define the NTRU lattice to be the convolution modular lattice $L(\mathbf{h}, q)$.

Spaces. The space of signable messages is $\mathcal{M} := \left(\mathbb{Q}[X] \,/\, (X^N - 1)\right)^2 \cong \mathbb{Q}^{2N}$. We will need a hash function that maps the document we want to sign in there.

The secret key spaces in NTRUSign are as follows. For a chosen parameter $d \in \mathbb{N}$ we define

$$\begin{aligned}
\mathcal{B}(d) &:= \left\{ \mathbf{f} \in \mathcal{R} \;\middle|\; \mathbf{f} \text{ is } \textit{binary} \text{ with } d \text{ ones.} \right\} \\
\mathcal{T}(d) &:= \left\{ \mathbf{f} \in \mathcal{R} \;\middle|\; \mathbf{f} \text{ is } \textit{trinary} \text{ with } d \text{ ones and } (d-1) \text{ minus ones.} \right\} \\
\delta(d) &:= \tfrac{2d+1}{N} - \tfrac{1}{N^2} \\
\mathcal{S}(d) &:= \left\{ \mathbf{f} \in \mathcal{R} \;\middle|\; \delta(d)N - 1 < \|\,\mathbf{f}\,\|_\mu^2 < \delta\,(d)N + 1 \right\}.
\end{aligned}$$

Here *Binary* means all polynomials with coefficients in $\{\,0,\,1\,\}$ and *trinary* analogously all polynomials with coefficients in $\{\,-1,\,0,\,1\,\}$.

Our secret key spaces have the form $\mathcal{K} = \mathcal{S}\,(d)$. So it is the set of polynomials which are *close* to trinary ones. In practice however, $\mathcal{K}$ is really just $\mathcal{T}(d)$, and of course we may use different ds for each key, so we get the spaces $\mathcal{K}_\mathbf{f} := \mathcal{T}(d_\mathbf{f})$ and $\mathcal{K}_\mathbf{g} := \mathcal{T}(d_\mathbf{g})$ for two natural numbers $d_\mathbf{f}$ and $d_\mathbf{g}$.

We are not forced to use trinary keys though. NTRUSign works just as well with binary ones, using $\mathcal{K}_\mathbf{f} := \mathcal{B}(d_\mathbf{f})$ and $\mathcal{K}_\mathbf{g} := \mathcal{B}(d_\mathbf{g})$. However, the length of keys differ in both cases:

$$\begin{aligned}
&\Big(\forall \mathbf{f} \in \mathcal{B}(d_\mathbf{f})\Big) \;\; \|\,\mathbf{f}\,\|_\mu = \sqrt{d_\mathbf{f}(1 - (d_\mathbf{f}/N))} \\
&\Big(\forall \mathbf{f} \in \mathcal{T}(d_\mathbf{f})\Big) \;\; \|\,\mathbf{f}\,\|_\mu = \sqrt{\delta(d_\mathbf{f})N}.
\end{aligned}$$

Key creation. We repeatedly choose $(\mathbf{f}, \mathbf{g})$ randomly in $\mathcal{K}_\mathbf{f} \times \mathcal{K}_\mathbf{g}$ until we have a pair where $\mathbf{f}$ is invertible in $\mathcal{R}_q$ and the following condition holds:

$$\gcd\big(\text{ resultant}\,(\mathbf{f}, X^N - 1),\, \text{resultant}\,(\mathbf{g}, X^N - 1)\,\big) = 1. \tag{3}$$

Here, a resultant is defined as follows:

Definition (Resultant)**.** For two polynomials $\mathbf{f}$ and $\mathbf{g}$ with leading coefficient equal to 1 we define

$$\text{resultant}\,(\mathbf{f}, \mathbf{g}) := \prod_{\mathbf{g}(x)=0} \mathbf{f}(x)$$

We will also need $\mathbf{g}$ to be invertible in $\mathcal{R}_q$ if we wish to use the *transpose* instead of the *standard* NTRU lattice. What the transpose lattice is and why we should wish to use it will be explained later.

We will see why condition (3) is necessary, when we create a lattice basis from $(\mathbf{f}, \mathbf{g})$. This pair will be our private key, and we create the public key as before by taking $\mathbf{h} = \mathbf{f}_q^{-1} * \mathbf{g} \mod q$.

2.4.2 Generating the Basis

The goal now is to use these keys to create two bases for $L(\mathbf{h}, q)$. To do this, we use another isomorphism similar to $\Phi : \mathcal{R} \to \mathbb{Z}^N$ (see 2.1):

$$\Psi \ : \ \mathbb{Z}^N \longrightarrow \mathbb{Z}^{N\times N} \ : \ \mathbf{v} \mapsto \begin{pmatrix} \rho^0(\mathbf{v}) & \cdots & \rho^{N-1}(\mathbf{v}) \end{pmatrix}.$$

The mapping Ψ which maps a vector to a matrix contain all its rotations is a ring-homomorphism with trivial kernel. So Ψ is an isomorphism onto its image. The image is composed of all circulant matrices of dimension N^2 over $\mathbb{Z}$. We call it $\mathcal{C}_N(\mathbb{Z}) := \mathrm{Im}(\Psi)$. Using Φ and Ψ, we see that these rings are all isomorphic: $\mathcal{R} \cong \mathbb{Z}^N \cong \mathcal{C}_N(\mathbb{Z})$.

We will view the NTRU lattice $L(\mathbf{h}, q)$ as an $\mathcal{R}$-module of rank 2 and search for two $\mathcal{R}$-bases. The $\mathcal{R}$-basis corresponding to the one from section 1.1 is

$$\mathcal{B}_{\mathsf{pub}} := \left\{ \begin{pmatrix} \mathbf{e} \\ \mathbf{h} \end{pmatrix}, \begin{pmatrix} \mathbf{0} \\ \mathbf{q} \end{pmatrix} \right\}.$$

Generating the second, private basis is a much harder issue. We will extend the private key $(\mathbf{f}, \mathbf{g})^T$ with another vector $(\mathbf{F}, \mathbf{G})^T$ such that the both of them $\{ (\mathbf{f}, \mathbf{g})^T, (\mathbf{F}, \mathbf{G})^T \}$ form a basis of $L(\mathbf{h}, q)$. Then we will make the length of the second basis vector $\left\| (\mathbf{F}, \mathbf{G})^T \right\|_L$ as small as possible. The size of the basis vectors is the fundamental difference between the public and the private basis.

In order to create the private basis, we will need to blow up $(\mathbf{f}, \mathbf{g})^T$ with another pair $(\mathbf{F}, \mathbf{G})^T$, satisfying

$$\mathbf{f} * \mathbf{G} - \mathbf{F} * \mathbf{g} = \mathbf{q}. \tag{4}$$

Then, $\{(\mathbf{f}, \mathbf{g})^T, (\mathbf{F}, \mathbf{G})^T\}$ really is a basis for $L(\mathbf{h}, q)$, since we can transform the public basis into this one:

$$\mathcal{B}_{\mathsf{pub}} = \begin{pmatrix} \mathbf{e} & \mathbf{0} \\ \mathbf{h} & \mathbf{q} \end{pmatrix} \underbrace{\begin{pmatrix} \mathbf{f} & \mathbf{F} \\ \frac{\mathbf{g}-\mathbf{f}*\mathbf{h}}{q} & \frac{\mathbf{G}-\mathbf{F}*\mathbf{h}}{q} \end{pmatrix}}_{=:T} = \begin{pmatrix} \mathbf{f} & \mathbf{F} \\ \mathbf{g} & \mathbf{G} \end{pmatrix}.$$

Note that $0 \equiv \mathbf{g} - \mathbf{f} * \mathbf{h} \overset{(4)}{\equiv} \mathbf{G} - \mathbf{F} * \mathbf{h} \mod q$, so T is well-defined. The matrix T is a transformation if its determinant is an $\mathcal{R}$ unit:

$$\begin{aligned} \det(T) &= \frac{\mathbf{f} * (\mathbf{G} - \mathbf{F} * \mathbf{h}) - \mathbf{F} * (\mathbf{g} - \mathbf{f} * \mathbf{h})}{q} \\ &= \frac{\mathbf{f} * \mathbf{G} - \mathbf{F} * \mathbf{g}}{q} \\ &\overset{(4)}{=} \mathbf{e}. \end{aligned}$$

In order to really find $(\mathbf{F}, \mathbf{G})^T$ satisfying (4), we will need condition (3), mentioned before. For (3) to hold, both resultants need to be non-zero, hence $\mathbf{f}$ and $(X^N - 1)$, as well as $\mathbf{g}$ and $(X^N - 1)$, have to be coprime.

We use the *sub-resultant gcd* algorithm on the two pairs to get $u, v, k, l \in \mathbb{Z}[X]$ such that

$$u * \mathbf{f} + k * (X^N - 1) = \text{resultant}\,(\mathbf{f},\, X^N - 1)$$
$$v * \mathbf{g} + l * (X^N - 1) = \text{resultant}\,(\mathbf{g},\, X^N - 1).$$

Note that we cannot use Euclid's algorithm here, since $\mathbb{Z}[X]$ is *not* a Euclidean ring, but only a unique factorisation domain. However, the sub-resultant gcd algorithm works almost as good in this context. For two polynomials a and b, it computes two new ones, A and B, such that

$$A \cdot a + B \cdot b = \text{resultant}\,(a, b) \cdot \gcd(a, b).$$

A description is given in [10, Algorithm 3.3.7], and a complete proof of validity for this algorithm is given in [35].

Since both resultants are coprime by (3), we can find integers α and β using Euclid's algorithm in $\mathbb{Z}$ now such that

$$\begin{array}{rcrcl} \alpha\ \text{resultant}\,(\mathbf{f}, (X^N - 1)) & + & \beta\ \text{resultant}\,(\mathbf{g}, (X^N - 1)) & = & 1 \\ \alpha\,(u * \mathbf{f} + k * (X^N - 1)) & + & \beta\,(v * \mathbf{g} + l * (X^N - 1)) & = & 1. \end{array}$$

We use the last equation to see that (4) is satisfied by

$$\mathbf{F}_{\mathsf{big}} := -\beta q\, v \mod (X^N - 1) \qquad \mathbf{G}_{\mathsf{big}} := \alpha q\, u \mod (X^N - 1).$$

We use the index big to indicate that the norm of this pair could still be too big to be used in a private basis.

If we retrace our steps, it is clear that our condition (3) was quite strong. Had the gcd of the resultants been any divisor d of q, then we could have constructed a pair $(\mathbf{F}_{\mathsf{big}}, \mathbf{G}_{\mathsf{big}})^T$ in a similar fashion.

We know that $\{(\mathbf{f}, \mathbf{g})^T, (\mathbf{F}_{\mathsf{big}}, \mathbf{G}_{\mathsf{big}})^T\}$ is a basis of $L(\mathbf{h}, q)$. In order to get $(\mathbf{F}_{\mathsf{big}}, \mathbf{G}_{\mathsf{big}})^T$ small, we may subtract any $\mathcal{R}$-multiple of $(\mathbf{f}, \mathbf{g})^T$ from it. We will use "Babai's inverting and rounding technique" for this. We solve the problem exactly in a bigger ring where inverting is possible, in this case $\mathcal{Q} := \mathbb{Q}[X]/(X^N - 1)$, and then round off the solution to get a fairly good one in $\mathcal{R}$.

We want to call the factor which we will use for the big polynomial $\mathbf{k} \in \mathcal{R}$ and $\mathbf{l} \in \mathcal{Q}$, depending on which ring we are in. To get from $\mathbf{l}$ to $\mathbf{k}$ we simply round each coefficient to its nearest integer. We denote this by $\mathbf{k} := \lfloor \mathbf{l} \rceil$. This leaves a polynomial of differences, $\mathbf{l}' := \mathbf{l} - \mathbf{k}$ whose coefficients are in

$]-1/2, 1/2]$. We want to find an $\mathbf{l}$ such that $\| \mathbf{F}_{\mathsf{big}} - \mathbf{l} * \mathbf{f} \|_\mu^2 + \| \mathbf{G}_{\mathsf{big}} - \mathbf{l} * \mathbf{g}) \|_\mu^2$ is minimal. To get just one summand small it would suffice to take

$$\mathbf{l}_\mathbf{F} := \mathbf{F}_{\mathsf{big}} * \mathbf{f}_\mathcal{Q}^{-1} \qquad\qquad \mathbf{l}_\mathbf{G} := \mathbf{G}_{\mathsf{big}} * \mathbf{g}_\mathcal{Q}^{-1}.$$

We try to see how far apart these choices are.

$$\begin{aligned} \| \mathbf{l}_F - \mathbf{l}_G \|_\mu &= \left\| \mathbf{F}_{\mathsf{big}} * \mathbf{f}_\mathcal{Q}^{-1} - \mathbf{G}_{\mathsf{big}} * \mathbf{g}_\mathcal{Q}^{-1} \right\|_\mu \\ &= \left\| (\mathbf{F}_{\mathsf{big}} * \mathbf{g} - \mathbf{G}_{\mathsf{big}} * \mathbf{f}) * \mathbf{f}_\mathcal{Q}^{-1} * \mathbf{g}_\mathcal{Q}^{-1} \right\|_\mu \\ &\overset{(4)}{=} \left\| \mathbf{q} * \mathbf{f}_\mathcal{Q}^{-1} * \mathbf{g}_\mathcal{Q}^{-1} \right\|_\mu \\ &\approx q\sqrt{1 - 1/N}\, (\| \mathbf{f} * \mathbf{g} \|_\mu)^{-1} \end{aligned}$$

So the choices for $\mathbf{l}$ are really close, and either will do. We can even do better and minimise both summands to get a better result by taking

$$\mathbf{l}_\mathbf{FG} := (\bar{\mathbf{f}} * \mathbf{F} + \overline{\mathbf{g}} * \mathbf{G})\, (\mathbf{f} * \bar{\mathbf{f}} + \mathbf{g} * \overline{\mathbf{g}})_\mathcal{Q}^{-1},$$

where $\bar{\mathbf{f}}$ is the polynomial $\mathbf{f}$ evaluated at $(1/X)$ instead of X. In other words, the coefficients of $\mathbf{f}$ are permuted such that the i-th coefficient becomes the $(N - i)$-th coefficient.

We finally get the reduced $\mathbf{F}$ and $\mathbf{G}$:

$$\mathbf{F} := \mathbf{F}_{\mathsf{big}} - \lfloor \mathbf{l}_\mathbf{FG} \rceil * \mathbf{f} \qquad\qquad \mathbf{G} := \mathbf{G}_{\mathsf{big}} - \lfloor \mathbf{l}_\mathbf{FG} \rceil * \mathbf{g}.$$

And with this, we can complete the *small* private basis:

$$\mathcal{B}_{\mathsf{pri}} = \left\{ \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix}, \begin{pmatrix} \mathbf{F} \\ \mathbf{G} \end{pmatrix} \right\}.$$

Length of the private basis vectors. The μ-norm of $\mathbf{f}$ and $\mathbf{g}$ is is very small by definition. So let us look at the μ-norm of $\mathbf{F}$ and $\mathbf{G}$ which we made small. We have

$$\| \mathbf{F} \|_\mu = \left\| \mathbf{l}'_\mathbf{FG} * \mathbf{f} \right\|_\mu \approx \left\| \mathbf{l}'_\mathbf{F} * \mathbf{f} \right\|_\mu,$$

because $\mathbf{l}_\mathbf{FG}$ is just very close but not equal to $\mathbf{l}_\mathbf{F}$. To proceed, we make the assumption that the coefficients of $\mathbf{l}'_\mathbf{F}$, which are in the interval $]-1/2,\, 1/2]$, are uniformly distributed.

We see these N coefficients as realisations of independent identically distributed random variables $X_1, \ldots, X_N \overset{\text{iid}}{\sim} U(-1/2, 1/2)$.

$$\mathrm{E}(\left\| \mathbf{l}'_\mathbf{F} \right\|_\mu^2) = \mathrm{E}(\sum_{i=1}^{N} X_i^2 - \mathrm{E}(X_i)^2) = \sum_{i=1}^{N} \mathrm{E}(X_i^2) = N\ \mathrm{Var}(X_1) = \frac{N}{12}$$

We see that $\left\| \mathbf{l}'_{\mathbf{F}} \right\|_\mu^2 \longrightarrow N/12$, which gives us the desired $\left\| \mathbf{l}'_{\mathbf{F}} \right\|_\mu \longrightarrow \sqrt{N/12}$.

$$\begin{aligned} \left\| \mathbf{l}'_{\mathbf{F}} * \mathbf{f} \right\|_\mu &\approx \left\| \mathbf{l}'_{\mathbf{F}} \right\|_\mu \left\| \mathbf{f} \right\|_\mu \\ &= \sqrt{N/12} \left\| \mathbf{f} \right\|_\mu \end{aligned}$$

Remember that $\left\| \mathbf{f} \right\|_\mu$ is different depending on the type of keys we use.

$$\begin{aligned} (\textit{binary} \text{ case}) \ \left\| \mathbf{F} \right\|_\mu &= \sqrt{(d_{\mathbf{f}}/12)(N - d_{\mathbf{f}})} \\ (\textit{trinary} \text{ case}) \ \left\| \mathbf{F} \right\|_\mu &= N \sqrt{\delta(d_{\mathbf{f}})/12} \end{aligned}$$

So the μ-norm of $\mathbf{F}$ is small either way. By the same reasoning, we can see that $\mathbf{G}$ is small and so we can be satisfied with the short vectors in our private basis.

Remark. There is another way of doing this, leading to a slightly different μ-norm estimate for $\mathbf{F}$ and $\mathbf{G}$. Instead of assuming that the coefficients of $\mathbf{l}'_{\mathbf{F}}$ are realisations of N independent random variables $X_1,\ldots,X_N$, we may assume that they are N realisations of one and the same random variable $X \sim U(-1/2, 1/2)$. In this case, $\left\| \mathbf{l}'_{\mathbf{F}} \right\|_\mu^2 / (N-1)$ is an *unbiased* estimator of $\mathrm{Var}(X) = 1/12$.

$$\left\| \mathbf{l}'_{\mathbf{F}} \right\|_\mu \approx \sqrt{(N-1)/12}$$

We assume that there is no practical difference between these two ways of estimating $\left\| \mathbf{l}'_{\mathbf{F}} \right\|_\mu$.

The transpose lattice. There is an alternative lattice to the one we have built so far. It uses a different basis which is closely related to the ones we have. We define the transpose private basis to be the transpose of the basis we have so far:

$$\mathcal{B}_{\mathsf{pri}}^T = \begin{pmatrix} \mathbf{f} & \mathbf{F} \\ \mathbf{g} & \mathbf{G} \end{pmatrix}^T = \begin{pmatrix} \mathbf{f} & \mathbf{g} \\ \mathbf{F} & \mathbf{G} \end{pmatrix}.$$

We observe how a matching public basis would look:

$$\mathcal{B}_{\mathsf{pri}}^T = \begin{pmatrix} \mathbf{f} & \mathbf{g} \\ \mathbf{F} & \mathbf{G} \end{pmatrix} = \underbrace{\begin{pmatrix} \mathbf{e} & \mathbf{0} \\ \mathbf{h}' & \mathbf{q} \end{pmatrix}}_{=:\mathcal{B}_{\mathsf{pub}}^T} \underbrace{\begin{pmatrix} \mathbf{f} & \mathbf{g} \\ \frac{\mathbf{F}-\mathbf{f}*\mathbf{h}'}{q} & \frac{\mathbf{G}-\mathbf{g}*\mathbf{h}'}{q} \end{pmatrix}}_{=:T}.$$

The determinant of T is $\mathbf{e}$ again, thanks to (4). However, T is only well-defined if $0 \equiv \mathbf{F} - \mathbf{f} * \mathbf{h}' \equiv \mathbf{G} - \mathbf{g} * \mathbf{h}' \mod q$. This can be achieved with

$$\mathbf{h}' := \mathbf{f}_q^{-1} * \mathbf{F} \mod q \qquad \mathbf{h}' \overset{(4)}{=} \mathbf{g}_q^{-1} * \mathbf{G} \mod q.$$

So this $\mathbf{h}'$ defines the transpose NTRU lattice $L(\mathbf{h}', q)$. Using the transpose instead of the standard lattice makes several differences. For example, the smallest known vector in $L(\mathbf{h}', q)$ is $(\mathbf{f}, \mathbf{F})^T$, which is bigger than $(\mathbf{f}, \mathbf{g})^T$. Another difference which we will see, is that when using the transpose lattice, signing only involves multiplications with the small polynomials $\mathbf{f}$ and $\mathbf{g}$, making it faster.

This whole secret-basis generation process can be approached in a different way for both the standard and the transpose NTRU lattice. After we pick our secret keys $\mathbf{f}$ and $\mathbf{g}$, we can easily calculate $\mathbf{h}$, and the public basis. Then we create the following generating system, where Ψ maps a vector to the matrix containing all N of its rotations.

$$\mathcal{B}'_{\mathsf{priv}} = \begin{pmatrix} \Psi(\mathbf{f}) & \Psi(\mathbf{e}) & \Psi(\mathbf{0}) \\ \Psi(\mathbf{g}) & \Psi(\mathbf{h}) & \Psi(\mathbf{q}) \end{pmatrix}$$

This is just the normal public basis with all rotations of the private keys up front. We use BKZ with minimal parameters $\beta = 2$ on this generating system, to reduce it to a basis. This will not take long, and the second shortest vector in the reduced basis will be our $(\mathbf{F}, \mathbf{G})^T$. Of course we have to check whether

$$\mathbf{f} * \mathbf{G} - \mathbf{g} * \mathbf{F} \equiv \mathbf{q}$$

to ensure that $(\mathbf{f}, \mathbf{g})^T$ and $(\mathbf{F}, \mathbf{G})^T$ really do form a basis of $L(\mathbf{h}, q)$. If this condition does not hold, we should reselect our private keys $\mathbf{f}$ and $\mathbf{g}$ and try again. This is just as unlikely as before. For the transpose lattice, we proceed just like before.

Due to limited time, we were only able to perform a few experiments with this new approach. However, the results indicate, that this way of completing the basis produces slightly better results (i.e a smaller private basis), than what NTRU suggests in [20] (which is the approach we described before this one).

2.4.3 Using the Basis

Signing. In order to sign a document D, we will need to hash it into the space of signable messages $\mathcal{M} = \mathcal{Q}$. We will actually sign this hash of D, which we call $(\mathbf{m}_1, \mathbf{m}_2)^T$. Whoever can solve an APPR-CVP of $(\mathbf{m}_1, \mathbf{m}_2)^T$ in the lattice $L(\mathbf{h}, q)$ has signed D.

Notice that we need not actually solve the APPR-CVP for $(\mathbf{m}_1, \mathbf{m}_2)^T$, but for $(\mathbf{0}, \mathbf{m}_2 - \mathbf{m}_1 * \mathbf{h})^T$. This is sufficient, because we can add the lattice vector $(\mathbf{m}_1, \mathbf{m}_1 * \mathbf{h})^T$ to the solution of our modified problem and solve the original

APPR-CVP. Therefore, we define $\mathbf{m} := \mathbf{m}_2 - \mathbf{m}_1 * \mathbf{h}$ and demonstrate the solution for $(\mathbf{0}, \mathbf{m})^T$.

We calculate our solution using the inverting and rounding technique again. The exact solution in $\mathcal{Q}$ is given by $(\mathbf{x}, \mathbf{y})^T$ as follows:

$$\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} := (\mathcal{B}_{\mathsf{pri}})^{-1} \begin{pmatrix} \mathbf{0} \\ \mathbf{m} \end{pmatrix} = \begin{pmatrix} \mathbf{f} & \mathbf{F} \\ \mathbf{g} & \mathbf{G} \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{0} \\ \mathbf{m} \end{pmatrix} = \begin{pmatrix} \frac{-\mathbf{F}*\mathbf{m}}{q} \\ \frac{\mathbf{f}*\mathbf{m}}{q} \end{pmatrix}.$$

Using this $(\mathbf{x}, \mathbf{y})$, we can again round off to get $(\tilde{\mathbf{s}}, \tilde{\mathbf{t}})$

$$\begin{aligned} \begin{pmatrix} \tilde{\mathbf{s}} \\ \tilde{\mathbf{t}} \end{pmatrix} &:= \mathcal{B}_{\mathsf{pri}} \left\lfloor (\mathcal{B}_{\mathsf{pri}})^{-1} \begin{pmatrix} \mathbf{0} \\ \mathbf{m} \end{pmatrix} \right\rceil = \begin{pmatrix} \mathbf{f} & \mathbf{F} \\ \mathbf{g} & \mathbf{G} \end{pmatrix} \cdot \left\lfloor \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \right\rceil \\ &= \left\lfloor \tfrac{-\mathbf{F}*\mathbf{m}}{q} \right\rceil \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix} + \left\lfloor \tfrac{\mathbf{f}*\mathbf{m}}{q} \right\rceil \begin{pmatrix} \mathbf{F} \\ \mathbf{G} \end{pmatrix}, \end{aligned}$$

and see how close this APPR-CVP solution is to the desired point:

$$\begin{pmatrix} \tilde{\mathbf{s}} \\ \tilde{\mathbf{t}} \end{pmatrix} - \begin{pmatrix} \mathbf{0} \\ \mathbf{m} \end{pmatrix} = \left(\tfrac{-\mathbf{F}*\mathbf{m}}{q}\right)' \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix} + \left(\tfrac{\mathbf{f}*\mathbf{m}}{q}\right)' \begin{pmatrix} \mathbf{F} \\ \mathbf{G} \end{pmatrix}.$$

The final signature $(\mathbf{s}, \mathbf{t})^T$ is obtained by adding $(\mathbf{m}_1, \mathbf{m}_1 * \mathbf{h})^T$. Notice that $\mathbf{s}$ suffices as the signature, because $\mathbf{t} = \mathbf{s} * \mathbf{h} \mod q$. So in practice, we only calculate $\mathbf{s}$.

$$(\textit{standard} \text{ case}) \ \mathbf{s} := \left(\tfrac{-\mathbf{F}*\mathbf{m}}{q}\right)' * \mathbf{f} + \left(\tfrac{\mathbf{f}*\mathbf{m}}{q}\right)' * \mathbf{F} + \mathbf{m}_1 \mod q$$

We did use the *standard* and not the *transpose* lattice in this calculation. Using the transpose lattice with $\mathcal{B}^T_{\mathsf{pri}}$ instead yields the formula

$$(\textit{transpose} \text{ case}) \ \mathbf{s} := \left(\tfrac{-\mathbf{g}*\mathbf{m}}{q}\right)' * \mathbf{f} + \left(\tfrac{\mathbf{f}*\mathbf{m}}{q}\right)' * \mathbf{g} + \mathbf{m}_1 \mod q.$$

As mentioned before, we only have multiplications with the trinary (or binary) polynomials $\mathbf{f}, \mathbf{g}$ in the transpose case, which makes calculating the signature very fast.

Verifying. Verifying takes two more public parameters.

$\mathcal{N} \in \mathbb{R}$	*Norm bound*	Bound for valid signatures.
$\beta \in \mathbb{R}$	*Scaling constant*	Makes forgery harder.

To verify the validity of a signature $\mathbf{s}$, we check whether it solves the $\mathcal{N}$-APPR-CVP of the corresponding document hash $(\mathbf{m}_1, \mathbf{m}_2)$:

$$\begin{aligned} & \left\| \begin{pmatrix} \mathbf{m}_1 \\ \mathbf{m}_2 \end{pmatrix} - \begin{pmatrix} \mathbf{s} \\ \mathbf{s} * \mathbf{h} \end{pmatrix} \right\|_\beta^2 \\ = & \left\| \left(\begin{pmatrix} \mathbf{0} \\ \mathbf{m} \end{pmatrix} + \begin{pmatrix} \mathbf{m}_1 \\ \mathbf{m}_1 * \mathbf{h} \end{pmatrix} \right) - \left(\begin{pmatrix} \tilde{\mathbf{s}} \\ \tilde{\mathbf{s}} * \mathbf{h} \end{pmatrix} + \begin{pmatrix} \mathbf{m}_1 \\ \mathbf{m}_1 * \mathbf{h} \end{pmatrix} \right) \right\|_\beta^2 \\ = & \left\| \begin{pmatrix} \tilde{\mathbf{s}} \\ \mathbf{m} - \tilde{\mathbf{s}} * \mathbf{h} \end{pmatrix} \right\|_\beta^2 \leq \mathcal{N}^2. \end{aligned}$$

We have seen before that $\mathbf{s}$ is exactly as good a signature of $(\mathbf{m}_1, \mathbf{m}_2)$, as $\tilde{\mathbf{s}}$ is one for $(\mathbf{0}, \mathbf{m})$.

The scaling constant β and the way it should be used will be explained as part of the defence against forgery attacks in section 3.3 and 3.4. For now, we want to find a sensible normbound $\mathcal{N}$.

To define a sensible $\mathcal{N}$, we look at the *expected* size $\mathcal{E}$ of $\| (\tilde{\mathbf{s}}, \mathbf{m} - \tilde{\mathbf{s}} * \mathbf{h}) \|_\beta$, when $\tilde{\mathbf{s}}$ is calculated with a proper private basis. We split up $\mathcal{E}$ into two separate parts, $\mathcal{E}_\mathbf{s}$ and $\mathcal{E}_\mathbf{t}$:

$$\begin{aligned}\mathcal{E}^2 &= \left\| \begin{pmatrix} \tilde{\mathbf{s}} \\ \mathbf{m} - \tilde{\mathbf{s}} * \mathbf{h} \end{pmatrix} \right\|_\beta^2 \\ &= \underbrace{\| \tilde{\mathbf{s}} \|_q^2}_{=:\mathcal{E}_\mathbf{s}^2} + \beta^2 \underbrace{\| \mathbf{m} - \tilde{\mathbf{s}} * \mathbf{h} \|_q^2}_{=:\mathcal{E}_\mathbf{t}^2}.\end{aligned}$$

In the case of a *standard* private basis we get:

$$\begin{aligned}\mathcal{E}_\mathbf{s} &= \left\| \left\lfloor \tfrac{-\mathbf{F}*\mathbf{m}}{q} \right\rceil * \mathbf{f} + \left\lfloor \tfrac{\mathbf{f}*\mathbf{m}}{q} \right\rceil * \mathbf{F} \right\|_q \\ &\leq \left\| \left(\tfrac{-\mathbf{F}*\mathbf{m}}{q} \right)' * \mathbf{f} \right\|_\mu + \left\| \left(\tfrac{\mathbf{f}*\mathbf{m}}{q} \right)' * \mathbf{F} \right\|_\mu \\ &\approx \sqrt{\tfrac{N}{12}} \left(\| \mathbf{f} \|_\mu + \| \mathbf{F} \|_\mu \right).\end{aligned}$$

And similarly for $\mathcal{E}_\mathbf{t}$:

$$\mathcal{E}_\mathbf{t} = \left\| \mathbf{m} - \left\lfloor \tfrac{-\mathbf{F}*\mathbf{m}}{q} \right\rceil * \mathbf{g} + \left\lfloor \tfrac{\mathbf{f}*\mathbf{m}}{q} \right\rceil * \mathbf{G} \right\|_q \approx \sqrt{\tfrac{N}{12}} \left(\| \mathbf{g} \|_\mu + \| \mathbf{G} \|_\mu \right).$$

Analogously, for a *transpose* private basis we get:

$$\mathcal{E}_\mathbf{s} \approx \sqrt{\tfrac{N}{12}} \left(\| \mathbf{f} \|_\mu + \| \mathbf{g} \|_\mu \right) \qquad \mathcal{E}_\mathbf{t} \approx \sqrt{\tfrac{N}{12}} \left(\| \mathbf{F} \|_\mu + \| \mathbf{G} \|_\mu \right).$$

Using the different sizes of $\| \mathbf{f} \|_\mu$,$\| \mathbf{g} \|_\mu$,$\| \mathbf{F} \|_\mu$ and $\| \mathbf{G} \|_\mu$, which depended on whether we use binary or trinary keys, we may now calculate the correct $\mathcal{E}$.

We built $\mathcal{E}$ such that all validly generated signatures are expected to fulfil:

$$\left\| \begin{pmatrix} \mathbf{m}_1 \\ \mathbf{m}_2 \end{pmatrix} - \begin{pmatrix} \mathbf{s} \\ \mathbf{s} * \mathbf{h} \end{pmatrix} \right\|_\beta^2 \approx \mathcal{E}^2.$$

Using $\mathcal{E}$ we will define $\mathcal{N}$ with the help of another parameter, that states how tolerant we are with signing failures.

$\rho \in \mathbb{R}$	*Tolerance*	$\rho = 1.1$ means few failures

We set

$$\mathcal{N} := \rho\mathcal{E}.$$

This bound is easily reachable for a valid signer and hard to get to for a forger. But to be on the safe side we recommended that the signer checks his own signature to see if the bound was not too small. Should this be the case, he can append a random quantity to the document and resign it.

For example, in the case of *trinary* keys and the *transpose* lattice, where $d_{\mathbf{f}} = d_{\mathbf{g}} = d$, we get the following $\mathcal{N}$ and $\mathcal{E}_{\mathbf{s}}/\mathcal{E}_{\mathbf{t}}$:

$$\| \mathbf{f} \|_\mu^2 = \| \mathbf{g} \|_\mu^2 = N\delta(d) \qquad \| \mathbf{F} \|_\mu^2 \approx \| \mathbf{G} \|_\mu^2 \approx N^2\delta(d)/12.$$

By the previous observations, this gives us:

$$\begin{aligned}
\mathcal{N} &= \rho\sqrt{\mathcal{E}_{\mathbf{s}}^2 + \beta^2\mathcal{E}_{\mathbf{t}}^2} & \frac{\mathcal{E}_{\mathbf{s}}}{\mathcal{E}_{\mathbf{t}}} &= \frac{2\| \mathbf{f} \|_\mu}{\| \mathbf{F} \|_\mu + \| \mathbf{G} \|_\mu} \\
&= \rho\sqrt{\tfrac{N}{12}\,(\,2\,\| \mathbf{f} \|_\mu^2 + \beta^2(\| \mathbf{F} \|_\mu^2 + \| \mathbf{G} \|_\mu^2)\,)} & &\approx \sqrt{\tfrac{N\delta(d)\,12}{N^2\delta(d)}} \\
&\approx \rho N\sqrt{\tfrac{\delta(d)}{3}\,(1 + \beta^2\tfrac{N}{12})} & &= \sqrt{\tfrac{12}{N}} \\
&= \tfrac{\rho N}{6}\sqrt{\delta(d)\,(12 + \beta^2 N)}
\end{aligned}$$

2.4.4 Perturbations

To minimise the amount of information that is leaked by a transcript of signatures, we recommend to use NTRUSign with at least one perturbation. Craig Gentry and Mike Szydlo showed in [15] that with enough transcripts to mount a second order attack, they can reveal the matrix BB^T, where B is the private basis. A fourth order attack even reveals B itself. NTRU tells us in [20] that according to experiments for the standard case, one would need about 10 thousand transcripts for second and more than 100 million transcripts for fourth order information.

It is an open problem whether the knowledge of BB^T can significantly help the attacker. So until this is answered in the affirmative, the lack of perturbations is not dangerous when the standard basis is used. In the transpose case however, experiments have shown that less transcripts are needed for second and fourth order information, so the use of perturbations is recommended.

The basic idea of perturbations is to perturb the message hash $(\mathbf{0}, \mathbf{m})$ prior to signing and solve the appr-CVP of the perturbed message. If the bound $\mathcal{N}$ is big enough, the signature of the perturbed message will also solve the

original $\mathcal{N}$-appr-CVP. In the case of NTRUSign, we use many lattices instead of just one to perform the perturbation.

In order to use NTRUSign with b perturbations, we need to create the public and private bases for b different lattices with the same NTRUSign-parameters. We will number these bases $1, \ldots, b$. In the transpose case, this will be:

$$\begin{pmatrix} \mathbf{e} & \mathbf{0} \\ \mathbf{h}_1 & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{f}_1 & \mathbf{g}_1 \\ \mathbf{F}_1 & \mathbf{G}_1 \end{pmatrix}, \cdots, \begin{pmatrix} \mathbf{e} & \mathbf{0} \\ \mathbf{h}_b & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{f}_b & \mathbf{g}_b \\ \mathbf{F}_b & \mathbf{G}_b \end{pmatrix}.$$

We start to sign a message $(0, \mathbf{m})$ by solving its appr-CVP in $L(\mathbf{h}_1, q)$, resulting in $\mathbf{s}_1$:

$$\mathbf{s}_1 = \left(\frac{-\mathbf{g}_1 * \mathbf{m}}{q}\right)' * \mathbf{f}_1 + \left(\frac{\mathbf{f}_1 * \mathbf{m}}{q}\right)' * \mathbf{g}_1 \mod q.$$

This is our perturbed message which we sign again in lattice 2. We solve the appr-CVP for $(\mathbf{s}_1, \mathbf{s}_1 * \mathbf{h}_1)^T$ in $L(\mathbf{h}_2, q)$ resulting in $\mathbf{s}_2$.

$$\mathbf{s}_2 = \mathbf{s}_1 + \left(\frac{-\mathbf{g}_2 * \mathbf{s}_1 * (\mathbf{h}_1 - \mathbf{h}_2)}{q}\right)' * \mathbf{f}_2 + \left(\frac{\mathbf{f}_2 * \mathbf{s}_1 * (\mathbf{h}_1 - \mathbf{h}_2)}{q}\right)' * \mathbf{g}_2 \mod q$$

If we had wanted just one perturbation, we could stop at this point. On the other hand, we may say that $\mathbf{s}_2$ is the perturbed message point and solve the appr-CVP for $(\mathbf{s}_2, \mathbf{s}_2 * \mathbf{h}_2)^T$ in $L(\mathbf{h}_3, q)$.

We continue to do this until we solve $(\mathbf{s}_{b-1} * (\mathbf{h}_{b-1} - \mathbf{h}_b))^T$ in $L(\mathbf{h}_b, q)$ with $\mathbf{s}_b$:

$$\begin{aligned} \mathbf{s}_b = \mathbf{s}_1 &+ \sum_{i=2}^{b} \left(\frac{-\mathbf{g}_i * \mathbf{s}_{i-1} * (\mathbf{h}_{i-1} - \mathbf{h}_i)}{q}\right)' * \mathbf{f}_i \\ &+ \sum_{i=2}^{b} \left(\frac{\mathbf{f}_i * \mathbf{s}_{i-1} * (\mathbf{h}_{i-1} - \mathbf{h}_i)}{q}\right)' * \mathbf{g}_i \mod q. \end{aligned}$$

So our final signature is $(\mathbf{s}_b, \mathbf{s}_b * \mathbf{h}_b)^T$, and only $L(\mathbf{h}_b, q)$ is the public basis. All the other ones are secret.

Even if we use just one perturbation in the weaker setting of transpose lattices, NTRU tells us that finding the matrix BB^T requires a sixth-order attack, with at least 10^{18} signatures. Using perturbations does increase the time for signing, because multiple appr-CVPs have to be solved. It also increases the expected size $\mathcal{E}$ of a validly generated signature, and thus also the normbound $\mathcal{N}$ we would wish to use.

In the most significant case of one perturbation for a transpose lattice, we get the following $\mathcal{E}$:

$$\begin{aligned}
\mathcal{E} &= \sqrt{\mathcal{E}_{\mathbf{s}}^2 + \beta^2 \mathcal{E}_{\mathbf{t}}^2} \\
\mathcal{E}_{\mathbf{s}} &= \sqrt{\tfrac{N}{12}} \left(\| \mathbf{f}_1 \|_\mu + \| \mathbf{g}_1 \|_\mu + \| \mathbf{f}_2 \|_\mu + \| \mathbf{g}_2 \|_\mu \right) \\
\mathcal{E}_{\mathbf{t}} &= \sqrt{\tfrac{N}{12}} \left(\| \mathbf{F}_1 \|_\mu + \| \mathbf{G}_1 \|_\mu + \| \mathbf{F}_2 \|_\mu + \| \mathbf{G}_2 \|_\mu \right).
\end{aligned}$$

To get the proper values, we would still have to insert the key length. We can see at this point that the increase is not much though, since both keys are small.

Perturbations are only good to protect against transcript averaging attacks. They do not provide any protection against lattice based attacks or combinatorial attacks. These will be explained in section 3. All these attacks work directly with the *public* part of an NTRUSign-system. In the case of perturbation, these are the NTRUSign-parameters and the last public key $\mathbf{h}_b$.

3 Attacks on NTRUSign

There are two main attacks on NTRUSign. The combinatorial attack and the lattice-based attack. Both of them can be used either to find out the private key directly or to forge a valid signature. This section is therefore divided into four subsections where each of these attacks is explained.

For this section, we will assume that NTRUSign is used with trinary keys and the transpose lattice, since this seems to be the most secure version (compare 2.4). In each subsection, we will derive the bit-security ω of an NTRUSign instance using a given parameter set against the particular attack explained there.

3.1 Combinatorial Attack

A combinatorial attack is realised using a meet-in-the-middle technique on the known private key space $\mathcal{T}(d)$. For the binary case see [29].

Recall that the elements of $\mathcal{T}(d)$ are polynomials of degree N, which have $d+1$ coefficients equal to 1 and d coefficients equal to -1. One of these elements is the private key $\mathbf{f}$. We know that $\mathbf{f}$ can be written as $\mathbf{f} = \mathbf{f}_1 - \mathbf{f}_{-1}$. Here, $\mathbf{f}_1$ is the polynomial which has those $(d+1)$ coefficients equal to 1 that are 1 for $\mathbf{f}$, but has zero coefficients otherwise. Analogously, $\mathbf{f}_{-1}$ has d coefficients equal to 1, where $\mathbf{f}$ has -1, and has zero coefficients in all other places. So both parts of $\mathbf{f}$ are binary: $\mathbf{f}_1 \in \mathcal{B}(d+1)$ and $\mathbf{f}_{-1} \in \mathcal{B}(d)$.

The target of this attack will be the set of all rotations (see section 1.1) of the private key $\mathbf{f}$, $R_\mathbf{f} := \{\, \rho^k(\mathbf{f}) \mid k \in \mathbb{N} \,\}$. We consider the attack successful if one of the elements of $R_\mathbf{f}$ is recovered from the public parameters. We know that $\rho^N(\mathbf{f}) = \mathbf{f}$, so $|R_\mathbf{f}| \leq N$. Also $\rho(\mathbf{f}) \neq \mathbf{f}$, since $\mathbf{f}$ is not 0 or -1, so $|R_\mathbf{f}| > 1$. But since N is prime, this means that $|R_\mathbf{f}| = N$, because otherwise it would be a divisor of N. So we have exactly N different targets. By the same argument, $R_{\mathbf{f}_1}$ and $R_{\mathbf{f}_{-1}}$ have N distinct elements as well.

We use a hash function $\theta : \mathcal{R}_q \to \{0,1\}^N$, which maps each polynomial to a list containing the top bit of *each* coefficient. By this we mean the top bit of the $\log_2(q)$ possible bits that each coefficient has. We also create a hash-table of size 2^N with a place for every image of θ.

For any element in $R_\mathbf{f}$, say $\rho^k(\mathbf{f})$, we know this:

$$\begin{aligned} \rho^k(\mathbf{f}_1 - \mathbf{f}_{-1}) * \mathbf{h} &\equiv \rho^k(\mathbf{F}) \mod q \\ \rho^k(\mathbf{f}_1) * \mathbf{h} &\equiv \rho^k(\mathbf{f}_{-1}) * \mathbf{h} + \rho^k(\mathbf{F}) \mod q \\ \theta(\rho^k(\mathbf{f}_1) * \mathbf{h}) &= \theta(\rho^k(\mathbf{f}_{-1}) * \mathbf{h} + \rho^k(\mathbf{F})). \end{aligned}$$

So we are motivated to find candidates $\mathbf{c}_1$ and $\mathbf{c}_{-1}$ such that

$$\theta(\mathbf{c}_1 * \mathbf{h}) = \theta(\mathbf{c}_{-1} * \mathbf{h} + \rho^k(\mathbf{F})).$$

After picking two candidates $\mathbf{c}_1 \in_R \mathcal{B}(d+1)$ and $\mathbf{c}_{-1} \in_R \mathcal{B}(d)$, we save $\mathbf{c}_1$ at the spot $\theta(\mathbf{c}_1 * \mathbf{h})$ in our hash-table. We cannot save $\mathbf{c}_{-1}$ at $\theta(\mathbf{c}_{-1} * \mathbf{h} + \rho^k(\mathbf{F}))$, since we don't know $\mathbf{F}$ or k. So we assume there is a way to estimate the largest coefficients of $\mathbf{F}$ (this is not necessary in the non-transpose case, since $\mathbf{F}$ is binary then). We call this estimate F_{max}. We calculate all the other possible hashes, which can be obtained by adding F_{max} or 0 to each coefficient of $\mathbf{c}_{-1} * \mathbf{h}$ and we save $\mathbf{c}_{-1}$ in all those places as well. Since F_{max} is very small, compared to q, there will not be many top bits changed by adding it. So the number of possible hashes we will find this way is small.

If we save a $\mathbf{c}_1$ candidate and a $\mathbf{c}_{-1}$ candidate to the same spot, we check whether $\| (\mathbf{c}_1 - \mathbf{c}_{-1}) * \mathbf{h} \|_\mu \approx \| \mathbf{F} \|_\mu$. We do this using the approximation from section 2.4.2. If the check comes back positive, we have found a rotation of the private key, and if not we go on searching. We might get a rotation of $\mathbf{g}$ instead of $\mathbf{f}$ at this point, because $\mathbf{g}$ splits into binary polynomials exactly like $\mathbf{f}$, and $\| \mathbf{g} * \mathbf{h} \|_\mu \approx \| \mathbf{F} \|_\mu$. But in that case, we may easily recover $\mathbf{f}$ by lattice reduction techniques (see 3.2).

In fact, this is an unnecessary advantage for the attacker. In NTRUSIGN-parametersets using trinary keys, NTRU uses only one d instead of different $d_{\mathbf{f}}$ and $d_{\mathbf{g}}$. But that makes the attack we have just described more likely to succeed, because it has $2N$ targets instead of just N. Since this is easily remedied by taking two ds, we will omit this advantage in the rest of the analysis.

Using a hash-table, we have ensured that sorting and comparing happen instantly when we save a candidate into its spot. So the main work is done in picking the candidates.

The set containing all possible candidates $\mathbf{c}_1$ for $\mathbf{f}_1$ is $\mathcal{B}(d+1)$. Only N of these are rotations of $\mathbf{f}_1$. After randomly picking $\sqrt{N}(\,|\mathcal{B}(d+1)|/N\,)$ candidates, we hope to have picked $\sqrt{N}$ different rotations of $\mathbf{f}_1$. We pick the same amount of $\mathbf{f}_{-1}$ candidates from $\mathcal{B}(d)$ and again hope to get $\sqrt{N}$ different rotations.

Let us find out how good our chances are to find enough rotations, such that one of $\mathbf{f}_1$ actually fits to one of $\mathbf{f}_{-1}$. In order to see this, we will need the probability of certain events.

First we draw candidates from $\mathcal{B}(d+1)$. Our chances of getting a rotation of $\mathbf{f}_1$ on one draw is $p_1 = N/|\mathcal{B}(d+1)|$. The number of rotations we get after $t = \lfloor \sqrt{N}/p_1 \rfloor$ draws (with replacement) can be described by a binomially distributed random variable $X_1 \sim B(t, p_1)$. From $\mathcal{B}(d)$, we draw t the same

number of times, but our chances to get a rotation are slightly better, namely $p_2 = N/|\mathcal{B}(d)|$. The number of rotations we find drawing candidates for $\mathbf{f}_{-1}$ can be described by $X_2 \sim B(t, p_2)$.

Since the numbers involved are so big, we will not be able to calculate $\mathrm{P}(X_1 = A)$ effectively. We recommend to use the normal distribution $X_1' \sim N(tp_1, \sqrt{tp_1(1-p_1)})$ with $tp_1 = \sqrt{N}$ instead.

$$\mathrm{P}(X_1 = A) \approx \mathrm{P}(X_1' \leq A + 1/2) - \mathrm{P}(X_1' \leq A - 1/2)$$

The variables X_1, X_2 are independent, so the chance of getting A rotations of $\mathbf{f}_1$ and B rotations of $\mathbf{f}_{-1}$ is $\mathrm{P}(X_1 = A)\ \mathrm{P}(X_2 = B)$.

We will assume now that we have not only found these rotations, but that we found X_1 respectively X_2 different ones. We let $C_1 \subset R_{\mathbf{f}_1}$ be the set of all candidates which are rotations of $\mathbf{f}_1$. It has X_1 elements, just like the closely related set $C_1' := \{\, k \mid \rho^k(\mathbf{f}_1) \in C_1 \,\}$. An analogously defined C_{-1}' has X_2 elements. When these two sets intersect, our attack was successful. We think of the elements in C_1' as special elements of the set $S = \{0, \ldots, N-1\}$. Drawing B times from S without replacement, we let Y be the number of special elements we find.

The number of matching rotations we find can be described by the hypergeometrically distributed random variable $Y \sim H(|S|, X_1, X_2)$:

$$\mathrm{P}(Y \geq 1 | X_1 = A, X_2 = B) = 1 - \prod_{i=0}^{B-1} \left(1 - \frac{A}{N-i}\right).$$

Our chance of success is $\mathrm{P}(Y \geq 1)$:

$$\mathrm{P}(Y \geq 1) = \sum_{1 \leq A, B \leq N} \mathrm{P}(Y \geq 1 | X_1 = A, X_2 = B)\, \mathrm{P}(X_1 = A)\, \mathrm{P}(X_2 = B).$$

For the 80-bit secure parameters suggested in [21], $N = 157$ and $d = 29$, this works out to be $\mathrm{P}(\text{`success'}) = 0.6087$.

So in summation, if we perform $|\mathcal{B}(d+1)|/\sqrt{N}$ steps of picking candidates for $\mathbf{f}_1$ and $\mathbf{f}_{-1}$ and saving them in a hash-table, we have a good chance to succeed in finding a rotation of the private key. So picking this many elements should take long. The security against this attack is given by

$$\omega_{\mathsf{cmb}}(N, d) := \log_2 \left(\frac{\binom{N}{d+1}}{\sqrt{N}} \right).$$

3.2 Lattice-Based Attack

Now we will focus on the lattice based attacks on NTRUSIGN. The basic idea is to use a modification of the LLL-algorithm by Lenstra, Lenstra and Lovász [39] to find short vectors in the NTRU lattice. If the vectors we recover are short enough, then a rotation of the private key $(\mathbf{f}, \mathbf{g})^T$ / $(\mathbf{F}, \mathbf{G})^T$ (or, in the transpose case, $(\mathbf{f}, \mathbf{F})^T$ / $(\mathbf{g}, \mathbf{G})^T$) is among them and the attack is successful. Zero-forcing [42] can be used to reduce the dimension of the NTRU lattice, before the actual algorithmic attack starts.

This attack will be explained in detail, and we will find out the bit strength ω_{lk} against a lattice key attack for a given NTRUSIGN variable set (N, q, d). We start with a couple of observations.

NTRU has experimented a lot with LLL-type attacks on convolution modular lattices in general and NTRU lattices in particular. They have come up with the following result. When two ratios a, c are approximately constant, then the runtime T of a successful attack grows exponentially in N. The ratios are

$$a = \frac{N}{q} \qquad c = \sqrt{2N}\,\frac{\text{length of shortest vector}}{\text{estimated length of shortest vector}}.$$

To understand what the second ratio c represents, we need to know a heuristic made long ago by Gauss. He stated that in a random lattice L, there are few vectors shorter than his estimate

$$\lambda_1(L) \approx \sigma(L) := \sqrt{\frac{\dim(L)}{2\pi e}}\,\det(L)^{\frac{1}{\dim(L)}}.$$

In the case of convolution modular lattices like $L(\mathbf{h}, q)$, we remember that $\dim(L) = 2N$ and $\det(L) = (\lambda q)^N$, for some balancing constant λ (compare 1.1). So for balanced CMLs, the Gaussian estimate of the shortest vector is

$$\sigma(L) = \sqrt{\frac{N\lambda q}{\pi e}}.$$

The shortest vector used to calculate c is of course the private key $(\mathbf{p}_1, \mathbf{p}_2)^T$. An attacker must estimate the length of the private key using only the public parameters N, q, d. The attacker also wants to balance the lattice before reduction starts (refer to section 1.1). We will see that balancing can decrease c, leading to a faster reduction.

In order to get the optimal use out of our balancing constant λ, we want to choose it such that c is as small as possible. It turns out, that this corresponds to balancing the shortest vector. To find the optimal λ, we use

it to minimise the function f representing the ratio between the balanced private key and the Gauss estimate:

$$f(\lambda) := \frac{\left\| (\lambda \mathbf{p}_1, \mathbf{p}_2)^T \right\|_L}{\sigma(L)} = \frac{\| \text{ shortest vector in the balanced lattice } \|}{\text{Gauss estimate in balanced lattice}}.$$

Actually, we could put different λs in front of both parts—$(\lambda_1 \mathbf{p}_1, \lambda_2 \mathbf{p}_2)$—before starting our reduction, in order to get c even smaller. But it turns out that scaling the whole lattice and thus any vector by a factor μ does not change the ratio that f represents, and so we could not hope to get a smaller c this way:

$$\begin{pmatrix} \lambda_1 \mathbf{p}_1 \\ \lambda_2 \mathbf{p}_2 \end{pmatrix} = \lambda_2 \begin{pmatrix} \lambda_2^{-1} \lambda_1 \mathbf{p}_1 \\ \mathbf{p}_2 \end{pmatrix}.$$

The general case is just a scaling by $\mu = \lambda_2$ of the case which we look at with our definition of f, meaning that we will find the minimal possible c this way.

Since the private key is different in the standard and transpose case, we will separate these two cases.

Standard case. In accordance with [42], we will see that NTRUSIGN is more vulnerable to lattice-based attacks in the standard case. As we might suspect, balancing makes no difference in the standard case, since we have that $\| \mathbf{f} \|_q \approx \| \mathbf{g} \|_q$ anyway. The balanced secret key is $(\lambda \mathbf{p}_1, \mathbf{p}_2) = (\lambda \mathbf{f}, \mathbf{g})$.

$$\begin{aligned} f_{\mathsf{Std}}(\lambda) &:= \frac{\left\| (\lambda \mathbf{f}, \mathbf{g})^T \right\|_L}{\sigma(L)} \\ &= \sqrt{(\lambda^2 \| \mathbf{f} \|_q^2 + \| \mathbf{g} \|_q^2) \, \frac{\pi e}{N \lambda q}} \approx \sqrt{(\lambda^2 + 1) \, \delta(d) N \, \frac{\pi e}{N \lambda q}} \end{aligned}$$

Differentiation yields that $\lambda_{\mathsf{Std}} = \| \mathbf{g} \|_q \, / \, \| \mathbf{f} \|_q \approx 1$ minimises f_{Std}, giving us

$$f_{\mathsf{Std}}(1) \approx \sqrt{2\delta(d) \, \frac{\pi e}{q}} \qquad c_{\mathsf{Std}} = \sqrt{2N} \; f_{\mathsf{Std}}(1) \approx 2\sqrt{N\delta(d) \frac{\pi e}{q}}.$$

Transpose case. In the transpose case, balancing does indeed make a difference. We will look at $(\lambda \mathbf{p}_1, \mathbf{p}_2)^T = (\lambda \mathbf{f}, \mathbf{F})^T$:

$$\begin{aligned} f_{\mathsf{Trp}}(\lambda) &:= \frac{\left\| (\lambda \mathbf{f}, \mathbf{F})^T \right\|_L}{\sigma(L)} \\ &= \sqrt{(\lambda^2 \| \mathbf{f} \|_q^2 + \| \mathbf{F} \|_q^2) \, \frac{\pi e}{N \lambda q}} \approx \sqrt{(\lambda^2 \delta(d) N + \frac{\delta(d) N^2}{12}) \, \frac{\pi e}{N \lambda q}}. \end{aligned}$$

Again, we find the minimum using differentiation. And this minimum is again the λ which balances the shortest vector:

$$\lambda_{\mathsf{Trp}} = \frac{\| \mathbf{F} \|_q}{\| \mathbf{f} \|_q} \approx \sqrt{\frac{\delta(d)N^2}{12} \frac{1}{\delta(d)N}} = \sqrt{\frac{N}{12}}.$$

The resulting f_{Trp} and most importantly c_{Trp} are

$$f_{\mathsf{Trp}}(\lambda_{\mathsf{Trp}}) \approx \sqrt{\frac{\delta(d)\ \pi e\sqrt{N}}{q\sqrt{3}}}$$

$$c_{\mathsf{Trp}} = \sqrt{2N}\ f_{\mathsf{Trp}}(\lambda_{\mathsf{Trp}}) \approx \sqrt{\frac{2\delta(d)\pi e\ N^{\frac{3}{2}}}{q\sqrt{3}}}\ .$$

Notice that for $N > 12$, the ratio

$$\frac{c_{\mathsf{Trp}}}{c_{\mathsf{Std}}} = \sqrt{\frac{\sqrt{N}}{2\sqrt{3}}}$$

becomes greater than 1. All interesting parameter sets will have $N > 12$, so the standard case is more vulnerable to lattice-based attacks.

We may now calculate our ratio pair (a, c) using only the public parameters. According to NTRU's heuristic, the logarithm of the runtime T for LLL-type attacks is linear in N. So for our ratio pair (a, c), there are constants A and B such that

$$\log_2(T) = AN + B.$$

NTRU has calculated a lot of these constants A, B, and they are given as a table in [23] or [28]. We could stop right here and say that the bit strength of a transpose NTRU lattice against this attack is $\omega_{\mathsf{lk}}(A, B, N) := AN + B$—were it not for the work presented in [42], which has introduced zero-forcing.

3.2.1 Zero-Forcing

From here on we will focus on a setting with trinary keys and the transpose lattice. The case standard case with binary keys is described in [42]. Zero-forcing works as follows: We try to guess a pattern of zeros (a list of zero-positions), which is fitting for the key vectors $\mathbf{f}$ or $\mathbf{g}$ or one of their rotations. We use this pattern to cross out the corresponding rows of the NTRU-lattice, thus reducing the dimension. Lattice reduction works much faster on smaller lattices, and if we guessed a correct pattern, the secret key is still a small vector in the reduced lattice, which can be recovered by lattice reduction.

Let r be the actual numbers of zero-positions that we wish to guess. We look at the probability for a randomly selected trinary $\mathbf{f} \in_R \mathcal{T}(d)$ to fit our pattern. That amounts to $d+1$ ones and d minus ones missing the r zero-spots in our pattern. The number of zero-positions which are hit when we pick $2d+1$ times without replacement is described by the hypergeometric random variable $X \sim H(N, r, 2d+1)$:

$$\mathrm{P}(X=0) = \prod_{i=0}^{2d} \left(1 - \frac{r}{N-i}\right).$$

A secret key remains in the reduced lattice, if any one vector amongst the N rotations of $\mathbf{f}$ or the N rotations of $\mathbf{g}$ fits the pattern. These rotations of the secret keys are our target vectors. We let Y be the number of target vectors which fit the pattern. Then we have $Y \sim B(N, P(X=0))$, giving us:

$$\mathrm{P}(Y \geq 1) = 1 - \mathrm{P}(X=0)^{2N} = 1 - \left(1 - \prod_{i=0}^{2d} \left(1 - \frac{r}{N-i}\right)\right)^{2N}.$$

We actually ignore some targets at this point. Imagine $\mathbf{f}+\mathbf{g}$ or $\mathbf{f}-\mathbf{g}$ fit our pattern, while all of the rotations of $\mathbf{f}$ or $\mathbf{g}$ alone miss it. In this scenario the attacker will easily break the lattice by adding his zeros back to the vector and then reducing the public basis with the vector he found. However, since none of the rotations of $\mathbf{f}$ and $\mathbf{g}$ fit, $Y=0$ so this case is not counted as a success for the attacker.

This means the actual chances of the attacker are better than we make them, but we imagine the effect is slight. Due to a lack of time, this effect has not been investigated by us yet.

Let us see how an attacker who does not use zero-forcing compares to one who does. We already said that the normal runtime of the attack is

$$T_{\mathsf{normal}} = 2^{AN+B}$$

for some constants A, B depending on a, c.

We call the zero-forced lattice L_{ZF}. It is the old lattice L, but in the upper half of it, r rows corresponding to the attacker's pattern are deleted. We call the key vectors where the same r entries are deleted $\mathbf{f}_{\mathsf{ZF}}$ and $\mathbf{g}_{\mathsf{ZF}}$. In the upper half of the zero-forced lattice, we will have to use a different μ-norm where our vectors only have $N-r$ entries instead of N. We denote this norm by $\|\cdot\|_{\mu,N-r}$. The corresponding L-norm on L_{ZF} will be denoted $\|\cdot\|_{L,(N-r)}$. The following changes occur:

$$\dim(L_{\mathsf{ZF}}) = 2N - r \quad \det(L_{\mathsf{ZF}}) = \lambda^{N-r} q^N \quad \| \mathbf{f}_{\mathsf{ZF}} \|_{\mu,N-r} = \sqrt{\tfrac{2d}{N-r}}.$$

Using these, we may calculate the Gaussian estimate and the length of the shortest vector in L_{ZF}:

$$\begin{aligned}\sigma(L_{\mathsf{ZF}}) &= \sqrt{\tfrac{2N-r}{2\pi e}}\left(q^N\lambda^{N-r}\right)^{\frac{1}{2N-r}}\\ \left\|\,(\lambda\mathbf{f}_{\mathsf{ZF}},\mathbf{F})^T\,\right\|_{L,(N-r)} &= \sqrt{\lambda^2\left\|\,\mathbf{f}_{\mathsf{ZF}}\,\right\|^2_{\mu,(N-r)}+\left\|\,\mathbf{F}\,\right\|^2_{\mu}}\\ &= \sqrt{\lambda^2\tfrac{2d}{N-r}+N^2\tfrac{\delta(d)}{12}}.\end{aligned}$$

With these two formulas, we can use differentiation like before to find the λ minimising the ratio ($\|$ shortest vector $\|$ /gaussian estimate). We give this special λ the index ZF. As before, this λ_{ZF} also minimises c_{ZF}:

$$\lambda_{\mathsf{ZF}} = \sqrt{\frac{N-r}{N}}\frac{\left\|\,\mathbf{F}\,\right\|_{\mu}}{\left\|\,\mathbf{f}_{\mathsf{ZF}}\,\right\|_{\mu,(N-r)}} = (N-r)\sqrt{\frac{N\delta(d)}{24d}}$$

$$c_{\mathsf{ZF}} = \sqrt{2N}\frac{\left\|\,(\lambda_{\mathsf{ZF}}\mathbf{f}_{\mathsf{ZF}},\mathbf{F})\,\right\|_{L,(N-r)}}{\sigma(L_{\mathsf{ZF}})} = \sqrt{2N\frac{\lambda^2_{\mathsf{ZF}}\frac{2d}{N-r}+N^2\frac{\delta(d)}{12}}{\frac{2N-r}{2\pi e}\left(q^N\lambda_{\mathsf{ZF}}^{N-r}\right)^{\frac{2}{2N-r}}}}.$$

This ratio c_{ZF} and $a_{\mathsf{ZF}} = (2N-r)/2q$ can be used to get the runtime which a zero-forcing attacker needs to find the secret key. For the counterparts of these formulas in the standard lattice, we refer the reader to [23], where NTRU also shows that these zero-forcing ratios can be better (i.e. smaller) than the normal ones. So the resulting constants A_{ZF} and B_{ZF} may well be more favourable to the zero-forcer. In [21], NTRU actually ignores this case, which we think should not be done. So our final formula is slightly different from the one they give.

The attack runtime is found as before by looking up constants in the table given from [23] or [28]. The attacker using zero-forcing can only be successful with certain patterns. We take this into account by multiplying his time with $1/\,\mathrm{P}(Y \geq 1)$, meaning that he has to try attacking that many times.

$$T_{\mathsf{ZF}} = \tfrac{1}{\mathrm{P}(Y\geq 1)}\,2^{A_{\mathsf{ZF}}(N-\frac{r}{2})+B_{\mathsf{ZF}}}$$

To express the *gain* that zero forcing gives, we look at the ratio $\mathcal{G}$:

$$\begin{aligned}\mathcal{G} &:= \tfrac{T_{\mathsf{normal}}}{T_{\mathsf{ZF}}}\\ &= \left(1-\left(1-\prod_{i=0}^{2d}\left(1-\frac{r}{N-i}\right)\right)^{2N}\right)\,2^{(A-A_{\mathsf{ZF}})N+(B-B_{\mathsf{ZF}})+A_{\mathsf{ZF}}\frac{r}{2}}.\end{aligned}$$

We finish this section with the complete bit strength of a given system against an LLL-type attack supported by zero-forcing. We assume that the

attacker will correctly precalculate the pattern length $r \in [0, 2N-2d-1]$ which maximises the gain $\mathcal{G}$.

$$\omega_{\mathsf{lk}}(N,d,A,B,A_{\mathsf{ZF}},B_{\mathsf{ZF}}) := \log_2(\ T_{\mathsf{normal}}\ ) - \max_{0\le r\le 2N-2d-1}\left\{\ \log_2(\ \mathcal{G}\ )\ \right\}$$

3.3 Combinatorial Forgery

We now come to the forgery attacks, where an attacker does not try to discover the private key $\mathbf{f}$ or $\mathbf{g}$, but instead tries to use the public parameters $(N, q, d, \beta, \mathcal{N})$ and the public key $\mathbf{h}$ to make a valid signature $\mathbf{s}$ for a given document hash $(\mathbf{m}_1, \mathbf{m}_2)^T$. Refer to section 2.4.3 to see that being able to sign $(\mathbf{0}, \mathbf{m})^T$ is already sufficient to sign any document. Also remember that β is the public scaling constant used to calculate the β-norm of a lattice vector $(\mathbf{v}, \mathbf{w})^T \in L(\mathbf{h}, q)$ (compare section 2.2):

$$\left\| \begin{pmatrix} \mathbf{v} \\ \mathbf{w} \end{pmatrix} \right\|_\beta = \sqrt{\| \mathbf{v} \|_q^2 + \beta^2 \| \mathbf{w} \|_q^2}.$$

This becomes important now, because the signature that the attacker tries to find is an $\mathbf{s} \in \mathcal{R}_q$ such that

$$\left\| \begin{pmatrix} \mathbf{s} \\ \mathbf{s} * \mathbf{h} - \mathbf{m} \end{pmatrix} \right\|_\beta^2 \le \mathcal{N}^2.$$

For a combinatorial forgery, the attacker starts by choosing a random $\mathbf{s} \in_R \mathcal{R}_q$. There are q^N many of those. Then he takes his chosen $\mathbf{s}$ and calculates $\mathbf{t} = \mathbf{s} * \mathbf{h} - \mathbf{m} \mod q$. If $\mathbf{h}$ is invertible (which is usually the case), then this calculation is reversible—meaning that there are again q^N distinct $\mathbf{t} \in \mathcal{R}_q$. If $\mathbf{h}$ is not invertible, then some of the $\mathbf{t}$ are not reached.

From these $\mathbf{t}$, only the ones in the ball B of radius $\mathcal{N}/\beta$ are interesting:

$$\mathbf{t} \in B = \left\{ \mathbf{v} \in \mathcal{R}_q \mid \| \mathbf{v} \|_q < \frac{\mathcal{N}}{\beta} \right\} \implies \left\| \begin{pmatrix} \mathbf{s} \\ \mathbf{t} \end{pmatrix} \right\|_\beta < \sqrt{\| \mathbf{s} \|_q^2 + \mathcal{N}^2}.$$

So $\| (\mathbf{s}, \mathbf{t})^T \|_\beta$ is fairly close to $\mathcal{N}$ and there is a good chance that $\mathbf{s}$ is a valid signature.

We say that this attack is dangerous if the probability of getting a $\mathbf{t} \in B$ from our randomly chosen $\mathbf{s}$ is too high. In order to obtain this probability, we need to find out how many elements are contained in B. We have that B is defined as a ball in the q-norm. For all elements $\mathbf{v} \in \mathcal{R}_q$, this is just the same as the μ-norm of the best representative (see 2.2). In the μ-norm, vectors in the direction of $\mathbf{1} = (1, \ldots, 1)^T$ are mapped to zero. And in the perpendicular space $\{\ \mathbf{1}\ \}^\perp$, the μ-norm is just the Euclidean norm.

By this observation we can estimate the number of points in B within the subspace $\{\,\mathbf{1}\,\}^{\perp}$ by the usual $(N-1)$-dimensional Euclidean volume of a ball with radius $\mathcal{N}/\beta$:

$$|B \cap \{\,\mathbf{1}\,\}^{\perp}| \approx \frac{\pi^{\frac{N-1}{2}}}{\Gamma(1+\frac{N-1}{2})} \left(\frac{\mathcal{N}}{\beta}\right)^{N-1} \overset{(1)}{=} \frac{\pi^{\frac{N-1}{2}}}{(\frac{N-1}{2})!} \left(\frac{\mathcal{N}}{\beta}\right)^{N-1}.$$

Equality (1) holds, since $N > 2$ is prime and hence odd.

Notice that for $\mathbf{v} \in B \cap \{\,\mathbf{1}\,\}^{\perp}$ we may add any of the q multiples of $\mathbf{1}$ from $\mathcal{R}_q$ without changing $\|\,\mathbf{v}\,\|_q$. So we get q different elements in B for any element in $B \cap \{\,\mathbf{1}\,\}^{\perp}$. In other words, the set B looks like a cylinder of length q and radius $\mathcal{N}/\beta$ along the vector $\mathbf{1}$:

$$|B| \approx q\,|B \cap \{\,\mathbf{1}\,\}^{\perp}| = q\,\frac{\pi^{\frac{N-1}{2}}}{(\frac{N-1}{2})!} \left(\frac{\mathcal{N}}{\beta}\right)^{N-1}.$$

Using this formula, the probability of guessing a $\mathbf{t} \in B$ becomes

$$\mathrm{P}(\mathbf{s} * \mathbf{h} - \mathbf{m} \in B) \overset{(1)}{=} \frac{|B|}{q^N} \approx \frac{\pi^{\frac{N-1}{2}}}{q^{N-1}(\frac{N-1}{2})!} \left(\frac{\mathcal{N}}{\beta}\right)^{N-1}.$$

Equation (1) holds only if $\mathbf{h}$ is invertible, otherwise the denominator is not q^N but less, namely $|T|$.

$$T = \left\{\, \mathbf{s} * \mathbf{h} - \mathbf{m} \;\middle|\; \mathbf{s} \in \mathcal{R}_q \,\right\}$$

The formula which NTRU gives on the same issue in [21] is different. They calculated the probability we see here using a different ball B', which uses the normal Euclidean norm instead of the centred one. This causes their ball to have less volume and so they underestimate the chances of the attacker.

The attacker will want to combine his guesses to have a better chance of success. This is possible in the following way: He chooses an $\mathbf{s}_i \in_R \mathcal{R}_q$ randomly again and calculates $\mathbf{t}_i = \mathbf{s}_i * \mathbf{h} - \mathbf{m}$. If the $\mathbf{t}_i$ is not in B, then he saves $\mathbf{t}_i + \mathbf{m}$ in a list. For all further $\mathbf{t}_j$, he checks whether there is a list element $\mathbf{t}_i + \mathbf{m}$ which is close to $\mathbf{t}_j$, i.e. $(\mathbf{t}_i + \mathbf{m}) - \mathbf{t}_j \in B$.

Should the attacker be lucky enough to find such a pair, he calculates back $\mathbf{s}_i$ and $\mathbf{s}_j$, and $\mathbf{s}_i - \mathbf{s}_j$ will most likely be a signature, since $(\mathbf{t}_i + \mathbf{m}) - \mathbf{t}_j \in B$:

$$\|\,(\mathbf{s}_i - \mathbf{s}_j) * \mathbf{h} - \mathbf{m}\,\|_q = \|\,\mathbf{t}_i - \mathbf{t}_j + \mathbf{m}\,\|_q < \frac{\mathcal{N}}{\beta}.$$

Should $\mathbf{h}$ not be invertible then the attacker cannot get back at $\mathbf{s}_i$ or $\mathbf{s}_j$ from his list, so he has to make a list of pairs $(\mathbf{s}_i, \mathbf{t}_i + \mathbf{m})$.

This meet-in-the-middle-type technique similar to the one introduced by Odlyzko in [29] square roots the amount of guesses our attacker has to make. We may equally say that the probability of his success is square rooted:

$$\mathrm{P}(\text{combinatorial forgery}) \approx \sqrt{\frac{\pi^{\frac{N-1}{2}}}{q^{N-1}(\frac{N-1}{2})!}\left(\frac{\mathcal{N}}{\beta}\right)^{N-1}} \stackrel{!}{<} 2^{-k}.$$

We are k-bit secure against this attack if the second inequality is satisfied:

$$\omega_{\mathsf{cfrg}}(N,q,\beta,\mathcal{N}) := -\frac{1}{2}\log_2\left(\frac{\pi^{\frac{N-1}{2}}}{q^{N-1}(\frac{N-1}{2})!}\left(\frac{\mathcal{N}}{\beta}\right)^{N-1}\right).$$

In the case of trinary keys and the transpose lattice, we have already calculated $\mathcal{N}$ in 2.4.3.

$$\begin{aligned}\omega_{\mathsf{cfrg}}(N,q,d,\beta) = {} & \frac{1-N}{4}\log_2\left(\frac{\pi\rho^2N^2\delta(d)}{q^2}\left(12\beta^{-2}+N\right)\right) \\ & + \frac{1}{2}\log_2\left((\tfrac{N-1}{2})!\right)\end{aligned}$$

3.4 Lattice-based Forgery

This is an alternative forgery attack. Again, the attacker wants to find a vector $(\mathbf{s},\mathbf{t})^T \in L(\mathbf{h},q)$ close to $(\mathbf{0},\mathbf{m})^T$, giving him the desired validation:

$$\left\| \begin{pmatrix} \mathbf{s} \\ \mathbf{t} \end{pmatrix} - \begin{pmatrix} \mathbf{0} \\ \mathbf{m} \end{pmatrix} \right\|_\beta^2 = \left\| \begin{pmatrix} \mathbf{s} \\ \mathbf{t}-\mathbf{m} \end{pmatrix} \right\|_\beta^2 \leq \mathcal{N}^2.$$

We may think of this as an $\mathcal{N}$-APPR-CVP in the lattice

$$L_\beta(\mathbf{h},q) = \left\{ (\mathbf{a},\beta\mathbf{b})^T \in \mathbb{Z}^N \times \beta\mathbb{Z}^N \;\middle|\; \mathbf{a} * \mathbf{h} = \mathbf{b} \mod q \right\}.$$

The difficulty of solving this CVP can be tied to two important lattice constants. This is the same basic observation that was made for regular lattice attacks on the private key, only that that was an SVP.

$$\gamma = \frac{\mathcal{N}}{\sigma(L_\beta(\mathbf{h},q))\sqrt{2N}} \qquad\qquad a = \frac{N}{q}$$

Experiments made by NTRU (see [21]) have shown—under the condition that these two fractions are roughly constant—that the runtime T of the attack increases exponentially as follows

$$\log(T) = A \cdot N + B$$

for certain constants A, B depending on the values of γ, a. Experiments show that for constant a and increasing γ, T increases exponentially. T also increases for constant γ and increasing a, but only slightly. So γ and a should be kept small.

$$\begin{aligned}\gamma &= \frac{\mathcal{N}}{\sigma(L_\beta(\mathbf{h}, q))\sqrt{2N}} = \frac{\mathcal{N}}{N}\sqrt{\frac{\pi e}{2q\beta}} \\ &= \frac{\rho}{N}\sqrt{\frac{\pi e}{2q}\left(\tfrac{1}{\beta}\cdot\mathcal{E}_{\mathbf{s}}^2 + \beta\cdot\mathcal{E}_{\mathbf{t}}^2\right)} \qquad \text{which is minimal for } \beta = \tfrac{\mathcal{E}_{\mathbf{s}}}{\mathcal{E}_{\mathbf{t}}} \\ &\geq \frac{\rho}{N}\sqrt{\frac{\pi e\ \mathcal{E}_{\mathbf{s}}\mathcal{E}_{\mathbf{t}}}{q}}\end{aligned}$$

Note that we may not choose $\beta = \mathcal{E}_{\mathbf{s}}/\mathcal{E}_{\mathbf{t}}$ freely, because we have to ensure a security against combinatorial forgery as well with the same β. So the β of choice is the smallest one bigger than $\mathcal{E}_{\mathbf{s}}/\mathcal{E}_{\mathbf{t}}$ which guarantees us a big enough $\omega_{\mathsf{cfrg}}(N, q, \beta, \rho)$. The actual security against lattice-based forgery ω_{lfrg} is given in table 2 for a collection of sensible pairs (γ, a).

upper bound for (γ, a)		$\omega_{\mathsf{lfrg}}(N)$
0.1774	1.305	$0.995113N - 82.6612$
0.1413	0.707	$1.16536N - 78.4659$
0.1400	0.824	$1.14133N - 76.9158$

Table 2: Experimental security against lattice-based forgery ω_{lfrg} as shown in [21].

4 Parameter Algorithm for NTRUSign

This section describes a parameter generation algorithm for NTRUSign which was proposed in [21]. We added some minor improvements where we found them. It uses all the ω bit-securities that were found in the previous section. In accordance with these ω, the parameters we calculate will be for the NTRUSign instance using *trinary* keys and the *transpose* lattice. The algorithm produces a list $\mathcal{L}$ of possible NTRUSign parameters for a given bit security level k.

4.1 Input and Output

The input of our algorithm does not only encompass the desired bit security level k, but also a signing tolerance ρ, which was defined along with NTRUSign itself in section 2.4.3. It measures how tolerant we are towards signature failures. A signature failure occurs when we produce an invalid signature with a valid secret key. For example, $\rho = 1$ corresponds to a normal tolerance, and $\rho = 1.1$ is a setting where failures become very unlikely. If signature failures do occur, we may always append a random string to our document and re-hash and re-sign it.

Another input to the algorithm is an upper bound for the NTRUSign parameter N. It is called N_{max} and ensures that the parameters we generate do not make the whole system too slow. Recall that N is the size of the elements we use and thus corresponds directly to the speed and size requirements.

We added another parameter c_{min} to the algorithm in [21], which gives the user direct control over the general lattice security of all generated NTRUSign instances. This parameter is a lower bound for the ratio c, which was introduced in section 3.2. The bigger c is, the harder lattice-based attacks are in general ([21] uses $c_{\mathsf{min}} = 3.8$):

k	Desired bit security level	$k = 80$ for standard security.
ρ	Signing failure tolerance	$\rho = 1.1$ for little tolerance.
N_{max}	Upper bound for N	$N_{\mathsf{max}} = 2\rho k$ is a good bound.
c_{min}	Lower bound for c	$c_{\mathsf{min}} = 3.8$ as in [21].

The elements of the output list $\mathcal{L}$ are NTRUSign parameters. They were introduced in section 2 and 3.

N	Polynomials in $\mathcal{R}$ have a degree less than N
q	The coefficients of all polynomials in $\mathcal{R}_q$ are reduced modulo q
d	The keyspace $\mathcal{K}$ contains all trinary polynomials from $\mathcal{T}(d)$
β	The scaling constant used to prevent forgeries
$\mathcal{N}$	The norm-bound used to verify a signature

Algorithm 1: Parameter generation

input : $k, \rho, N_{\mathsf{max}}, c_{\mathsf{min}}$
output: A list $\mathcal{L}$ containing all parameter sets $(N, q, d, \beta, \mathcal{N})$ with $N \leq N_{\mathsf{max}}$, $c \geq c_{\mathsf{min}}$, and k-bit security.

1 $N \longleftarrow 1$
2 $N \longleftarrow \min\{\, p \in \mathbb{N} \mid p \text{ is prime and } N < p \,\}$; // N (prime) increases
3 **if** $N > N_{max}$ **then return** $\mathcal{L}$
4 $q_{\mathsf{maxExact}} \longleftarrow \left(2\pi e\, \delta(\frac{N}{3})\, \sqrt{N^3}\right) / \left(c_{\mathsf{min}}^2 \sqrt{3}\right)$; // see below
5 $q_{\mathsf{max}} \longleftarrow 2^{\lfloor \log_2(q_{\mathsf{maxExact}}) \rfloor}$
6 $q \longleftarrow 2^4$
7 $q \longleftarrow 2q$; // q (power of 2) increases
8 **if** $q > q_{max}$ **then goto** 2 ; // 1 -- 8 set N and q
9 $D = \{\, d \in \mathbb{N} \mid d \leq \frac{N}{3} \text{ and } \omega_{\mathsf{cmb}}(N, d) \geq k \,\}$
10 **if** $D = \varnothing$ **then goto** 2 ; // ensure $\omega_{\mathsf{cmb}} \geq k$
11 $d \longleftarrow \min D$
12 $c \longleftarrow \sqrt{(2\pi e\, \delta(d)\, \sqrt{N^3})/(q\sqrt{3})}$; // compare 3.2
13 $a \longleftarrow N/q$
14 get runtime constants A, B from a table with c, a
15 get r maximising the ZF gain $\mathcal{G}$, where $A_{\mathsf{ZF}} = A$ and $B_{\mathsf{ZF}} = B$
16 calculate c_{ZF} and a_{ZF} according to section 3.2.1
17 get ZF-runtime constants $A_{\mathsf{ZF}}, B_{\mathsf{ZF}}$ from a table with $c_{\mathsf{ZF}}, a_{\mathsf{ZF}}$
18 **if** $\omega_{lk}(N, d, A, B, A_{ZF}, B_{ZF}) < k$ **then** // ensure $\omega_{\mathsf{lk}} \geq k$
19 $\quad d \longleftarrow d + 1$
20 $\quad$ **goto** 12
21 **if** $d > \frac{N}{3}$ ***or*** $c < c_{min}$ **then goto** 2 ; // 9 -- 21 set d
22 $B = \{\, \beta \in [\sqrt{12/N}, 1] \mid \omega_{\mathsf{cfrg}}(N, q, d, \beta) \geq k \,\}$
23 **if** $B = \varnothing$ **then goto** 7 ; // ensure $\omega_{\mathsf{cfrg}} \geq k$
24 $\beta \longleftarrow \min B$
25 $\mathcal{N} \longleftarrow \frac{\rho N}{6} \sqrt{\delta(d)\,(12 + \beta^2 N)}$; // compare 2.4.3
26 $\gamma \longleftarrow \mathcal{N}\sqrt{\pi e} \,/\, (N\sqrt{2q\beta})$; // compare 3.4
27 **if** $\omega_{lfrg}(\gamma, a) < k$ **then goto** 7 ; // ensure $\omega_{\mathsf{lfrg}} \geq k$
28 $\mathcal{L} \overset{\text{append}}{\longleftarrow} (N, q, d, \beta, \mathcal{N})$; // 22 -- 27 set β
29 **goto** 2

4.2 The Algorithm

Steps 1 – 8 . This is the basic setup of the two most important parameters, N and q. We try to make parameters with N from the first prime upward till we go over N_{max}. For each N, we ensure that q is a power of 2 which is smaller than q_{maxExact}. The purpose of this q_{maxExact} is to ensure that c is still big enough, as q gets larger.

$$c = \sqrt{(2\pi e\, \delta(d)\, \sqrt{N^3}\,)/(q\sqrt{3}\,)} \qquad \text{as } d \to \tfrac{N}{3}$$

$$c \to \sqrt{(2\pi e\, \delta(\tfrac{N}{3})\, \sqrt{N^3}\,)/(q\sqrt{3}\,)} \qquad \text{since } q_{\mathsf{maxExact}} \geq q$$

$$c \geq \sqrt{(2\pi e\, \delta(\tfrac{N}{3})\, \sqrt{N^3}\,)/(q_{\mathsf{maxExact}}\sqrt{3}\,)} = c_{\mathsf{min}}$$

In other words, if we increased q beyond q_{maxExact}, then c would certainly fall below c_{min} once d gets big. So once q reaches its bound, we fall back to raising N, which in turn increases the bound on q.

Steps 9 – 11 , 12 – 21 . We find the smallest d for which $\omega_{\mathsf{cmb}} \geq k$. The value of ω_{cmb} will increase further for bigger d, so we increment d by steps of 1 until either $\omega_{\mathsf{lk}} \geq k$ or $d > N/3$. If $d > N/3$, then d is too big and we have to fall back. Since a raising q increases neither ω_{cmb} nor ω_{lk}, we have to fall back to raising N.

The formula for **14** – **17** are given in section 3.2. All except for the one providing the r. Since there are only $2(N-d)$ possible candidates for r, the maximising one can be found by calculating all possible gains.

Steps 22 – 24 , 25 – 27 . In the same way that we found a fitting d in the last steps, we now try to find a fitting β. We know from section 3.4 that we want β to be the smallest possible number bigger than $\mathcal{E}_{\mathbf{s}}/\mathcal{E}_{\mathbf{t}} = \sqrt{12/N}$ (compare 2.4.3) which satisfies $\omega_{\mathsf{cfrg}} \geq k$. So we take this β, if it exists, and check whether $\omega_{\mathsf{lfrg}} \geq k$ as well. We can now fall back to raising q instead of N, since this will significantly increase ω_{cfrg}. If we raise q beyond q_{max} at this point, q will be reset and N will be raised instead.

Remark. We would in practice replace steps **18** and **19** by setting β equal to the root of $f(\beta) = \omega_{\mathsf{cfrg}}(N, q, d, \beta) - k$, and then checking whether this is in the interval $[\sqrt{12/N}, 1]$.

Steps 24 – 25 . All parameter combinations reaching **22** are usable, so we append them to $\mathcal{L}$ and try the next bigger N. We could also try for a bigger q at this point and fall back to increasing N later, once q_{max} is reached, but this would be unpractical. For, once we have found a working k-bit secure parameter combination for a fixed N, finding another one with a bigger q would only increase the size of the public/private-key and the signature.

Finding the best combination. For every element of $\mathcal{L}$, we may calculate some basic information about the NTRUSIGN instance it would create.

$T_{\mathsf{S}} = 8dN + N^2$	time to sign
$T_{\mathsf{V}} = N^2$	time to verify
$b = N \log_2(q)$	bitsize of public/private-key and signature

We may then choose the best element of $\mathcal{L}$ fitting to our needs.

5 Testing of Lattice-Based Attacks

In this section, we want to test whether the security against lattice-based attacks in section 3.2 is well founded. In order to do this, we will run two tests. In the first test, we will check whether changing both from binary to trinary keys and from the standard to the transpose lattice has any effect on the linearity assumption we used. This assumption was made by NTRU in [23]. They state that if the two ratios a, c are kept constant, the logarithm of the average breaking time for an NTRU instance is linear in N, i.e. $\log(T_{\mathsf{avg}}(N)) = AN + B$.

$$a = \frac{N}{q} \qquad\qquad c = \sqrt{2N}\,\frac{\|\text{shortest vector}\|}{\text{Gaussian estimate}}$$

NTRU claims this is true for binary-key systems with the standard lattice, such as the ones used for NTRUEncrypt. We recheck the claim here, because in an NTRUSign setting one would rather use trinary keys and the transpose lattice.

In the second test, we want to check whether the improvements to lattice reduction in general made by Schnorr (RSR, [48]), and in particular their practical counterpart developed by Ludwig (SR, [8]), do significantly outperform the standard LLL/BKZ reductions (see 1.3.2) on NTRU lattices. There has been some controversy about this. After the posting of [8], NTRU was displeased with the analysis behind a claim that their systems were not as secure against lattice attacks as they are supposed to be [25]. So we will check whether the linearity mentioned in test one still holds for RSR/SR attacks. If this is so, we can proceed and imitate the way NTRU themselves analysed their lattice based security to see if there is a significant change.

In either test we will not concern ourselves with the lattice reduction improvements derived from *zero-forcing* (ZF, [42]). We can do this, because as we have seen in section 3.2.1, these improvements can be taken into account later, when the final security of an NTRU system against lattice-based attacks is calculated. Of course, this belated changing of the security can only be done if NTRU's linearity claim mentioned before holds in both tests.

In our test-attacks we will generally follow the ideas which Coppersmith and Shamir published in [11]. They were the first to use lattice reduction techniques in order to break NTRU systems, and their way of attacking is still the best one known.

5.1 Setting up

The problem we want to solve using lattice reduction is this: Given an NTRU parameter set $(N, q, d, \beta, \mathcal{N})$ and a public key $\mathbf{h}$, find the corresponding

private key $\mathbf{f}$. In fact, β and $\mathcal{N}$ have no influence on the ratios a, c or the NTRU lattice $L(\mathbf{h}, q)$ (see 2.4.1), so we will omit them. Normally, we would only consider prime N, but we will omit this as well to get more test results. Keep in mind that we would never use the parameter sets we generate here in practice, since we want to break them, but they should resemble practical ones as closely as possible.

The next question is which a, c should be considered. NTRU themselves tell us in [23], that a smaller a or c both result in faster breaking times, which is reasonable. They use a particularly small pair ($a = 0.535, c = 1.73$), because this way they get a lot of points, i.e. lots of pairs $(N, \log(T))$ for their linear regression line. In their technical report, NTRU goes from $N = 80$ up to $N = 122$. This makes the estimations for higher N (the ones which are used in practice) more exact, and it increases the chance of spotting a mistake in the assumption that there is a linear relationship between N and $\log(T)$. For these reasons we will use a similar pair of ratios.

Note that the ratio c is below anything we would calculate with the parameter generation algorithm from section 4. As we have seen, the algorithm ensures that $c \geq 3.8$. In fact, the parameter sets calculated with this algorithm that were presented in [21] have $c = 5.33$ as a minimum. So there is a good margin between what we will run our tests on and what is actually used.

Since we fixed a and c, once we also fix an N we have already fixed the additional parameters q and d as well:

$$q = \frac{N}{a} \qquad\qquad d = \sqrt{\frac{3}{N}}\frac{qc^2}{4\pi e} + \frac{1}{2N} - \frac{1}{2}.$$

However, q is a power of 2, and d is an integer parameter, so we need to do some rounding to make sure that the real ratios, the ones resulting from the proper q and d, are always bigger than the original minimal ($a = 0.535, c = 1,73$). As a result, the parameter sets we will test our algorithms on are the following:

N	q	d	a	c	N	q	d	a	c
80	128	2	0.625	1.854	112	128	2	0.875	2.017
84	128	2	0.656	1.877	116	128	2	0.906	2.035
88	128	2	0.688	1.899	120	128	2	0.938	2.052
92	128	2	0.719	1.920	124	128	2	0.969	2.069
96	128	2	0.750	1.941	128	128	2	1.000	2.086
100	128	2	0.781	1.961	132	128	2	1.031	2.102
104	128	2	0.813	1.980	136	128	2	1.063	2.118
108	128	2	0.844	1.999	140	256	3	0.547	1.785

The random NTRU instances for each parameter set will be generated using a Java program which implements the NTRUSign public key cryptosystem. This implementation is in full accordance with the efficiency standards (EESS, [52]) published by the "Consortium for Efficient Embedded Security (CEES)," of which NTRU is an active part. The generation program is named *NTRUSignProvider* and is an experimental part of the *FlexiProvider*, which has been tested for proper usage by the faculty of computer science at the Technical University of Darmstadt (the link is [45]).

The implementation we use to realise the attacks in both tests is the LaRed-package for Python/C, which was written by Ludwig as a part of [41]. Should he decide to publish his code, it will be available as a part of the LiDIA C++ library [40]. In the case of LLL and BKZ attacks, Ludwig's package is just a wrapper for the standard implementation of these algorithms, which were written by Shoup (available here: [50]).

The actual lattice we reduced was not the NTRU lattice, but as suggested in [11] we use a similar lattice were all vectors have mean zero. We will give the $\mathcal{R}$-basis of the lattice, because as discussed earlier, CMLs can be seen as $\mathcal{R}$-modules of rank 2 (see 2.4.2).

$$\mathcal{B}_{\mathsf{CS}} = \left\{ \begin{pmatrix} \lambda\mathbf{e} - \mu(\lambda\mathbf{e})\mathbf{1} \\ \mathbf{h} - \mu(\mathbf{h})\mathbf{1} \end{pmatrix}, \begin{pmatrix} \mathbf{0} \\ \mathbf{q} - \mu(\mathbf{q})\mathbf{1} \end{pmatrix} \right\}$$

In this lattice we will not find the balanced secret keys $(\lambda\mathbf{f}, \mathbf{F})^T$ / $(\lambda\mathbf{g}, \mathbf{G})^T$, but their zero-mean counterparts:

$$(\lambda\mathbf{f} - \mu(\lambda\mathbf{f})\mathbf{1}, \mathbf{F} - \mu(\mathbf{F})\mathbf{1})^T \ / \ (\lambda\mathbf{g} - \mu(\lambda\mathbf{g})\mathbf{1}, \mathbf{G} - \mu(\mathbf{G})\mathbf{1})^T.$$

Should we find any of these small vectors, we can reconstruct $\mathbf{f}$ or $\mathbf{g}$, because we know that $\mu(\lambda\mathbf{f}) = \mu(\lambda\mathbf{g}) = \lambda/N$. The Euclidean norm in this lattice corresponds to the L-norm in the normal NTRU lattice. This will usually cause the c ratio to be better for this lattice, because the Gauss-estimate used to calculate c was supposed to be an estimate for the normal Euclidean norm and not the centred one. We didn't have time to find out how much c changes though. The different norm in the zero-meaned lattice also causes that the counterparts of the secret keys to be balanced in the Euclidean norm. And the dimension of the changed lattice is not $2N$, but $2N-2$. The two missing dimensions are the $\mathbb{Z}$-span of the vectors $(\mathbf{1}, \mathbf{0})^T$ and $(\mathbf{0}, \mathbf{1})^T$.

5.2 LLL and BKZ

When we started testing, we encountered some problems. We started with $N = 80$ and found that some instances where broken after 100 seconds, while others were aborted after taking more than 10.000 seconds runtime. When

we reduced the dimension to $N = 60$, we still found instances which were unbroken after 10.000 seconds. Only when we retreated to $N = 40$ was it possible to solve them all. It became apparent that the time taken to break an instance for a fixed dimension was hard to predict, because some were broken really fast and some took very long.

We found that this effect was correlated with the balancing of the lattice we reduced. When balancing the transpose lattice, you have to estimate the length of the $\mathbf{F}$ or $\mathbf{G}$ key. In the settings we chose, i.e. the ones with a small d (number of ones and minus ones in the other key $\mathbf{f}$), this estimate is very unpredictable. It can happen that it is very exact, but most of the time it is off by a factor of 3 to 10. This leads to an enormous variance in our results, which contradicted the linearity we wanted to find.

We also found that if the settings were more realistic, i.e. if we generated systems with more realistic c-ratio, then the estimates became far better. In realistic settings they were very good most of the time, and only in some cases off by a factor of up to 2. So to make our tests more realistic, we decided to give the attacker the knowledge of the balancing constant up to the second decimal after the comma, which is $\lfloor \| \mathbf{F} \|_\mu / \| \mathbf{f} \|_\mu * 100 \rfloor$. A real attacker does not have this knowledge, but he does have a good estimate and can run many reductions in parallel with balancing constants close to his estimate. So an attacker might make up for his lack of knowledge by running his attacks in parallel on multiple machines.

Note that the problem we had with balancing has not been encountered by NTRU or Coppersmith and Shamir in [23, 11], because they reduced the standard lattice where balancing is not an issue. This is because the length of the private key $\mathbf{f}$ is known from public parameters for both binary and trinary keys, and the standard lattice is balanced with $\| \mathbf{g} \|_\mu / \| \mathbf{f} \|_\mu \approx 1$.

Our test results are displayed in figure 2. In this graph, we see little black crosses representing the breaking time in seconds. We considered an instance broken if either of the two keys was recovered. The red line indicates an estimate made by NTRU for a similar setting, but with the standard lattice and binary keys (see [23]). We converted their results, which were given in MIPS years, to a corresponding runtime in seconds on our 2.4 GHz machine.

The thin blue line indicates the average breaking times for each N and the thick blue line is the linear regression through those. As we can see, the slope is about the same as NTRU's estimate, but the offset is a little higher. This might be an effect of using the new keytype or latticetype. We were able to break all instances for $N \leq 96$.

In those results, we saw that there were always some weak instances among our set of 30. To explore this effect, we decided to break only 3 instances (10%) for each N and then move on. The results are represented by the

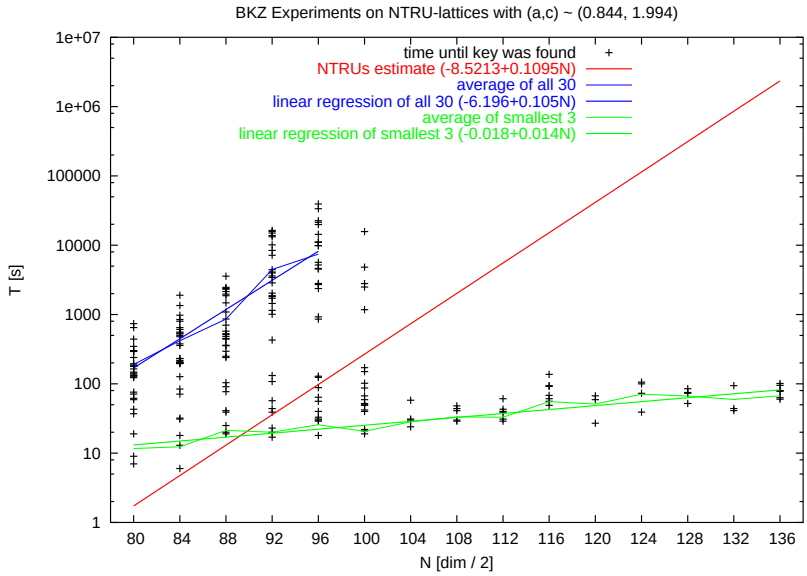

Figure 2: BKZ Breaking times for NTRU lattices, average $a = 0.8, c = 2$.

green lines. The thin line is again the average over the first three broken instances, and the thick line is again the linear regression through those. We can plainly see that the slope here is far shallower than the estimate, which basically means you have a 10% chance that an NTRU instance is less secure than estimated.

Enlightened by this result, we also tried to break some weak instances in a stronger setting, i.e. with a higher a and c. We observed the same effect (shown in figure 3) but with a steeper slope.

In this stronger setting, we do not have a comparable result from NTRU, but it is plain that even the weak instances are much harder to break here. For $N = 88$, we observe the upward concavity, that NTRU stated a lot.

On another note, we found that almost all instances were broken directly after the reduction methods found a vector shorter than the rotations of the q-vector $(\mathbf{0}, \mathbf{q})^T$, which are in the public basis of the lattice. This means that the observation by Coppersmith and Shamir, that a vector of twice or even four times the size of the private key could be used for partial decoding is not applicable in NTRU's settings.

The next step would be to put together a table of extrapolated slopes and offsets, for different c-ratios. This could then be used instead of the one from section 3.2, to eliminate the 10% chance of the attacker that a weak lattice was generated. On the other hand, one might investigate the differences between the instances that broke so fast and the average instances, which

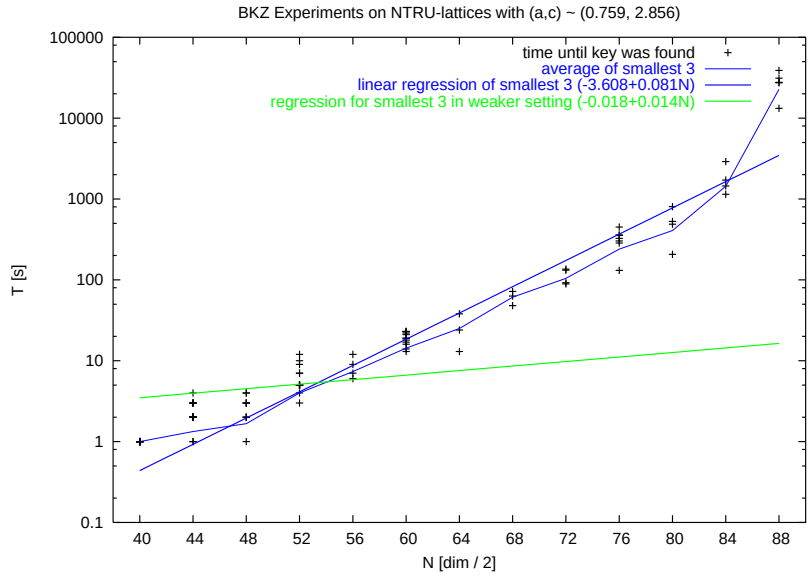

Figure 3: BKZ Breaking times for NTRU lattices, average $a = 0.7, c = 2.8$.

took much longer. In the scope of this work however, we have time for neither.

5.3 RSR and SR

In test two, we are curious how new reduction techniques like RSR and SR change the graph of the attacks. We tried to attack the NTRU lattices in the following way: We used normal BKZ reductions with increasing BKZ-parameter β and checked after each iteration how closely the orthogonalisation of the basis vectors follows a geometric series. Here we have two examples of how this can look:

We see here two graphs that were made after the 1^{st} (left side) and the 8^{th} (right side) BKZ reduction of an $N = 80$ instance. In both cases, only the first 40 vectors of the basis are shown. The graph shows the length of each base-vector squared (green line) and the length of their orthogonalisation squared (red line). The length scale is logarithmic, so the geometric series we look for is a straight line in that scale (dashed line).

What we do is sum the differences between the red and the dashed line to get a measure of how close the two are. If the sum is too big (like on the left side), we either increase β and perform more BKZ reductions, or, if only a few vectors are away from the geometric line, we use ShortProjectionSR to specifically shorten these few and get a smaller sum of differences. When these measures are successful, the red and dashed line are fairly close, which

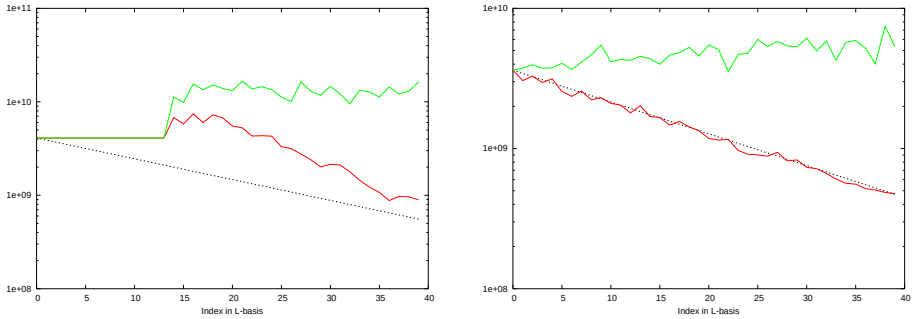

Figure 4: We either find no geometric series (left), or we find one (right).

gives us an extremely high chance of guessing a very short vector using RSR. After this short vector that we guessed is inserted to the front of the basis, we are either done, or the sum of differences is very big again, which means more projections or BKZ reductions are in order.

During our tests, we observed that using either ShortPoolSR (to guess more than one short vector) or adding all N *rotations* of a new short vector given by an RSR step does *not* speed up the reduction.

We discovered—similar to what Ludwig mentions in [8] and later in [41]—that the ShortProjectionSR attack can be used to slightly reduce the BKZ-parameter needed to break an NTRU instance. However, to achieve this effect one needs to run ShortProjectionSR *interactively* and not through a script. This is necessary, because it is hard to tell exactly which vector can be reduced with a projection, and giving the algorithm a wrong target takes lots of runtime—and sometimes even worsens the reduction-situation. So you need to be able to backtrack your attacks and target a different basis vector sometimes.

Being non-scriptable, this attack cannot be investigated in the way we set out to do it. But using RSR to our advantage may still be possible if we approach the problem in another way. We propose to reduce a different public basis, which contains longer vectors to begin with, but where RSR can be used directly. We needed to switch to ShortProjectionSR because the orthogonalised basis did not follow a geometric series. This effect is caused by the different length of the vectors in the public basis we use. The rotations of the q-vector are much shorter than all other basis vectors, and they are also orthogonal, which means that the length of their orthogonalisation yields a plateau, which destroys the geometric series we look for (this effect can be observed in figure 4 (left)).

We look for an alternate public bases. Using linear algebra methods similar

to what we did in section 2.4.2, we can show that

$$\mathcal{B} := \left\{ \begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix}, \begin{pmatrix} \mathbf{c} \\ \mathbf{d} \end{pmatrix} \right\}$$

is an $\mathcal{R}$-basis of $L(\mathbf{h}, q)$ if and only if

$$\begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix}, \begin{pmatrix} \mathbf{c} \\ \mathbf{d} \end{pmatrix} \in L(\mathbf{h}, q) \qquad \text{and} \qquad \mathbf{a} * \mathbf{d} - \mathbf{b} * \mathbf{c} \equiv \mathbf{q}$$

So two vectors in L form a basis if the matrix containing them has determinant q in $\mathcal{R}$. We assume that $\mathbf{h}$ the public key is invertible modulo q. Then one such alternate basis is

$$\left\{ \begin{pmatrix} \mathbf{h}_q^{-1} \\ \mathbf{e} \end{pmatrix}, \begin{pmatrix} \mathbf{q} \\ \mathbf{0} \end{pmatrix} \right\}.$$

The vectors in this basis are about as big as the ones of the usual basis. The $(\mathbf{q}, \mathbf{0})^T$-vector has exactly the same size as its counterpart. In the usual basis, the public key $\mathbf{h}$ is very big, and $\mathbf{h}_q^{-1}$ is not much bigger in practice.

Balancing does not suit this alternate basis well, because it makes the big terms in the upper part even bigger. The result is that the vector-sizes we try to reduce are bigger to begin with, but as a tradeoff the problems we mentioned earlier with the q-vectors messing up the geometric series does not occur here. So far tests indicate, that the overall reduction time in these bases is slightly faster when using RSR compared to BKZ alone. However it is not as fast as using regular BKZ on the normal basis.

During our experiments with the alternate basis, we found that $\mathbf{h}$ is often invertible for small primes p like 2 and 3. In these cases the vector $(\mathbf{h}_p^{-1}, \mathbf{h} * \mathbf{h}_p^{-1})^T$, was sometimes slightly shorter than the one resulting from the public key itself $(\mathbf{e}, \mathbf{h})^T$. In these cases, adding this vector to the lattice basis before beginning the reduction may speed up the process.

Conclusion

In conclusion, we would like to say first that following the results from section 5, a serious change should be made in the way the lattice based security is calculated by NTRU. Or, as a different solution, it would be nice to be able to identify the 10% of weak NTRU lattices without actually attacking them.

We recommend to investigate the changes to c made by using the zero-meaned lattice, which we used upon the suggestion of Coppersmith and Shamir in [11]. Then one could find out if the weak 10% have a smaller c value, or if the a and c ratios are maybe too coarse a measure to analyse lattice reduction in the way NTRU wants to (compare section 3.2).

We were sad to leave open the analysis of using modern lattice reduction methods, like Schnorr's RSR or Ludwig's practical SR on the new public basis suggested at the end of section 5. This was due to a lack of time, so further answers in that direction will definitely be found in the near future.

Another interesting point for further work is the new private basis generation method, which we presented early on in section 2.4.2. We recommend to further compare the two methods from this section, in order to find out how much better the new one works in practice, and if it takes more or less time on average.

Another good subject to follow up on came up during our analysis of the combinatorial security in section 3.1. Here, we found a new way to predict the probability of success for an attacker. For an 80-bit secure example, this probability was about 2/3, which is quite high. So one could investigate if it is possible for the attacker to risk more and have a smaller chance of success, but finish the attack much faster, falling below the 80-bit boundary.

As a final word, we want to state that NTRUSign appears to be a good system to investigate more. It is not secure enough yet to recommend it for common usage, but there seems to be much room for improvement.

References

[1] Leonard M. Adleman. On breaking generalized knapsack public key cryptosystems. In *Proceedings of the 15th annual ACM Symposium on Theory of Computing*, pages 402–412. ACM Press, 1983.

[2] Miklós Ajtai. The shortest vector problem is NP-hard for randomized reductions. In *Proceedings of the 30th annual ACM Symposium on Theory of Computing*, pages 10–19. ACM Press, 1998.

[3] Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the 29th annual ACM Symposium on Theory of Computing*, pages 284–293. ACM Press, 1997.

[4] Miklós Ajtai, Ravi Kumar, and Dendapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the 33rd annual ACM Symposium on Theory of Computing*, pages 601–610. ACM Press, 2001.

[5] H. Alt and M. Habib, editors. *STACS 2003: 20th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2607 of *LNCS*. Springer, 2003.

[6] Wieb Bosma, editor. *Algorithmic Number Theory, ANTS-IV*, volume 1838 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[7] Johannes Buchmann, Carlos Coronado, Martin Döring, Daniela Engelbert, Christoph Ludwig, Raphael Overbeck, Arthur Schmidt, Ulrich Vollmer, and Ralf-Philipp Weinmann. Post-quantum signatures. Technical Report 297, Cryptology ePrint Archive, 2004. `http://eprint.iacr.org/2004/297/`.

[8] Johannes Buchmann and Christoph Ludwig. Practical lattice basis sampling reduction. Technical Report 072, Cryptology ePrint Archive, 2005. `http://eprint.iacr.org/2005/072/`.

[9] Joe P. Buhler, editor. *Algorithmic Number Theory, ANTS-III*, volume 1423 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[10] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993.

[11] Don Coppersmith and Adi Shamir. Lattice attacks on NTRU. In Fumy [13], pages 52–61.

[12] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical Report 04, Mathematische Instituut, University of Amsterdam, 1981. http://staff.science.uva.nl/~peter/vectors/abstract.html.

[13] Walter Fumy, editor. *Advances in Cryptology – EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[14] Craig Gentry. Key recovery and message attacks on NTRU-composite. *Lecture Notes in Computer Science*, 2045:182–194, 2001.

[15] Craig Gentry and Mike Szydlo. Cryptanalysis of the revised NTRU siganture scheme. *Lecture Notes in Computer Science*, 2332:299–320, 2002.

[16] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Kaliski [32], pages 112–131.

[17] Oded Goldreich, Daniele Micciancio, Shmuel Safra, and Jean-Pierre Seifert. Approximating shortest lattice vectors is not harder than approximating closet lattice vectors. Technical Report 002, Electronic Colloquium on Computational Complexity, 1999. http://eccc.hpi-web.de/eccc-reports/1999/TR99-002/.

[18] Gene H. Golub and Charles F. van Loan. *Matrix Computations 2nd Edition*. The John Hopkins University Press, 1989.

[19] Bettina Helfrich. Algorithms to construct Minkowski reduced and Hermite reduced bases. *Theoretical Computer Science*, 41:125–139, 1985.

[20] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. NTRUsign: Digital signatures using the NTRU lattice. *Lecture Notes in Computer Science*, 2612:122–140, 2003.

[21] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. Performance improvements and a baseline parameter generation algorithm for NTRUsign. http://grouper.ieee.org/groups/1363/lattPK/submissions.html, 2005.

[22] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Buhler [9], pages 267–288.

[23] Joffrey Hoffstein, Jospeh H. Silverman, and William Whyte. Estimated breaking times for NTRU lattices. Technical Report 012, Version 2, NTRU Cryptosystems, 2003. http://ntru.com/cryptolab/tech_notes.htm#012.

[24] Alston S. Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM*, 5(4):339–342, 1958.

[25] Nick Howgrave-Graham, Jeffrey Hoffstein, Jill Pipher, and William Whyte. On estimating the lattice security of NTRU. Technical Report 104, Cryptology ePrint Archive, 2005. `http://eprint.iacr.org/2005/104/`.

[26] Nick Howgrave-Graham, Phong Q. Nguyen, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, and William Whyte. The impact of decryption failures on the security of NTRU encryption. *Lecture Notes in Computer Science*, 2729:226–246, 2003.

[27] Nick Howgrave-Graham, Joseph H. Silverman, Ari Singer, and William Whyte. NAEP: Provable security in the presence of decryption failures. Technical Report 172, Cryptology ePrint Archive, 2003. `http://eprint.iacr.org/2003/172/`.

[28] Nick Howgrave-Graham, Joseph H. Silverman, and William Whyte. Choosing parameter sets for NTRUencrypt with NAEP and SVES-3. *Lecture Notes in Computer Science*, 3376:118–135, 2005.

[29] Nick Howgrave-Graham, Jospeh H. Silverman, and William Whyte. A meet-in-the-middle attack on an NTRU private key. Technical Report 004, Version 2, NTRU Cryptosystems, 2003. `http://ntru.com/cryptolab/tech_notes.htm#004`.

[30] Nick A. Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.

[31] Éliane Jaulmes and Antoine Joux. A chosen-cipher attack against NTRU. *Lecture Notes in Computer Science*, 1880:20–35, 2000.

[32] Burton S. Kaliski, editor. *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[33] Ravi Kannan. Improved algorithm for integer programming and related lattice problems. In *Proceedings of the 15th annual ACM Symposium on Theory of Computing*, pages 193–206. ACM Press, 1983.

[34] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12(5):415–440, 1987.

[35] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Edition): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[36] Henrik Koy. Primale-duale segment-reduktion. http://www.mi.informatik.uni-frankfurt.de/research/papers.html , 2004.

[37] Henrik Koy and Claus Peter Schnorr. Segment and strong segment LLL-reduction of lattice bases. http://www.mi.informatik.uni-frankfurt.de/research/papers.html , 2002.

[38] Jeffrey C. Lagarias, Hendrik W. Lenstra, and Claus Peter Schnorr. Korkin-Zolotarev bases and successive minima of a lattice and its reciprocal lattice. *Combinatorica*, 10(4):333–348, 1990.

[39] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lóvasz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[40] Lidia a c++ library for computational number theory. http://www.informatik.tu-darmstadt.de/TI/LiDIA/.

[41] Christoph Ludwig. *Practical Lattice Basis Sampling Reduction*. PhD thesis, Technische Universität Darmstadt, 2005. http://elib.tu-darmstadt.de/diss/000640/.

[42] Alexander May and Joseph H. Silverman. Dimension reduction methods for convolution modular lattices. In Silverman [51], pages 110–125.

[43] Phong Q. Nguyen and Jacques Stern. Lattice reduction in cryptology: An update. In Bosma [6], pages 85–112.

[44] Phong Q. Nguyen and Jacques Stern. The two faces of lattices in cryptology. In Silverman [51], pages 146–180.

[45] Stefan Pochmann. NTRUsign provider for flexiprovider. http://www.flexiprovider.de/.

[46] Claus Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53:201–224, 1987.

[47] Claus Peter Schnorr. Block reduced lattice bases and successive minima. *Combinatorics, Probability and Computing*, 4:1–16, 1994.

[48] Claus Peter Schnorr. Lattice reduction by random sampling and birthday methods. In Alt and Habib [5], pages 146–156.

[49] Claus Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 1994.

[50] Victor Shoup. Number theory library (NTL) for c++. `http://www.shoup.net/ntl/`.

[51] Joseph H. Silverman, editor. *Cryptography and Lattices*, volume 2146 of *Lecture Notes in Computer Science*, Providence, RI, USA, 2001. Springer-Verlag.

[52] William Whyte, editor. *Efficient Embedded Security Standard #1, Version 2.* Consortium for Efficient Embedded Security, 2003. `http://www.ceesstandards.org/`.

A Code

In this appendix, we will shortly introduce the code which we used to gather the results for section 5. A complete code-listing of the programs that we mention will follow.

The first program we used is written in Java (`NTRUSignFileWriter.java`). It is an interface which uses the experimental NTRUSignProvider to generate an NTRUSIGN instance. Afterwards, the program saves all vital information about the instance in a file. The way this information is saved imitates the Python dictionary style, so that the file is easily readable later on.

An exemplary data file `N60_#01 - NTRU.dat` is given next in the code-listing. In it, we can see that all private and public information is saved in an easily accessible way. The two ratios a, c introduced in section 3 are also saved here, as well as the balancing estimate λ and the factor by which it is off for both of the big private keys $\mathbf{F}$ and $\mathbf{G}$.

The next program `runexp.py` is a script written in Python. It ran our experiments by generating lattices from the dat files and using the LaRed scripts written by Christoph Ludwig to reduce them. The way the experiment is run by this script can be manipulated by adding certain key words into the `strategy` string parameter . A small description of these key-words is given in the code.

The lattice-basis generation in the last script uses yet another Python script, namely `bases.py`. This one can generate five different types of bases from a dat-file. These can then be plugged into a reduction experiment with `runexp.py`. Again, a small description of the generated bases is given in the code. The best basis for our experiments was `base5`. This basis is also described in section 5.1.

As a finishing touch, we attach the final Python script `matrix.py`, which is a compilation of convenience functions for matrices. This script is heavily used by both of the other scripts. It is for example able to add all rotations of a vector into a matrix, or to check whether a vector is consistent with a given set of NTRU-parameters.

```java
/*** NTRUSign Provider File-Writer Interface.

package ntru.interfaces;

import java.io.*;

public class NtruSignFileWriter {

    public static void writekeys(String DIR, KeyPair keyPair, String counter) {
        /*** Take an NTRUSign KeyPair and create a Python Dictionary File
             containing all vital informations. ***/

        NtruSignPublicKey publicKey = (NtruSignPublicKey)keyPair.getPublic();
        NtruSignPrivateKey privateKey = (NtruSignPrivateKey)keyPair.getPrivate();

        // Getting all polynomials h,f,g,F,G
        Polynomial h = publicKey.getH();
        Polynomial f = privateKey.getF()[0];
        Polynomial g = privateKey.getFPrime()[0];
        NtruSignAlgorithmParameters parameters = privateKey.getParameters();

        int N = parameters.N();
        int q = parameters.q();
        int df = parameters.df();
        int dg = parameters.dg();

        Polynomial F = h.multiplyRDC(N,q,f);
        Polynomial G = h.multiplyRDC(N,q,g);

        // Get the smallest representatives
        F = F.reduceCoefficientsInto(-q/2,q/2);
        G = G.reduceCoefficientsInto(-q/2,q/2);
        h = h.reduceCoefficientsInto(-q/2,q/2);

        // Create lists of coefficients
        int[] hAllCoeffs = makeAllCoeffs(N,q,h.degrees(),h.coefficients());
        int[] fAllCoeffs = makeAllCoeffs(N,q,f.degrees(),f.coefficients());
        int[] gAllCoeffs = makeAllCoeffs(N,q,g.degrees(),g.coefficients());
        int[] FAllCoeffs = makeAllCoeffs(N,q,F.degrees(),F.coefficients());
        int[] GAllCoeffs = makeAllCoeffs(N,q,G.degrees(),G.coefficients());

        // Calculate extra information a, c, lambda
        int scale = 100;
        double delta_f = (2.0*df+1.0) / N - 1.0 /(N*N);
        double delta_g = (2.0*dg+1.0) / N - 1.0 /(N*N);
        double delta = (delta_f + delta_g) / 2.0;
        double a = (double)N / (double)q;
        double c = Math.sqrt(2.0*Math.PI*Math.E*delta*N*Math.sqrt(N)/(q* Math.sqrt(3)));
        double lambda = Math.sqrt(N / 12.0);

        String dirname = "";
        if (N < 100) dirname = DIR+"N0"+N+"\\";
        else dirname = DIR+"N"+N+"\\";

        /*** Write it all vital data in the Python dictionary style to an ...NTRU.dat file ***/
        try {
            File directory = new File(dirname);
            directory.mkdirs();

            FileWriter FW = new FileWriter(dirname+"N"+N+"_#"+counter+" - NTRU.dat");
            BufferedWriter BW = new BufferedWriter(FW);

            BW.write("{'L': {'lattice type': 'NTRU/SVP',\n");
            BW.write("       'N': "+N+",\n");
            BW.write("       'h': ");
            BW.write("[");
            for(int i=0;i<hAllCoeffs.length;i++) {
                BW.write(String.valueOf(hAllCoeffs[i]));
                if(i<hAllCoeffs.length-1) BW.write(", ");
            }
            BW.write("],\n");
            BW.write("       'q': "+q+",\n");
            BW.write("       'd_f': "+df+",\n");
            BW.write("       'd_g': "+dg+",\n");
            BW.write("       'scale': "+scale+",\n");
            BW.write("      },\n");
            BW.write(" 'a': "+a+",\n");
            BW.write(" 'c': "+c+",\n");
            BW.write(" 'lambda': "+lambda+",\n");
            BW.write(" 'f': ");
            BW.write("[");
            for(int i=0;i<fAllCoeffs.length;i++) {
                BW.write(String.valueOf(fAllCoeffs[i]));
                if(i<fAllCoeffs.length-1) BW.write(", ");
            }
            BW.write("]");
            BW.write(",\n");
            BW.write(" 'delta_f': "+delta_f+",\n");
            BW.write(" 'fNorm': "+f.centeredNorm(N,q)+",\n");
            BW.write(" 'fNormEstimate': "+Math.sqrt(delta_f*N)+",\n");
            BW.write(" 'g': ");
            BW.write("[");
            for(int i=0;i<gAllCoeffs.length;i++) {
                BW.write(String.valueOf(gAllCoeffs[i]));
                if(i<gAllCoeffs.length-1) BW.write(", ");
            }
            BW.write("]");
            BW.write(",\n");
            BW.write(" 'delta_g': "+delta_g+",\n");
            BW.write(" 'gnorm': "+g.centeredNorm(N,q)+",\n");
            BW.write(" 'gNormEstimate': "+Math.sqrt(delta_g*N)+",\n");
            BW.write(" 'F': ");
            BW.write("[");
            for(int i=0;i<FAllCoeffs.length;i++) {
                BW.write(String.valueOf(FAllCoeffs[i]));
                if(i<FAllCoeffs.length-1) BW.write(", ");
            }
            BW.write("]");
            BW.write(",\n");
            BW.write(" 'FNorm': "+F.centeredNorm(N,q)+",\n");
            BW.write(" 'FNormEstimate': "+Math.sqrt(delta_f*N*N/12.0)+",\n");
            BW.write(" 'FNorm / FNormEstimate': "+F.centeredNorm(N,q) / Math.sqrt(delta_f*N*N/12.0)+",\n");
            BW.write(" 'G': ");
            BW.write("[");
            for(int i=0;i<GAllCoeffs.length;i++) {
                BW.write(String.valueOf(GAllCoeffs[i]));
                if(i<GAllCoeffs.length-1) BW.write(", ");
            }
            BW.write("]");
            BW.write(",\n");
            BW.write(" 'GNorm': "+G.centeredNorm(N,q)+",\n");
            BW.write(" 'GNormEstimate': "+Math.sqrt(delta_g*N*N/12.0)+",\n");
            BW.write(" 'GNorm / GNormEstimate': "+G.centeredNorm(N,q) / Math.sqrt(delta_g*N*N/12.0)+",\n");
            BW.write(" 'h': ");
            BW.write("[");
            for(int i=0;i<hAllCoeffs.length;i++) {
                BW.write(String.valueOf(hAllCoeffs[i]));
                if(i<hAllCoeffs.length-1) BW.write(", ");
            }
            BW.write("]");
            BW.write(",\n");
            BW.write(" 'hnorm': "+h.centeredNorm(N,q)+",\n");
            BW.write("}\n");
            BW.close();
        } catch(IOException e) {
            throw new RuntimeException( e );
        }

        /*** Write the NTRU lattice to an ...base.lat file ***/
        int dim = 2*N;
        try {
            FileWriter FW = new FileWriter(dirname+"N"+N+"_#"+counter+" - base.lat");
            BufferedWriter BW = new BufferedWriter(FW);

            // First the dimensions
            BW.write("["+dim+","+dim+"]\n(\n");

            // Then a list of rows
            for(int i=0; i<N;i++) {
                BW.write("(");
                for(int j=0; j<dim;j++) {
                    if(i==j) {BW.write(String.valueOf((int)(lambda*scale)));}
                    else {BW.write("0");}
                    if(j!=dim-1) {BW.write(", ");}
                }
                BW.write("),\n");
            }

            for(int i=0; i<N;i++) {
                BW.write("(");
                for(int j=0; j<dim;j++) {
                    if(j<N) {
                        BW.write(String.valueOf((int)(hAllCoeffs[(i-j+N)%N]*scale)));
                    } else if (i+N == j) {
                        BW.write(String.valueOf((int)(q*scale)));
                    } else {
                        BW.write("0");
                    }
                    if(j!=dim-1) {BW.write(", ");}
                }
                if(i!=N-1) {
                    BW.write("),\n");
                } else {
                    BW.write(")\n");
                }
            }
            BW.write(")\n");

            BW.close();
        } catch(IOException e) {
            throw new RuntimeException( e );
        }

    }

    private static int[] makeAllCoeffs (int N, int q, int[] degs, int[] coeffs) {
        /*** Creates a list of all coefficients from a given
             list of degrees and coefficients at these degrees ***/

        int[] allCoeffs = new int[N];
        for(int i=0;i<degs.length-1;i++)
            allCoeffs[degs[i]] = coeffs[i];
```

```
        return allCoeffs;
    }

    public static void main(String[] args) {

        final int START = 1;                                // Create instances numbering from
        final int END = 30;                                 // START to END
        final String DIR = "c:\\NTRUtrails_weak\\";         // Directory where instances are saved

        /*** Prepare Provider and keyPairGenerator ***/
        Security.addProvider( new NtruSignProvider() );
        try {
            KeyPairGenerator kpg = KeyPairGenerator.getInstance( "NTRUSign" );

            for (int i=START;i<=END;i++) {
                KeyPair keyPair = kpg.generateKeyPair();    // Generate Keys
                if(i<10) NtruSignFileWriter.writekeys(DIR, keyPair, "0"+String.valueOf(i));
                else NtruSignFileWriter.writekeys(DIR, keyPair, String.valueOf(i));
                System.out.println(i+". written");
            }
        } catch ( NoSuchAlgorithmException e ) {
            throw new RuntimeException( e );
        }
    }
}
```



```
{'L': {'lattice type': 'NTRU/SVP',
       'N': 60,
       'h': [-24, 0, 0, 0, 7, 0, 0, 0, -2, 0, 0, 0, -19, 0, 0, 0, 27, 0, 0, 0,
14, 0, 0, 0, -24, 0, 0, 0, 28, 0, 0, 0, -29, 0, 0, 0, 15, 0, 0, 0, 16, 0, 0, 0,
 -4, 0, 0, 0, -12, 0, 0, 0, -7, 0, 0, 0, -18, 0, 0, 0],
       'q': 64,
       'd_f': 1,
       'd_g': 1,
       'scale': 100,
      },
 'a': 0.9375,
 'c': 1.886929079072279,
 'lambda': 2.23606797749979,
 'f': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'delta_f': 0.04972222222222222,
 'fNorm': 1.7272328544032325,
 'fNormEstimate': 1.7272328544042153,
 'g': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'delta_g': 0.04972222222222222,
 'gnorm': 1.7272328544032325,
 'gNormEstimate': 1.7272328544042153,
 'F': [5, 0, 0, 0, -5, 0, 0, 0, -3, 0, 0, 0, -2, 0, 0, 0, -2, 0, 0, 0, -9, 0,
0, 0, -4, 0, 0, 0, 11, 0, 0, 0, -3, 0, 0, 0, 1, 0, 0, 0, -14, 0, 0, 0, 4, 0, 0,
 0, -7, 0, 0, 0, -1, 0, 0, 0, -3, 0, 0, 0],
 'FNorm': 23.429326352529213,
 'FNormEstimate': 3.8622100754188224,
 'FNorm / FNormEstimate': 6.066300355241166,
 'G': [2, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 12, 0, 0, 0, 1, 0, 0, 0, 4, 0, 0, 0,
 10, 0, 0, 0, 4, 0, 0, 0, -2, 0, 0, 0, 2, 0, 0, 0, -8, 0, 0, 0, 2, 0, 0, 0, -1,
 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0],
 'GNorm': 18.73321470899535,
 'GNormEstimate': 3.8622100754188224,
 'GNorm / GNormEstimate': 4.8503873023954815,
 'h': [-24, 0, 0, 0, 7, 0, 0, 0, -2, 0, 0, 0, -19, 0, 0, 0, 27, 0, 0, 0, 14, 0,
 0, 0, -24, 0, 0, 0, 28, 0, 0, 0, -29, 0, 0, 0, 15, 0, 0, 0, 16, 0, 0, 0, -4,
0, 0, 0, -12, 0, 0, 0, -7, 0, 0, 0, -18, 0, 0, 0],
 'hnorm': 71.50477839510681,
}
```

```python
#!/usr/bin/env python


def getnorms(dat):
    """ Get all the f,g and q vector norms form a dat-dictionary.
        The norm we use here is the centered euclidean norm.
        We also return the balancing constant lambda for the bigger of the two key-vectors. """

    from math import sqrt
    from polynomial import poly

    N = dat['L']['N']
    q = dat['L']['q']
    f = poly(dat['f'])
    F = poly(dat['F'])
    g = poly(dat['g'])
    G = poly(dat['G'])
    qpol = poly([q] + [0]*(N-1))

    scale = 100
    l = f
    L = F
    if ((g.meannormsqrd() + G.meannormsqrd()) > (f.meannormsqrd() + F.meannormsqrd())):
        l = g
        L = G

    lam = int( sqrt(L.meannormsqrd() / float(l.meannormsqrd())) * scale )

    fnorm = round(  f.meannormsqrd()*(lam*N)**2 + F.meannormsqrd()*(scale*N)**2 )
    gnorm = round(  g.meannormsqrd()*(lam*N)**2 + G.meannormsqrd()*(scale*N)**2 )
    qnorm = round( 0 + qpol.meannormsqrd()*(scale*N)**2 )

    return { 'fnorm': fnorm,
             'gnorm': gnorm,
             'gnorm': qnorm,
             'lambda': lam, }


def exp(outputdir, iterationdir, info, N, IT, I, strategy):
    """ Runs an experiment in iterationdir according to strategy. This also updates info. """

    from os import linesep, remove
    from os.path import join, isfile
    from math import log

    from io import writefile, readfile, readLog, readMatrix, deldir, writefilewrapped
    from matrix import removeZeros              # removes the zero columns from a matrix
    from plotting import drawNorms, drawLists   # for plotting after every reduction

    import LaRed
    import LaRedExp
    from LaRed.OrthNormModule import OrthNorm           # find the norms of the orthogonalization
    from LaRedExp.BKZModule import BKZ                  # performs BKZ
    from LaRedExp.RSRModule import ShortProjectionSR    # performs ShortProjectionSR
    from LaRedExp.RSRModule import ShortPoolRSR         # performs ShortPoolSR


    BKZargs = info['BKZargs']
    PoolSRargs = info['PoolSRargs']
    ProjSRargs = info['ProjSRargs']

    runtime = 0
    norm_sqrs = []
    orth_norm_sqrs = []

    logfile = join(outputdir, 'logfile')
    resfile = join(outputdir, 'result')


    ################### BKZ ###################
    if info['algo'] == 'BKZ':
        if BKZargs['delta'] < 0.99: info['BKZargs']['delta'] = 0.99    # raise delta to .99
        else: info['BKZargs']['beta'] += 1                             # or increase beta

        print 'Params:', BKZargs

        BKZ( source=info['source'],                                    # run BKZ reduction
             targetdir=outputdir,
             beta = BKZargs['beta'],
             delta = BKZargs['delta'],
             prune = BKZargs['prune'] )

        logdict = readfile(logfile)[0]                                 # get runtime of reduction
        runtime += logdict['runtime']
        info['course'] += ['beta<-%d'%(info['BKZargs']['beta'])]       # update info-course
```



```python
    ################### ProjSR ###################
    if info['algo'] == 'ProjSR':
        print 'Params:', ProjSRargs

        gamma = ProjSRargs['gamma']
        umax = ProjSRargs['umax']
        rerun = True

        while rerun:                                                   # rerun until some runtime
            ShortProjectionSR( source=info['source'],                  # has passed
                               targetdir=outputdir,
                               rsrSearchSpace=LaRed.SearchSpace(ProjSRargs['rank'],umax),
                               beta = BKZargs['beta'],
                               delta = BKZargs['delta'],
                               prune = BKZargs['prune'],
                               reduction_factor= gamma,
                               target_set = ProjSRargs['target_set'],
                               max_runtime = ProjSRargs['max_runtime'],
                               dump_interval = 0 )

            metalog = readLog(logfile)
            for i in range(len(metalog)):
                runtime += metalog[i]['runtime']

            if (runtime <= 1):
                runtime = 0                                            # if runtime is too small,
                if gamma < 0.95: gamma += 0.05                         # we were too greedy.
                else: umax += 1                                        # compensate by increasing
                deldir(outputdir)                                      # gamma or umax
            else:
                rerun = False

        info['course'] += ['Proj;UMax<-%d; Gam<-%1.2f'%(umax, gamma)]  # update info-course
        info['BKZargs']['beta'] -= 1                                   # decrease beta because
                                                                       # severe change has occured
                                                                       # in basis

    ################### PoolSR ###################
    if info['algo'] == 'PoolSR':
        print 'Params:', PoolSRargs

        beta = info['BKZargs']['beta']

        gamma = PoolSRargs['gamma']
        umax = PoolSRargs['umax']
        rerun = True


        while rerun:
            ShortPoolRSR( source=info['source'],
                          targetdir=outputdir,
                          rsrSearchSpace=LaRed.SearchSpace(PoolSRargs['rank'],umax),
                          beta = BKZargs['beta'],
                          delta = BKZargs['delta'],
                          prune = BKZargs['prune'],
                          reduction_factor= gamma,
                          poolsize=PoolSRargs['poolsize'],
                          max_runtime = PoolSRargs['max_runtime'],
                          dump_interval = 0 )

            metalog = readLog(logfile)
            for i in range(len(metalog)):
                runtime += metalog[i]['runtime']

            if (runtime <= 1):
                runtime = 0                                            # if runtime is too small,
                if gamma < 0.95: gamma += 0.05                         # we were too greedy.
                else: umax += 1                                        # compensate by increasing
                deldir(outputdir)                                      # gamma or umax
            else:
                rerun = False

        info['course'] += ['Pool;UMax<-%d; Gam<-%1.2f'%(umax, gamma)]
        info['BKZargs']['beta'] -= 1


    info['runtime'] += runtime                                         # update info-runtime
    info['source'] = resfile                                           # update info-source
    res = readMatrix(resfile)
    norm_sqrs = map(lambda x: float(str(x)), res.norm_sqrs())

    if 0 in norm_sqrs:                                                 # remove zero columns
        removeZeros(res)
        remove(resfile)
        writefile(resfile,res)
        norm_sqrs = map(lambda x: float(str(x)), res.norm_sqrs())

    orth_norm_sqrs = map(lambda x: float(str(x)), OrthNorm(res))       # get orth_norms
```

```
    shortest = min(norm_sqrs)
    shortestindex = norm_sqrs.index(shortest)
    info['shortest'] += ['%f:%d'%(shortest, shortestindex)]            # update info-shortest
    if shortest < info['qnorm']:                                       # mark qpassed and broken
        info['qpassed'] = True
    if (info['fnorm'] in norm_sqrs) or (info['gnorm'] in norm_sqrs):
        info['broken'] = True


    # Calculate the target set for ProjSR

    offset = log(orth_norm_sqrs[0],10)
    slope = (log(orth_norm_sqrs[2*N-3],10)-log(orth_norm_sqrs[N],10)) / N

    lower_set = map(lambda x: 10**(offset+x*slope),range(2*N))         # in log_10 scale
                                                                       # lower set is a line
    target_set = []                                                    # through the first and last
    target_size = 0                                                    # vector in the orth-basis
    target_size_set = []

    for i in range(len(orth_norm_sqrs)):
        difference = log(orth_norm_sqrs[i],10) - log(lower_set[i],10)  # we measure how far
        target_size_set += [abs(difference)]                           # orth-basis diverges from a
        target_size += difference                                      # geo-series
        target_set = [target_size_set.index(max(target_size_set))]     # the result is target_size
        info['ProjSRargs']['target_set'] = target_set
        info['ProjSRargs']['target_size'] = target_size

    f_set = map(lambda x: info['fnorm'], range(2*N))
    g_set = map(lambda x: info['gnorm'], range(2*N))
    q_set = map(lambda x: info['qnorm'], range(2*N))


    # After all calculations are done draw a graph with the information

    orthfile = join(iterationdir, 'plot#%02d.eps'%I)
    if not isfile(orthfile):
        drawLists(orthfile, [lower_set,orth_norm_sqrs,norm_sqrs,q_set,f_set,g_set])


    ################### Strategy ###################

    lastalgo = info['algo']
    print 'Lastalgo:', lastalgo
    nextalgo = 'BKZ'

    if not 'bkz' in strategy:
        if ('pool') in strategy:
            if (abs(target_size) <= info['maxtargetsize']):
                nextalgo = 'PoolSR'
        if ('psr' in strategy) and (BKZargs['beta'] >= BKZargs['minbeta']):
            if (lastalgo=='BKZ') or (lastalgo=='PoolSR'):
                nextalgo = 'ProjSR'
            else: nextalgo = 'BKZ'
            if (abs(target_size) <= info['maxtargetsize']):
                nextalgo = 'PoolSR'

    info['algo'] = nextalgo
    print 'Nextalgo', nextalgo

    info['course'] += ['#%d:%s:%d/%d:%1.4f'%(IT,lastalgo,runtime,info['runtime'],target_size)]
    return info


def performexp(dir, Nrange, Irange, ITrange, strategy):
    """ Performs a lattice reduction experiment with the given parameters.
        The strategy is a string which may contain certain key-words, that
        cause the experiment to follow a special strategy.
        Strategy Key-words:
            bkz         - use only(!) bkz reductions
            psr         - use practical sampling reduction with bkz
            base3/base5 - create a base number 3/5 with createbase script, see bases.py
            smallest    - iterate no further when 3 instances are broken """

    from io import deldir, readfile, writefile, writefilewrapped
    from os import sep, remove
    from os.path import join, isfile, isdir
    from math import log

    from bases import createbase    # see bases.py

    for N in Nrange:
        inputdir = dir + sep + 'N%03d' % N
        strategydir = inputdir + sep + strategy
```



```
        Iblacklist = []
        for IT in ITrange:
            iterationdir =  strategydir + sep + '%03d'%IT
            infos = []
            broken = 0                                  # number of broken instances this iteration
            brokenlist = []
            qpassed = 0                                 # number of iterations past q this iteration
            qpassedlist = []
            runtime = 0                                 # total runtime of this iteration

            for I in Irange:
                if I in Iblacklist: continue

                infofile = join(iterationdir, 'info#%02d'%I)
                if isfile(infofile):
                    newinfo = readfile(infofile)
                    infos.append(newinfo)
                    continue
                outputdir = iterationdir + sep + 'N%03d#%02d'%(N,I)
                if isdir(outputdir): deldir(outputdir)

                info = {}

                # in the first iteration an initial info dictionary is created for each instance
                if IT == 1:
                    datfile = join(inputdir, 'N%d_#%02d - NTRU.dat' % (N, I))
                    dat = readfile(datfile)
                    norms = getnorms(dat)
                    latfile = join(inputdir, 'N%d_#%02d - base.lat' % (N, I))

                    if 'base3' in strategy:
                        latfile = join(inputdir, 'N%d_#%02d - base3.lat' % (N, I))
                        if isfile(latfile): remove(latfile)
                        try:
                            createbase(latfile, dat, 3)
                        except ValueError:
                            print 'h not invertible for N%d_#%02d'%(N, I)
                            Iblacklist += [I]
                            continue

                    if 'base5' in strategy:
                        latfile = join(inputdir, 'N%d_#%02d - base5.lat' % (N, I))
                        if isfile(latfile): remove(latfile)
                        createbase(latfile, dat, 5)


                    # This sets the inital info
                    info = { 'BKZargs': { 'delta': 0.5,          # initial BKZ params
                                          'beta': 2,
                                          'prune': 0,
                                          'minbeta': N/10+2,
                                        },
                             'PoolSRargs': { 'gamma': 0.49,      # initial PoolSR params
                                             'umax': 30,
                                             'poolsize': 1,
                                             'max_runtime': 300,
                                             'rank': 2*N,
                                           },
                             'ProjSRargs': { 'gamma': 0.95,      # initial ProjSR params
                                             'umax': 25,
                                             'target_set': [],
                                             'target_size': 0,
                                             'max_runtime': 300,
                                             'rank': 2*N,
                                           },
                             'maxtargetsize': 4.0,
                             'course': [],       # keeps track of the course of the exp
                             'algo': 'BKZ',      # the next algorithm to be used
                             'qpassed': False,   # set true if q vector has been passed
                             'broken': False,    # set true if instance is broken
                             'source': latfile, # source for next iteration
                             'dat': datfile,
                             'lambda': norms['lambda'], # balancing constant used
                             'fnorm': norms['fnorm'],   # norm of the f key vector
                             'gnorm': norms['gnorm'],   # norm of the g key vector
                             'qnorm': norms['qnorm'],   # norm of the q vector
                             'runtime': 0,              # the total runtime for this instance
                             'shortest': [],            # list of the shortest vectors found
                           }
                    if ('base3' in strategy) or ('base3' in strategy):
                        info['PoolSRargs']['rank'] -= 2
                        info['ProjSRargs']['rank'] -= 2
                else:
                    lastiterationdir = strategydir + sep + '%03d'%(IT-1)
                    lastinfofile = join(lastiterationdir, 'info#%02d'%I)
                    info = readfile(lastinfofile)
```

```
                if info['broken'] == True:
                    info['runtime'] = 0
                    newinfo = info                                          # no reduction
necessary
                else:
                    newinfo = exp(outputdir, iterationdir, info, N, IT, I, strategy)  # the
reduction

                newinfofile = join(iterationdir, 'info#%02d'%I)
                writefilewrapped(newinfofile, newinfo)

                infos.append(newinfo)

            # after all instances are iterated we create a finalinfo containg all results
            finalinfo = {}
            for i in range(len(infos)):
                runtime += infos[i]['runtime']
                if infos[i]['broken'] == True:
                    broken += 1
                    brokenlist.append(i+1)
                if infos[i]['qpassed'] == True:
                    qpassed += 1
                    qpassedlist.append(i+1)
                finalinfo = {'runtime': runtime,
                             'broken': broken,
                             'brokenlist': brokenlist,
                             'qpassed': qpassed,
                             'qpassedlist': qpassedlist,
                             'numofinst': len(Irange),
                            }
            print ''
            print '*************************************************'
            print '***  Iteration %03d for N%03d done in %d seconds.'%(IT,N, runtime)
            print '***  qpassed: %d, broken: %d of %d instances'%(qpassed, broken, len(Irange))
            print '*************************************************'
            print ''
            finalinfofile = join(strategydir, 'info_of_iteration_%03d'%IT)
            writefile(finalinfofile, finalinfo)

            if 'smallest' in strategy:
                if (broken >=3): break     # use this to break once the smallest 3 are found


if __name__ == '__main__':
    """ This is run if the script is executed from the command-line. """

    args = { 'dir': '/home/rlindner/NTRUtrails_weak',  # Directory where experiment takes place
             'Nrange':  range(80, 121, 4),             # The range of N
             'Irange':  range(1, 31),                  # The range of instances
             'ITrange': range(1, 31),                  # The range of iterations
             'strategy': 'pool-base3-smallest', }      # The strategy of the experiment

    performexp(**args)
```



```
#!/usr/bin/env python


def createbase(latfile, dat, i):
    """ Creates different types of bases from a basic NTRU.dat dictionary
        and writes them into latfile.

        Supported types are:    1 - unbalanced NTRU lattice without scaling
                                2 - balanced NTRU lattice with scaling
                                3 - balanced mean h-inverse NTRU lattice scaled with N*scale
                                4 - unbalanced mean NTRU lattice scaled with N
                                5 - balanced mean NTRU lattice scaled with N*scale """

    from os import sep
    from os.path import join
    from math import sqrt,log

    from io import readfile, writefile, readMatrix
    from matrix import addCol, addRotations, completeVector, scaleVector
    from plotting import drawNorms as drawnorms
    from runexp import getnorms

    from polynomial import poly,inverse_mod_ppower
    from LaRed.IntMatrixModule import IntMatrix
    from LaRed.LLLPackage.LLLAlgoModule import LLL

    Ldat = dat['L']
    N = Ldat['N']
    q = Ldat['q']
    r = log(q,2)
    lambd = dat['lambda']
    scale = Ldat['scale']
    lam = int(lambd * scale)

    b = IntMatrix(2*N,0)
    h = poly(Ldat['h'])

    v1 = [1] + [0]*(N-1) + h.coeffs
    v2 = [0]*N + [q]+[0]*(N-1)

    if (i==2):
        scaleVector(v1,lam,scale)
        scaleVector(v2,lam,scale)

    hinv = poly([0]*N)
    if (i==3):
        hinv = inverse_mod_ppower(h, N, 2, r)
        hinv.modNQ(N,q)
        v1 = hinv.coeffs + [1] + [0]*(N-1)
        v2 = [q]+[0]*(2*N-1)

    if (i==3) or (i==4) or (i==5):
        v1lowsum = 0
        for i in v1[:N]: v1lowsum += i
        v1upsum = 0
        for i in v1[N:]: v1upsum += i
        v2lowsum = 0
        for i in v2[:N]: v2lowsum += i
        v2upsum = 0
        for i in v2[N:]: v2upsum += i

        scaleVector(v1,N,N)
        scaleVector(v2,N,N)

        v1 = map(lambda x: x-v1lowsum, v1[:N]) + map(lambda x: x-v1upsum, v1[N:])
        v2 = map(lambda x: x-v2lowsum, v2[:N]) + map(lambda x: x-v2upsum, v2[N:])

        if (i!=4):
            norms = getnorms(dat)

            lam = norms['lambda']
            scaleVector(v1,lam,scale)
            scaleVector(v2,lam,scale)


    addCol(b,v1,0)
    addRotations(b,0)
    addCol(b,v2,N)
    addRotations(b,N)

    writefile(latfile,b)
```

```
#!/usr/bin/env python

################### C O L U M N - operations ###################

def getCol(m,j):
    """ getcol(m, j) returns the j-th column of the matrix m """
    a = []
    for i in range(m.dims[0]):
        a.append(m.get_elem(i,j))
    return a

def addCol(m,v,j):
    """ add a column v to the matrix m at position j """
    m.dims = (m.dims[0], m.dims[1]+1)
    dims = m.dims
    for k in range(dims[1]-1,j,-1):
        for i in range(dims[0]):
            m.set_elem(i, k , m.get_elem(i, k-1))
    for i in range(dims[0]):
        m.set_elem(i, j , v[i])

def removeCol(m,j):
    """ remove a column from a matrix """
    dims = m.dims
    for k in range(j+1, dims[1]):
        for i in range(dims[0]):
            m.set_elem(i, k-1 , m.get_elem(i, k))
    m.dims = (m.dims[0], m.dims[1]-1)

def addCols(m,vlist,j):
    """ adds columns from the list vlist to the matrix m at position j """
    N = len(vlist)
    m.dims = (m.dims[0], m.dims[1]+N)
    dims = m.dims
    for k in range(dims[1]-1,j+1,-1):
        for i in range(dims[0]):
            if (k-N) >= 0:
                m.set_elem(i, k, m.get_elem(i, k-N))
    for k in range(N):
        for i in range(dims[0]):
            m.set_elem(i, j+k , vlist[k][i])


################### R O T A T I O N - operations ###################

def rotate(v,k):
    """ rotate a list v, k times in a clockwise fashion """
    if k == 0: return v
    N = len(v)
    if k > 0:
        v = v[N-k:] + v[:N-k]
    else:
        v = v[-k:] + v[:-k]
    return v

def rotateFrontPart(v,k,N):
    """ rotate the first N entries of a list v, k times in a clockwise fashion
"""
    return rotate(v[:N],k)+v[N:]

def rotateBackPart(v,k,N):
    """ rotate the last N entries of a list v, k times in a clockwise fashion
"""
    M = len(v)-N
    return v[:M]+rotate(v[M:],k)

def getRotations(v):
    """ return a list of all possible rotatations of v """
```



```
    N = len(v)
    list = []
    for i in range(N):
        list.append(rotate(v,i))
    return list

def getPartRotations(v,N):
    """ return a list of all possible combined rotations of the first and last
N entries of v """
    list = []
    for i in range(N):
        list.append(rotateBackPart(rotateFrontPart(v,i,N),i,N))
    return list


################### S E R V I C E - operations ###################

def removeZeros(m):
    """ removes zero columns from the matrix m """
    print 'Removing all zero vectors from basis.'
    list = m.norm_sqrs()
    for i in range(len(list)):
        if str(list[i]) == '0':
            removeCol(m,i)

def addRotations(m,j):
    """ adds all rotations of v at index j """
    print 'Adding all rotations of vector #%d into basis.'%(j+1)
    v = getCol(m,j)
    N = len(v)/2
    rotations = getPartRotations(v,N)
    removeCol(m,j)
    addCols(m,rotations,j)

def addFrontRotations(m,j,k):
    """ adds all rotations of v at index j """
    print 'Adding %d rotations of vector #%d to the front of the basis.'%(k,
j+1)
    v = getCol(m,j)
    N = len(v)/2
    rotations = getPartRotations(v,N)
    removeCol(m,j)
    addCols(m,rotations[:k],0)

def checkVector(v, h, q, scale, lam):
    """ checks whether a vector is consistent with a given polynomial
        in the fashion v[:N]*h = v[N:] mod q """
    from polynomial import poly

    N = len(v)/2
    fcoeffs = v[:N]
    gcoeffs = v[N:]

    for i in range(len(fcoeffs)):
        fcoeffs[i] /= int(lam*scale)
    for i in range(len(fcoeffs)):
        gcoeffs[i] /= scale

    f = poly(fcoeffs)
    g = poly(gcoeffs)
    prod = f * h
    prod.modNQ(N,q)
    g.modNQ(N,q)

    prodcoeffs = prod.coeffs
    gcoeffs = g.coeffs

    result = True
```

```
    for i in range(len(prodcoeffs)):
        result &= (int(str(prodcoeffs[i])) == int(str(gcoeffs[i])))

    return result

def checkMatrix(m, h, q, scale, lam):
    """ prints whether all columns in a matrix are consistent with a given
polynomial """
    dims = m.dims

    for i in range(dims[1]):
        v = getCol(m,i)
        print i, checkVector(v,h,q,scale,lam)

def pushback(m, k):
    """ pushes the first k entries of a matrix to the back """
    dims = m.dims

    for i in range(k):
        v = getCol(m,0)
        removeCol(m,0)
        addCol(m,v,dims[1]-1)

def addq(m, q):
    """ adds a q-vector to the front of the matrix """
    dims = m.dims
    N = dims[0] / 2

    v = []

    for i in range(dims[0]):
        if i == N: v.append(q)
        else: v.append(0)
    addCol(m,v,0)

def completeVector(f, h, q, scale, lam):
    """ completes a polynomial into a lattice vector """
    from polynomial import poly

    fpoly = poly(f)

    hcoeffs = h.coeffs
    N = len(hcoeffs)

    prod = fpoly * h
    prod.modNQ(N,q)

    prodcoeffs = prod.coeffs

    for i in range(len(f)):
        f[i] *= int(lam*scale)
    for i in range(len(prodcoeffs)):
        prodcoeffs[i] *= scale

    return f + prodcoeffs

def scaleVector(v, lam, scale):
    """ scale the first half of a vector with lam and the second with scale """
    N = len(v)/2
    for i in range(N):
        v[i] *= lam
        v[N+i] *= scale
```