University of South-Eastern Norway
Faculty of Technology, Natural Sciences and Maritime Sciences

–

Master's Thesis

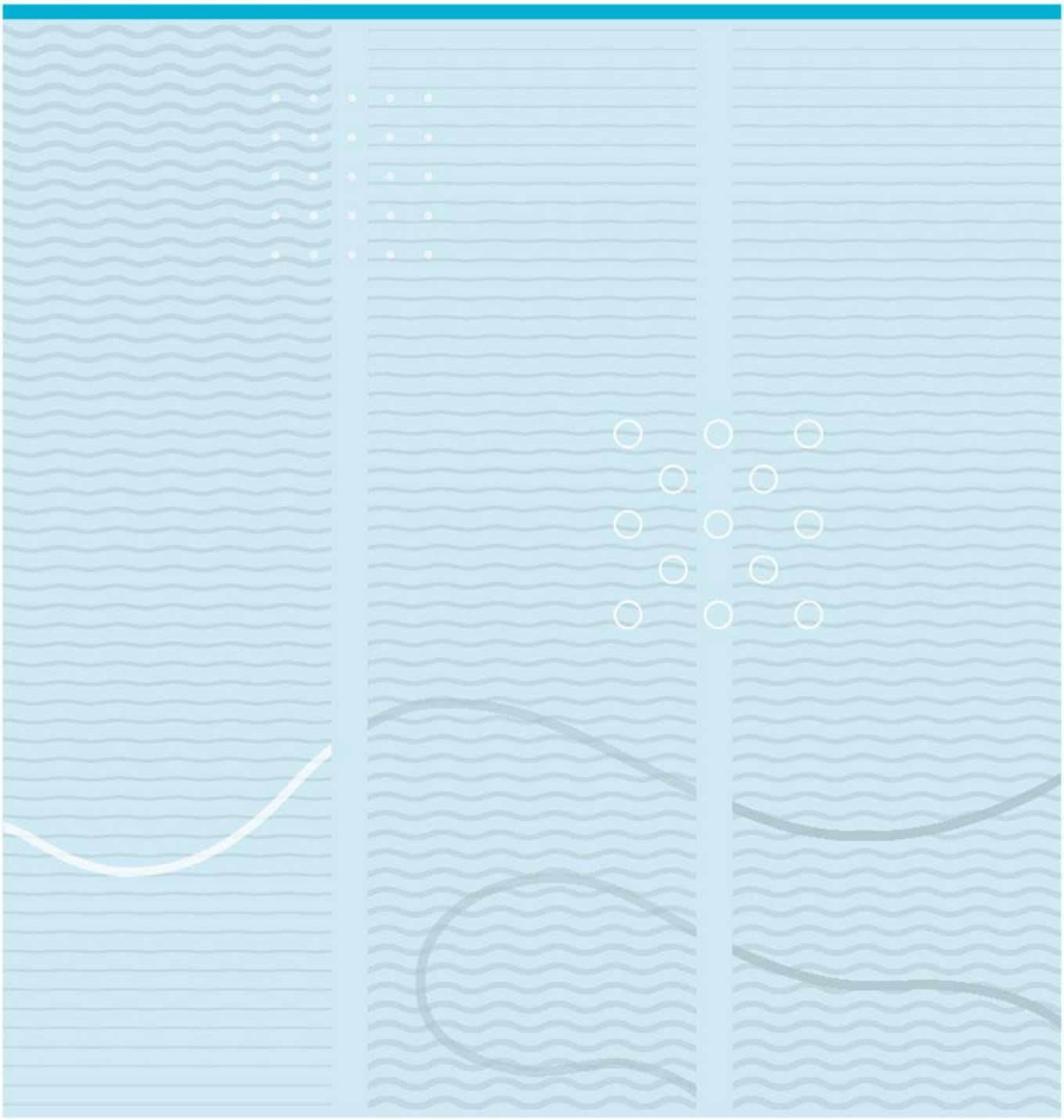
Study programme: Master's level teacher programme in natural sciences and technology

Spring 2023

Daniel Husebye Solheim

Efficiency of attacks on NTRUEncrypt

Comparing data from original presentation to attempts on a modern computer



University of South-Eastern Norway
Faculty of Faculty of Technology, Natural Sciences and Maritime Sciences
Department of Science and Industry Systems
PO Box 235
NO-3603 Kongsberg, Norway

http://www.usn.no

This thesis is worth 30 study points

Comparing efficiency of attacks on NTRUEncrypt between data from original presentation to attempts on a modern computer

Daniel Husebye Solheim

Supervisor: Måns Daniel Larsson

Spring 2023

Contents

Abstract

The motivation of this thesis was to make a comparison of the time needed to successfully break an encryption using NTRUEncrypt from data acquired from a computer in 1998, to data from a computer in 2023. A program was coded to simulate and attack an encryption using the algorithm, and attacks similar to those presented in the article “”NTRU: A ring-based public key cryptosystem” was executed. The results are presented and compared to those in the article, and differences and similarities are discussed.

1 Introduction

A Principle Goal of (Public Key) Cryptography is to allow two people to exchange confidential information, even if they have never met and can communicate only via a channel that is being monitored by an adversary. [1]

Cryptography is the field of study about securing communication from adversarial behavior. [2] It is a field that goes far back. The need to prevent information from leaking to a third party can easily be found in situations regarding politics and war. It is critical that essential information will not be leaked to adversaries in cases where strategies and planning is dependent on an opponent's unawareness. We are now in an era where we also send both open and confidential information through the internet, a medium where it easily can go astray.

A turning point in the history of cryptography is the work of Claude Shannon on information theory. Shannon was a mathematician and engineer in the 20th century that did a large amount of fundamental work on the application of electrical devices on mathematics and he worked a lot on cryptanalysis. In 1948, Shannon and his co-author, Warren Weaver, introduced the concept of information theory in the paper, "A Mathematical Theory of Communication." [3] The once unclear concept of information was now more clearly defined through information theory, which includes the fields of measuring, storing, transferring and securing information. [4] Shannon also reformulated Kerckhoffs principle from the 19th century which states that you need to expect that an opponent always have a full knowledge about how your own cipher-systems works:

Definition (Shannon's Maxim)**.** "One ought to design systems under the assumption that the enemy will ultimately gain full familiarity with them." [4]

Shannon's work made way for a new method of storing and communicating information. Parallel to this, there were a rapid development in the technology of computers which opened for new methods to process and calculate data, and made earlier cryptography too simple to be secure. This lead to the development of several new methods for encryption in the time that followed. These encryptions are conventionally divided into symmetrical ciphers, where all participants have the same private encryption-key, and asymmetrical ciphers, where each participant has their own private encryption-keys that are used to generate and make use of public keys that are shared openly to all who chooses to observe the transfer.

The transfer of the private key in a symmetrical cipher imposes an inherent security risk for the system. The risk may be mitigated by using asymmetrical

ciphers to encrypt the private key for the transfer. Asymmetrical ciphers however, often require longer encryptions than symmetrical ciphers to be secure, so conventionally both are used in the process of sending information.

New technology for computing and processing information are continuously being developed and can open for new methods of cracking contemporary established methods of cryptography. The cryptography systems in use are continually tested for weaknesses and risks, so there is always a need to reinforce and develop new methods of encryption. The concept of quantum computers appeared in the 1980s, and was proven to be able to crack current systems in a considerably shorter time, if the concept was realized. Today, it has already been built working quantum computers, although they can not be used for practical applications yet. However, it is estimated that within 10 years, practical quantum computers should be realized. [4] Because of this, there is a drive in the cryptological community to find quantum-resistant systems.

One of the methods considered is lattice-based cryptography, like the NTRU system. NTRU is a small lattice-based public key encryption system. This means that it is a symmetrical cipher that uses vectors of relatively short length in a lattice. It makes use of the independence of reduction modulo two relatively prime integers with polynomial algebra to gain security with high speed and reasonably short encryption keys. In 1998, the first version of an NTRU encryption system was presented [5] where the system's build was explained, and suggestions to parameters for security standards was argued for based on cryptanalisys of experimental data. The cryptanalisys focused on the generating a lattice matrix from the public key and parameters, and reducing the matrix, using the BKZ-QP1 algorithm, to find the short private keys. The estimates presented for the time require to break the different levels on security ranged from 9 days for the moderate security level, to $6.2 \cdot 10^{27}$ years for the highest security level. However, the processing power of computers have increased considerably since 1998, which presents the question of if you can easily break these security standards, and how much the time required to break them is reduced.

2 Theory

2.1 Basic definitions

Definition 2.1. Let G be a set with an operation $\oplus$. Then the pair $(G, \oplus)$ is a *group* if:

- There exist an identity element, e, such that

  $$e \oplus g = g \oplus e = g$$

  for all $g \in G$.

- For all $a, b, c \in G$, the operation is associative:

  $$a \oplus (b \oplus c) = (a \oplus b) \oplus c.$$

- For all $g \in G$ there exist an inverse, $g^{-1} \in G$ to g such that:

  $$g \oplus g^{-1} = g^{-1} \oplus g = e.$$

A group is *abelian* if:

- For all $a, b \in G$ the operation is commutative:

  $$a \oplus b = b \oplus a$$

Definition 2.2. A *ring* is an abelian group $R = (R, \oplus)$ with an additional operation $\otimes$ such that:

- For all $a, b, c \in R$ the associative law holds:

  $$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

- The distributive laws holds:

  $$a \cdot (b + c) = ab + ac \qquad \text{and} \qquad (b + c) \cdot a = ba + ca$$

- There exist an element of unity for multiplication, I, such that

  $$a \cdot I = I \cdot a = a$$

  for all $a \in R$.

Definition 2.3. An element $a \in R$ is *invertible* if there exist an element $b \in R$ such that

$$ab = ba = I$$

The inverse, b, is usually denoted as a^{-1}. If every element $a \in R$, except 0, is invertible, the ring is called a *field*.

Definition 2.4. An *ideal* of a ring R is a proper subgroup I, such that

$$r \in R,\ a \in I \Rightarrow ra \in I.$$

Definition 2.5. The coset (i.e., an equivalence class) of $a \in R$ under the equivalence relation

$$a \underset{I}{\sim} b \iff a - b \in I, \qquad \text{for all} \qquad a, b \in R,$$

is denoted $a + I$ or sometimes $[a]$ or $\overline{a}$. The set of cosets for a relation is denoted

$$R/I := \{a + I | a \in R\},$$

and is called a *quotient ring*.

Definition 2.6. A set of vectors, $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$, are *linearly independent* if

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + ... + a_n\mathbf{v}_n = 0 \quad \Rightarrow \quad a_1 = a_2 = ... = a_n = 0.$$

In other words, if no vector in the set can be described as a linear combination of other vectors in the set.

Definition 2.7. Let $\mathbf{v}_1, ..., \mathbf{v}_n \in \mathbb{R}^m$ be a set of linearly independent vectors. The **lattice** L generated by $\mathbf{v}_1, ..., \mathbf{v}_n$ is the set of linear combinations of $\mathbf{v}_1, ..., \mathbf{v}_n$ with coefficients in $\mathbb{Z}$,

$$L = \{a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n : a_1, a_2, ..., a_n \in \mathbb{Z}\}.$$

Any two bases for a lattice L are related by a matrix having integer coefficients and determinant equal to ± 1.

Definition 2.8. A subset L of $\mathbb{R}^m$ is an *additive subgroup* if it is closed under addition and subtraction. It is called a *discrete additive subgroup* if there is a positive constant $\epsilon > 0$ with the following property: for every $\mathbf{v} \in L$,

$$L \cap \{\mathbf{w} \in \mathbb{R}^m :|| \mathbf{v} - \mathbf{w} ||< \epsilon\} = \{\mathbf{v}\}.$$

In other words, if you take any vector $\mathbf{v}$ in L and draw a solid ball of radius ϵ around $\mathbf{v}$, then there are no other points of L inside the ball.

Definition 2.9. An *integral* (or *integer*) *lattice* is a lattice all of whose vectors have integer coordinates. Equivalently, an integral lattice is an additive subgroup of $\mathbb{Z}^m$ for some $m \geq 1$.

The group of matrices with integer entries whose inverses also have integer entries are called the *general linear group* over $\mathbb{Z}$ and is denoted by $GL_n(\mathbb{Z})$.

Definition 2.10. Let L be a lattice of dimesion n and let $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$ be a basis for L. The *fundamental domain* (or *fundamental parallelepiped*) for L corresponding to this basis is the set

$$\mathcal{F}(\mathbf{v}_1, ..., \mathbf{v}_n) = \{t_1\mathbf{v}_1 + t_2\mathbf{v}_2 + \cdots + t_n\mathbf{v}_n : 0 \leq t_i < 1\}.$$

Proposition 2.1. *Let $L \subset \mathbb{R}^n$ be a lattice of dimension n and let $\mathcal{F}$ be a fundamental domain for L. Then every vector $\boldsymbol{w} \in \mathbb{R}^n$ can be written in the form*

$$\boldsymbol{w} = \boldsymbol{t} + \boldsymbol{v} \quad \textit{for a unique } \boldsymbol{t} \in \mathcal{F} \textit{ and a unique } \boldsymbol{v} \in L.$$

Equivalently, the union of the translated fundamental domains

$$\mathcal{F} + \boldsymbol{v} = \{\boldsymbol{t} + \boldsymbol{v} : \boldsymbol{t} \in \mathcal{F}\}$$

as $\boldsymbol{v}$ ranges over the vectors in the lattice L exactly covers $\mathbb{R}^n$.

Definition 2.11. Let L be a lattice of dimension n and let $\mathcal{F}$ be a fundamental domain for L. Then the n-dimensional volume of $\mathcal{F}$ is called the *determinant* of L (or sometimes the *covolume* of L). It is denoted by $det(L)$.

Proposition 2.2 (Hadamard's Inequality)**.** *Let L be a lattice, $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n$ be a basis for L, and let $\mathcal{F}$ be a fundamental domain for L. Then*

$$det\ L = Vol(\mathcal{F}) \leq || \boldsymbol{v}_1 || \, || \boldsymbol{v}_2 || \ \cdots || \boldsymbol{v}_n || \, .$$

Proposition 2.3. *Let L be a lattice, $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n$ be a basis for L, and let $\mathcal{F} = \mathcal{F}(\boldsymbol{v}_1, ..., \boldsymbol{v}_n)$ be the associated fundamental domain. Write the coordinate of the i^{th} basis vector as*

$$\boldsymbol{v}_i = (r_{i1}, r_{i2}, ..., r_{in})$$

and use the coordinates of the $\boldsymbol{v}_i$ as the rows of a matrix,

$$F = F(\boldsymbol{v}_1, ..., \boldsymbol{v}_n) = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ r_{21} & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & \cdots & r_{nn} \end{pmatrix}$$

Then the volume of $\mathcal{F}$ is given by the formula

$$Vol(\mathcal{F}(\boldsymbol{v}_1, ..., \boldsymbol{v}_n)) = \mid det(F(\boldsymbol{v}_1, ..., \boldsymbol{v}_n)) \mid \, .$$

All fundamental domains of a lattice has the same volume, thus $det(L)$ is invariant for the lattice L.

2.2 LLL-Algorithm

Definition 2.12 (Shortest Vector Problem (SVP))**.** Find a shortest nonzero vector in a lattice L, i.e., find a nonzero vector $\mathbf{v} \in L$ that minimizes the Euclidean norm $|| \mathbf{v} ||$.

Definition 2.13 (The Closest Vector Problem (CVP))**.** Given a vector $\mathbf{w} \in \mathbb{R}^m$ that is not in L, find a vector $\mathbf{v} \in L$ that is closest to $\mathbf{w}$, i.e., find a vector $\mathbf{v} \in L$ that minimizes the Euclidean norm $|| \mathbf{w} - \mathbf{v} ||$.

Variants are:

- Shortest Basis problem (SBP): Set a requirement on the basis for a lattice about the norm of the vectors.
- Approximate Shortest Vector Problem (apprSVP): Find a vector that is no longer than the shortest vector times a function dependent on the dimension.
- Approximate Closest Vector Problem (apprCVP): Same as apprSVP, but for CVP.

Theorem 2.1 (Hermite's Theorem)**.** *Every lattice L of dimension n contains a nonzero vector $\boldsymbol{v} \in L$ satisfying*

$$|| \boldsymbol{v} ||\leq \sqrt{n}\ det(L)^{\frac{1}{n}}.$$

Definition 2.14. For a given dimension n, *Hermite's constant* γ_n is the smallest value such that every lattice L of dimension n contains a nonzero vector $\mathbf{v} \in L$ satisfying

$$|| \mathbf{v} ||^2\leq \gamma_n\ det(L)^{\frac{2}{n}}.$$

The exact value of γ_n is only known for $1 \leq n \leq 8$ and $n = 24$.

$$\gamma_2^2 = \frac{4}{3}, \quad \gamma_3^3 = 2, \quad \gamma_4^4 = 4, \quad \gamma_5^5 = 8, \quad \gamma_6^6 = \frac{64}{3}, \quad \gamma_7^7 = 64, \quad \gamma_8^8 = 256, \quad \gamma_{24} = 4.$$

For large values of n it is known [1] that Hermite's constant satisfies

$$\frac{n}{2\pi e} \leq \gamma_n \leq \frac{n}{\pi e}.$$

Definition 2.15. We define the *Hadamard ratio of the basis* $\mathcal{B} = \{\mathbf{v}_1, ..., \mathbf{v}_n\}$ to be the quantity

$$\mathcal{H}(\mathcal{B}) = \left(\frac{det\ L}{|| \mathbf{v}_1 || \, || \mathbf{v}_2 || \ldots || \mathbf{v}_n ||} \right)^{\frac{1}{n}}.$$

$0 < \mathcal{H}(\mathcal{B}) \leq 1$. The closer the value is to 1 the more orthogonal are the vectors of the basis.

Definition 2.16. Let L be a lattice of dimension n. The *Gaussian expected shortest length* is

$$\sigma(L) = \sqrt{\frac{n}{2\pi e}} (det\ L)^{\frac{1}{n}}.$$

The *Gaussian heuristic* says that a shortest nonzero vector in a "randomly chosen lattice" will satisfy

$$|| \mathbf{v}_{\text{shortest}} || \approx \sigma(L).$$

2.2.1 Gram-Schmidt Algorithm

Let $b_1, ..., b_n$ be a basis for a vector space $V \subset \mathbb{R}^m$. The following algorithm creates an orthogonal basis b_1^*,...,b_n^* for V:
Set $b_1^* = b_1$.
Loop $i = 2, 3, ..., n$.
Compute $\mu_{ij} = \frac{b_i \cdot b_j^*}{||b_j^*||^2}$ for $1 \leq j < i$.
Set $b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{ij} b_j^*$.
End loop

2.2.2 LLL-Algorithm

for $i = 1, 2, ..., n$;
  $b_i^* := b_i$;
  for $j = 1, 2, ..., i-1$;
    $\mu_{ij} := (b_i, b_j^*)/B_j$;
    $b_i^* := b_i^* - \mu_{ij} b_j^*$
  $B_i := (b_i^*, b_i^*)$
$k := 2$;

(1) perform $(*)$ for $l = k-1$;
if $B_k < (\frac{3}{4} - \mu_{k\,k-1}^2) B_{k-1}$, go to (2);
perform $(*)$ for $l = k-2, k-3, ..., 1$;
if $k = n$, terminate;
$k := k+1$;
go to (1);

(2) $\mu := \mu_{k\,k-1}$; $B := B_k + \mu^2 B_{k-1}$; $\mu_{k\;\;k-1} := \mu B_{k-1}/B$;
$B_k := B_{k-1} B_k / B$; $B_{k-1} := B$;

$$\begin{pmatrix} b_{k-1} \\ b_k \end{pmatrix} := \begin{pmatrix} b_k \\ b_{k-1} \end{pmatrix};$$

$$\begin{pmatrix} \mu_{k-1\,j} \\ \mu_{kj} \end{pmatrix} := \begin{pmatrix} \mu_{kj} \\ \mu_{k-1\,j} \end{pmatrix} \quad \text{for} \quad j = 1, 2, ..., k-2;$$

$$\begin{pmatrix} \mu_{i\,k-1} \\ \mu_{ik} \end{pmatrix} := \begin{pmatrix} 1 & \mu_{k\,k-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -\mu \end{pmatrix} \begin{pmatrix} \mu_{i\,k-1} \\ \mu_{ik} \end{pmatrix} \quad \text{for} \quad i = k+1, k+2, ..., n;$$

if $k > 2$, then $k := k-1$;
go to(1).

$(*)$ If $\mid \mu_{kl} \mid > \frac{1}{2}$, then :

$$\begin{cases} r : \text{integer nearest to } \mu_{kl}; b_k := b_k - r b_l; \\ \mu_{kj} := \mu_{kj} - r\mu_{lj} \quad \text{for} \quad j = 1, 2, ..., l-1; \\ \mu_{kl} := \mu_{kl} - r. \end{cases}$$

2.3 Definitions needed for the NTRU algorithm

Definition 2.17. Fix a positive integer N. The *ring of convolution polynomials* (*of rank* N) is the quotient ring

$$R = \frac{\mathbb{Z}[x]}{(x^N - 1)}.$$

Similarly, the *ring of convolution polynomials* (*modulo* q) is the quotient ring

$$R_q = \frac{(\mathbb{Z}/q\mathbb{Z})[x]}{(x^N - 1)}.$$

In brief, the exponents on the powers of x may be reduced modulo N in R and R_q.
It is often convenient to identify a polynomial

$$\mathbf{a}(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{N-1} x^{N-1} \in R_{(q)}$$

with its vector of coefficients

$$(a_0, a_1, a_2, ..., a_{N-1}) \in \mathbb{Z}^N.$$

Addition of polynomials corresponds to usual addition of vectors, but multiplication is more complicated:

Proposition 2.4. *The product of two polynomials* $\boldsymbol{a}(x), \boldsymbol{b}(x) \in R$ *is given by the formula*

$$\boldsymbol{a}(x) \star \boldsymbol{b}(x) = \boldsymbol{c}(x) \quad \textit{with} \quad c_k = \sum_{i+j \equiv k \pmod N} a_i b_j.$$

The product of the two polynomials in R_q *are also given by the formula, but the value of* c_k *is reduced modulo* q*.*

Definition 2.18. Let $\mathbf{a}(x) \in R_q$. The *centered lift of* $\mathbf{a}(x)$ *to* R is the unique polynomial $\mathbf{a}'(x) \in R$ satisfying

$$\mathbf{a}'(x) \pmod q = \mathbf{a}(x)$$

whose coefficients are chosen in the interval

$$-\frac{q}{2} < a'_i \leq \frac{q}{2}.$$

Proposition 2.5. *Let* q *be prime. Then* $\boldsymbol{a}(x) \in R_q$ *has a multiplicative inverse if and only if*

$$gcd(\boldsymbol{a}(x), x^N - 1) = 1 \quad \textit{in } (\mathbb{Z}/q\mathbb{Z})[x]$$

Definition 2.19. For any positive integers d_1 and d_2, we let

$$\mathcal{T}(d_1, d_2) = \begin{cases} & \mathbf{a}(x) \text{ has } d_1 \text{ coefficients equal to } 1, \\ \mathbf{a}(x) \in R: & \mathbf{a}(x) \text{ has } d_2 \text{ coefficients equal to } -1, \\ & \mathbf{a}(x) \text{ has all other coefficients equal to } 0, \end{cases}$$

Polynomials in $\mathcal{T}(d_1, d_2)$ are called *ternary* (or *trinary*) *polynomials*. They are analogous to *binary polynomials*, which have only 0's and 1's as coefficients.

With N prime and $gcd(N, q) = 1$, elements in $\mathcal{T}(d, d)$ will never have inverses in R_q, so $\mathcal{T}(d, d-1)$ is often used.

Proposition 2.6. *The L^2 norms of $f \in \mathcal{T}(d_1, d_1 - 1)$ and $g \in \mathcal{T}(d_2, d_2)$ are:*

$$\mid f \mid = \sqrt{2d_1 - 1 - N^{-1}} \qquad\qquad \mid g \mid = \sqrt{2d_2}$$

2.4 The NTRU Algorithm

The NTRU algorithm, also referred to as *NTRUEncrypt*, is a public key encryption system that makes use of public parameters to generate private and public keys. The algorithm is presented through describing an encryption and decryption procedure between two individuals, Alice and Bob:

A trusted party choses public parameters (N, p, q, d) with N and p prime, $gcd(p, q) = gcd(N, q) = 1$, and $q > (6d + 1)p$.

Alice:

- Chooses private $\mathbf{g} \in \mathcal{T}(d_g, d_g)$.
- Chooses private $\mathbf{f} \in \mathcal{T}(d_f, d_f - 1)$ that is invertible in R_q and R_p.
- Computes $\mathbf{F}_q$, the inverse of $\mathbf{f}$ in R_q.
- Computes $\mathbf{F}_p$, the inverse of $\mathbf{f}$ in R_p.
- Publishes the public key $\mathbf{h} = \mathbf{F}_q \star \mathbf{g}$.

Bob:

- Chooses plaintext $\mathbf{m} \in R_p$.
- Chooses a random ephemeral key $\mathbf{r} \in \mathcal{T}(d_r, d_r)$.
- Uses Alice's public key $\mathbf{h}$ to compute $\mathbf{e} \equiv p\mathbf{r} \star \mathbf{h} + \mathbf{m} \pmod{q}$.
- Sends ciphertext $\mathbf{e}$ to Alice.

Alice:

- Computes $\mathbf{f} \star \mathbf{e} \equiv p\mathbf{g} \star \mathbf{r} + \mathbf{f} \star \mathbf{m} \pmod{q}$.
- Centerlifts to $\mathbf{a} \in R$ and compute $\mathbf{m} \equiv \mathbf{F}_p \star \mathbf{a} \pmod{p}$.

2.4.1 Cryptanalysis of NTRUEncrypt

The restriction that $q > (6d + 1)p$ set for the NTRU algorithm is necessary for guaranteed successful decryption because of one step. The centerlift from $\mathbf{f} \star \mathbf{e} \in R_q$ to $\mathbf{a} \in R$. The restriction guarantees that the magnitude of every coefficient of $\mathbf{f} \star \mathbf{e}$ is strictly smaller than $q/2$ [1]. Because of this, $\mathbf{a} = \mathbf{f} \star \mathbf{e}$, not just modulo q.

If one finds an approximation $\mathbf{f}'$ of $\mathbf{f}$, it can be used to decrypt $\mathbf{e}$, as long as $\mathbf{f}' \star \mathbf{e}$ is not reduced modulo q:

$$\begin{aligned}
\mathbf{h} &= \mathbf{F}_q \star \mathbf{g} \\
\mathbf{e} &= p\mathbf{r} \star \mathbf{h} + \mathbf{m} \pmod{q} \\
\mathbf{a}_q &= \mathbf{f}' \star \mathbf{e} \equiv \mathbf{f}' \star p\mathbf{r} \star \mathbf{h} + \mathbf{f}' \star \mathbf{m} \pmod{q} \\
&\quad \text{(Centerlift } \mathbf{a}_q \text{ from } R_q \text{ to } \mathbf{a} \in R\text{).} \\
\mathbf{F}'_p \star \mathbf{a} &\equiv \mathbf{F}'_p \star \mathbf{f}' \star \mathbf{m} \equiv \mathbf{m} \pmod{p}
\end{aligned}$$

The restriction on q covers for the worst-case scenario of the coefficients value in $\mathbf{f} \star \mathbf{e}$, so it's possible to successfully decrypt a message, even with a weaker restriction. In this scenario, it becomes a matter of probability for the chance that the coefficients of $\mathbf{f} \star \mathbf{e}$ exceeds $q/2$.

Don Coppersmith and Adi Shamir wrote an article [6] where they study NTRUEncrypt as a lattice problem and makes estimations on the probability that an approximation of **f** can successfully be used for decryption, and to what degree they would fail. They summarize that the norm of **f** should not be too short compared with other vectors in the lattice, as it would be easy to find, but still sufficiently shorter than the other vectors, so that they can not be used to gain insight into an encryption. They also state that such a security would vanish with improvements to techniques for lattice basis reduction.

Another security risk of the NTRUEncrypt is using the same ephemeral key on two different messages, or encrypting the same message using two different ephemeral keys. By looking at the difference between the encrypted messages, you can find a lot of information about the ephemeral keys with relatively few encryptions of the same message:
Let $e_i = pr_i \star h + m$ be the i-th encryption of the same message m. Then

$$(e_i - e_1) \star h^{-1} \equiv p(r_i - r_1) \star h \star h^{-1} + (m - m) \star h^{-1} \equiv p(r_i - r_1) \pmod{q}$$

If sent a moderate amount of times (4 or 5), an attacker will be able to gain enough information to realistically find the message using brute force methods. [1]

2.4.2 NTRU: A ring-based public key cryptosystem

A possible method to break NTRUEncrypt is through calculating an LLL-reduction of a lattice-matrix with the form

$$\left[\begin{array}{cccc|cccc} \alpha & 0 & \cdots & 0 & h_0 & h_1 & \cdots & h_{N_1} \\ 0 & \alpha & \cdots & 0 & h_{N-1} & h_0 & \cdots & h_{N-2} \\ \cdots & \cdots & \ddots & \cdots & \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & \alpha & h_1 & h_2 & \cdots & h_0 \\ \hline 0 & 0 & \cdots & 0 & q & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & q & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots & \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & q \end{array}\right],$$

where h_i is the i-th coefficient in the public key, q is the public parameter, and α is a variable chosen to fit the public parameters. If the reduction algorithm is sufficiently precise, one can find a short vector whose length is close enough to the original vector that it can be used for decryption.

In "NTRU: A ring-based public key cryptosystem," [5] three choices of parameters are presented as different levels of security:

Moderate Security

$$(N, p, q) = (107, 3, 64)$$
$$f \in \mathcal{T}(15, 14) \qquad g \in \mathcal{T}(12, 12) \qquad r \in \mathcal{T}(5, 5)$$

High Security

$$(N, p, q) = (167, 3, 128)$$
$$f \in \mathcal{T}(61, 60) \qquad g \in \mathcal{T}(20, 20) \qquad r \in \mathcal{T}(18, 18)$$

Highest Security

$$(N, p, q) = (503, 3, 256)$$
$$f \in \mathcal{T}(216, 215) \qquad g \in \mathcal{T}(72, 72) \qquad r \in \mathcal{T}(55, 55)$$

The security of the system was tested through the use of an improved version of the LLL-algorithm, called BKZ. The algorithm was applied to a lattice-matrix of the form shown above, where the value of α was adjusted to optimize the efficiency of an attack. When the algorithm is applied to a matrix, there is an option to adjust certain parameters. The article chooses to focus on the "block size" parameter, which appears to have the strongest effect on precision and running time. The article also describes how the ratio c, between the length of the target vectors and the Gaussian expected shortest length can affect the reduction time, executing faster as c becomes smaller. Two versions of BKZ was considered as well: The floating point version, and the QP1-version, where QP1 yielded a higher precision.
The data from the tests presents the running time of the algorithm, the dimension N of the lattice, the block size parameter, and a comparison of the norms of the found vectors to the target vectors. Data is presented from three cases, representing levels of security, with different values of q and c. For each case it is presented running times for multiple attempts as, for increasing values of the dimension, the block size is increased until a sufficiently close approximation of the target vector is found.

2.5 Streamlined NTRU Prime

Parameters:
p: a prime number
q: a prime number
w: a positive integer

Definition 2.20. The rings $\frac{(\mathbb{Z})[x]}{(x^p-x-1)}$, $\frac{(\mathbb{Z}/3\mathbb{Z})[x]}{(x^p-x-1)}$, and the field $\frac{(\mathbb{Z}/q\mathbb{Z})[x]}{(x^p-x-1)}$ are abbreviated as $\mathcal{R}$, $\mathcal{R}/3$, and $\mathcal{R}/q$ respectively. An element of $\mathcal{R}$ is small if all of its coefficients are in $\{-1, 0, 1\}$, and it has *weight* w if it has exactly w nonzero coefficients. A set, $Short$, is defined as the set of all small, weight-w elements of $\mathcal{R}$.

Definition 2.21. The set $Rounded$ is defined as the set of polynomials $r_0 + r_1x + ... + r_{p-1}x^{p-1} \in \mathcal{R}$ where each coefficient r_i is in

$$\{-(q-1)/2, ..., -6, -3, 0, 3, 6, ..., (q-1)/2\}$$

for $q \in 1 + 3\mathbb{Z}$, or in

$$\{-(q+1)/2, ..., -6, -3, 0, 3, 6, ..., (q+1)/2\}$$

for $q \in 2 + 3\mathbb{Z}$.

Definition 2.22. $Round : \mathcal{R}/q \to Rounded$ is defined as: if

$$a_i \in \{-(q-1)/2, .., (q-1)/2\}$$

, and r_i is the element of $3\mathbb{Z}$ that is closest to a_i, then

$$Round(a_0 + a_1x + ... + a_{p-1}x^{p-1} = r_0 + r_1x + ... + r_{p-1}x^{p-1})$$

. Note that $Round(r) = r + e$ for some small $e \in \mathcal{R}$.

2.5.1 The key-generation algorithm:

- Generate a uniform random small element $g \in \mathcal{R}$. Repeat this step until g is invertible in $R/3$.
- Compute $v = 1/g$ in $\mathcal{R}/3$
- Generate a uniform random $f \in Short$
- Compute $h = g/(3f)$ in $\mathcal{R}/q$.
- h is your public key, and (f, v) are your secret keys.

2.5.2 The encryption algorithm:

- Let $r \in Short$ represent a message.
- Compute $hr \in \mathcal{R}/q$.
- Compute $c = Round(hr) \in Rounded$.
- Your encrypted message is c.

2.5.3 The Decryption algorithm:

- Compute $3fc \in \mathcal{R}/q$.
- Lift $3fc$ so that each integer is between $-(q-1)/2$ and $(q-1)/2$, and then reduce modulo 3 to get a polynomial $e \in \mathcal{R}/3$.
- Multiply e by v in $\mathcal{R}/3$.
- Lift ev to a small polynomial $r' \in \mathcal{R}$.
- If r' has weight w, output the message.

2.5.4 Cryptanalysis of Streamlined NTRU Prime

In the same way as NTRUEncrypt, successful decryption of a message sent in Streamlined NTRU Prime is dependent on that the coefficients in a certain step of the decryption does not exceed a value at which they will be reduced modulo q. If one can find an approximation f' of f, such that $3f'c$ is not reduced modulo q, then:

$$\begin{aligned} h = g/(3f) &\Leftrightarrow g = 3hf \\ c = Rounded(hr) &= hr + m, \quad \text{where } e \text{ is small} \\ a_q = 3f'c &= 3f'hr + 3f'e \\ &\text{(Centerlift } a_q \text{ from } R/q \text{ to } a \in R) \\ 1/(3hf') * a &\equiv 3f'hr/(3hf') \equiv r \pmod 3 \end{aligned}$$

In the submission document "NTRU Prime Round 3" [7] the condition for the coefficients of $3fc$ is ensured from the restriction that $q \geq 16w + 1$ through a proved theorem that states the coefficients of $3fc = 3fe + gr$ lies in the interval $[-8w, 8w]$, when f, g, r and m are small.

3 Method

The calculations done in this article are done using the software SageMath 9.5 in a Jupyter setup. The computer that made the calculations had a AMD Ryzen 7 5800X 8-Core Processor with a clock frequency of 3.80 GHz.

3.1 Programming in SageMath

In order to better understand and use the NTRU system, it was attempted to recreate encryption algorithms with SageMath. Three programs were coded. The first program attempts a simple encryption and decryption with the NTUREncrypt algorithm. The second program also attempts a simple encryption and decryption, but with the Streamlined NTRU Prime algorithm. The third program is a modification to the first, where a message is encrypted with NTRUEncrypt and then is attacked using the BKZ-function in SageMath.

3.1.1 Encryption and decryption with NTRUEncrypt

Exact code is shown in Appendix 7.1.

The rings used for the system was defined as quotient rings over integer rings with the appropriate modulo. A ring like this could be applied to a list of coefficients, or to a polynomial of an integer ring. However, to convert from a polynomial from a quotient ring, it was necessary to do a lift on the polynomial first. A general lift was often used to convert between quotient rings, but the algorithm required a centered lift in certain steps to ensure correct decryption, as can be seen in figure 8. To generate ternary polynomials, a list was created with the appropriate number of 1s and (-1)s and then shuffled. The list would represent a vector of the coefficients to the polynomial. The private keys was randomly generated through a defined function, but the message was manually written into the code to more easily confirm the outcome.

NTRUEncrypt uses rings on the form $\frac{\mathbb{Z}[x]/q}{(x^N-1)}$, where q is a power of 2. This presented a challenge, because a function to find an inverse to a non-field in SageMath was not found. The solution [8] was to first find the inverse modulo 2, and then "lifting" the inverse to the inverse modulo the last modulus squared. This is done until you have an inverse modulo a number greater than or the same as q, and then the found inverse is reduced modulo q, which gives the appropriate inverse.

3.1.2 Encryption and decryption with Streamlined NTRU Prime

The code for Streamlined NTRU Prime was written with SageMath online through the website `https://cocalc.com/`. The online version was not as fast as the downloaded version, but it was sufficient since the running time was not considered with Streamlined NTRU Prime. The exact code is shown in Appendix 7.2.

The code was done in a similar way to what was used for NTRUEncrypt, but with different values and requirements for parameters. An important detail is that p now determines the dimension, while the ring earlier defined by the variable p is now held constant to be modulo 3. Another difference is that ternary polynomials are now generated with a specified weight, instead of specific numbers of 1s and (-1)s. The encryption step of the algorithm is fairly simple, as it only rounds the product of the message and public key to the closest multiple of 3. Centered lifts are required in the decryption steps in the same way as with NTRUEncrypt.

3.1.3 Attack on NTRUEncrypt

Exact code is shown in Appendix 7.3

The code for the attack makes use of the code for the normal encryption and decryption, however it was condensed for the purpose of being able to run several attacks with different values of beta, without having to generate new keys for every attempt. The parameters used are adjusted for each attack to both increase the efficiency of the attack and to test different levels of security. The ratio c is taken into consideration when generating the private keys and their lengths in the lattice are made the same, in an attempt to recreate the method used in "NTRU: A ring-based public key cryptosystem." [5] The steps up to the completed encryption was not changed much, as the attack is executed on the public key and the encrypted message. After the encryption, the public key is converted to a list of its coefficients and in the case that the coefficients of the highest powers are 0, they are added to the list.

The lattice-matrix has four parts, were all parts can be seen as rotations of a single respective vector. Because of this, a function to rotate the elements of a list was made and is used to build up the matrix from the public key and parameters of the system. However, a BKZ-function that worked with a non-integer α could not be found, so it was replaced with 1 as this was deemed sufficient enough to execute a working attack. Then a starting block size is determined and the QP1-version of the BKZ-function is applied to the matrix, with the starting and ending time being recorded. If the reduction succeeded, then a sufficiently short vector should be found on the first row in the reduced matrix. The vector is converted to

a polynomial and checked if it is invertible, which would be false for most cases where the reduction fails to find a sufficiently short vector. If it is invertible, then the polynomial is used in an attempt to decrypt the encrypted message and if the result corresponds to the original message, the code will finish and show the final results for the attempted dimension. In the case that the short vector found is not invertible, the block size is increased by two and a new attempt to reduce the matrix is attempted. In the first attempts, the code would stop if an invertible vector was not usable for decryption, however the code was altered as show in figure 28 to just increase the block size instead. The alteration was made for the case representing the highest security shown in table 4 for dimension $N = 90$ and higher, and for the case represented in 5. Because the target ratio c was mistakenly not altered for the attempts shown in table 4, new attempts were made with a correct ratio, shown in table 5.

4 Results

The results in the following tables was produced by generating one encryption and repeatedly applying the BKZ-funcion to the resulting lattice, with increasing block size for each attempt, until a vector usable for decryption was found. When a satisfying vector was found, new attempts were started on a new encryption with a higher dimension N. The tables show the running time for every attempt, as well as a comparison of the norms to the shortest vector found and the original private key. In most cases where the decryption failed, the shortest vector found was a "q-vector", where all elements of the vector were 0 except for one which had the value q. This became less frequent when the parameter q became higher and vectors slightly shorter than the q-vectors were found.

Most attacks were executed with $\mid f \mid = \mid g \mid$, but for the attacks presented in table 6 $d_g = d_f - 3$ and table 7 shows the results of attacks on an encryption with the exact parameters as presented for moderate security, shown in section 2.4.2. The attack with block size 26 was terminated when it was not finished after a week of running the BKZ-function.

The attempts represented in table 1 and 2 were the most rigidly executed, with the block size starting at 4 for every dimension-increase. The following attempts on higher security levels were executed without the block size being reduced when the dimension was increased.

N	Block size	Running time (sec)	Actual total norm	Smallest norm found	Ratio of found to actual
75	4	3.50	6.16	64.00	10.38
75	6	5.59	6.16	64.00	10.38
75	8	6.40	6.16	6.16	1.00
80	4	4.42	6.48	64.00	9.88
80	6	7.30	6.48	64.00	9.88
80	8	9.73	6.48	64.00	9.88
80	10	11.96	6.48	6.48	1.00
85	4	6.59	6.48	64.00	9.88
85	6	7.51	6.48	64.00	9.88
85	8	14.43	6.48	64.00	9.88
85	10	16.62	6.48	64.00	9.88
85	12	21.35	6.48	64.00	9.88
85	14	28.66	6.48	6.48	1.00
90	4	6.99	6.78	64.00	9.44
90	6	9.07	6.78	64.00	9.44
90	8	12.87	6.78	64.00	9.44
90	10	16.27	6.78	64.00	9.44
90	12	20.45	6.78	64.00	9.44
90	14	40.37	6.78	64.00	9.44
90	16	63.78	6.78	64.00	9.44
90	18	61.98	6.78	64.00	9.44
90	20	94.71	6.78	6.78	1.00

Table 1: BKZ-QP1 with $Q = 64$, $c \approx 0.26$, $\delta = 0.99$, and prune $= 0$.

N	Block size	Running time (sec)	Actual total norm	Smallest norm found	Ratio of found to actual
95	4	7.54	7.07	64.00	9.05
95	6	14.24	7.07	64.00	9.05
95	8	19.08	7.07	64.00	9.05
95	10	31.79	7.07	64.00	9.05
95	12	34.58	7.07	64.00	9.05
95	14	50.48	7.07	64.00	9.05
95	16	76.51	7.07	64.00	9.05
95	18	114.64	7.07	64.00	9.05
95	20	120.05	7.07	64.00	9.05
95	22	1144.13	7.07	7.07	1.00
100	4	14.32	7.07	64.00	9.05
100	6	16.85	7.07	64.00	9.05
100	8	20.13	7.07	64.00	9.05
100	10	39.67	7.07	64.00	9.05
100	12	49.38	7.07	64.00	9.05
100	14	73.21	7.07	64.00	9.05
100	16	80.12	7.07	64.00	9.05
100	18	127.55	7.07	64.00	9.05
100	20	199.72	7.07	64.00	9.05
100	22	3218.81	7.07	7.07	1.00

Table 2: BKZ-QP1 with $Q = 64$, $c \approx 0.26$, $\delta = 0.99$, and prune $= 0$.

N	Block size	Running time (sec)	Actual total norm	Smallest norm found	Ratio of found to actual
75	2	2.11	7.87	128.00	16.26
75	4	4.35	7.87	128.00	16.26
75	6	5.94	7.87	7.87	1.00
80	6	8.93	8.12	128.00	15.76
80	8	16.68	8.12	8.12	1.00
85	8	16.30	8.37	128.0	15.29
85	10	21.57	8.37	126.40	15.11
85	10	17.41	8.37	128.00	15.30
85	12	39.30	8.37	8.37	1.00
90	12	38.06	8.60	128.00	14.88
90	14	46.14	8.60	128.00	14.88
90	16	59.78	8.60	8.60	1.00
95	16	61.82	8.83	128.00	14.49
95	18	80.98	8.83	8.83	1.00
100	18	91.40	9.06	128.00	14.14
100	20	216.96	9.06	128.00	14.14
100	22	2365.18	9.06	128.00	14.14
100	24	8703.01	9.06	9.06	1.00

Table 3: BKZ-QP1 with $Q = 128$, $c \approx 0.23$, $\delta = 0.99$, and prune $= 0$.

N	Block size	Running time (sec)	Actual total norm	Smallest norm found	Ratio of found to actual
75	2	2.88	11.05	256.00	23.18
75	4	4.92	11.05	187.92	17.01
75	6	7.51	11.05	11.05	1.00
80	6	8.97	11.40	186.94	16.40
80	8	11.46	11.40	8.94	0.78
80	8	11.47	11.40	11.40	1.00
85	8	13.87	11.74	197.11	16.78
85	10	14.98	11.74	188.54	16.05
85	10	19.61	11.75	177.09	15.07
85	12	30.87	11.75	11.75	1.00
90	12	31.95	12.08	187.41	15.51
90	14	38.18	12.08	176.80	14.63
90	16	51.63	12.08	12.08	1.00
95	16	60.72	12.41	201.11	16.21
95	18	86.99	12.41	12.41	1.00
100	18	119.23	12.73	230.09	18.08
100	20	191.06	12.73	204.17	16.04
100	22	2860.60	12.73	184.21	14.47
100	24	4924.21	12.73	12.73	1.00

Table 4: BKZ-QP1 with $Q = 256$, $c \approx 0.23$, $\delta = 0.99$, and prune $= 0$.

N	Block size	Running time (sec)	Actual total norm	Smallest norm found	Ratio of found to actual
75	2	3.24	8.60	256.00	29.76
75	4	5.22	8.60	8.60	1.00
80	4	6.33	9.06	233.41	25.78
80	6	16.68	9.06	9.06	1.00
85	6	14.45	9.27	9.27	1.00
90	6	12.19	9.49	256.00	26.98
90	8	20.36	9.49	225.25	23.74
90	10	24.00	9.49	9.49	1.00
95	10	27.07	9.70	256.00	26.40
95	12	34.71	9.70	239.29	24.68
95	14	53.87	9.70	9.70	1.00
100	14	60.05	10.10	215.17	21.30
100	16	66.50	10.10	10.10	1.00
108	16	110.98	10.49	256.00	24.41
108	18	138.72	10.49	256.00	24.41
108	20	276.46	10.49	256.00	24.41
108	22	2101.92	10.49	10.49	1.00

Table 5: BKZ-QP1 with $Q = 256$, $c \approx 0.18$, $\delta = 0.99$, and prune $= 0$.

N	Block size	Running time (sec)	Actual total norm	Smallest norm found	Ratio of found to actual
75	2	1.53	6.08	64.00	10.52
75	4	3.97	6.08	64.00	10.52
75	6	5.42	6.08	64.00	10.52
75	8	10.08	6.08	6.08	1.00
80	8	8.09	6.40	64.00	10.00
80	10	11.78	6.40	64.00	10.00
80	12	12.10	6.40	6.40	1.00
85	12	20.50	6.71	64.00	9.54
85	14	32.75	6.71	64.00	9.54
85	16	42.15	6.71	64.00	9.54
85	18	86.60	6.71	6.71	1.00
90	18	63.12	6.71	6.71	1.00
95	18	74.51	7.00	64.00	0.26
95	20	238.11	7.00	64.00	0.26
95	22	1440.97	7.00	64.00	0.26
95	24	2262.33	7.00	7.00	1.00

Table 6: BKZ-QP1 with $Q = 64$, $c \approx 0.26$, $\delta = 0.99$, and prune $= 0$.

N	Block size	Running time (sec)	Actual total norm	Smallest norm found	Ratio of found to actual
107	22	1547.75	7.28	64.00	8.79
107	24	38391.30	7.28	64.00	8.79
107	26	630610.59	7.28	n/a	n/a

Table 7: BKZ-QP1 with $Q = 64$, $c \approx 0.26$, $\delta = 0.99$, and prune $= 0$.

5 Discussion

The running time on a modern computer was a lot faster, as expected. The BKZ-algorithm would finish up to several hundred times faster than what was presented in the article from 1998, as shown in table 8. By itself, this could give an expectation that attacks on the security levels presented in the article could succeed in a short time. However, when attempts on encryptions of higher dimensions was executed, the running time would increase drastically. A part of the reason for this was the need to increase the block size to succeed in finding a suitable vector with the BKZ-function. The attacks in the article would in several cases find a suitable vector with a lower block size than the attempts done for this thesis, especially when $q = 64$. There was found no certain explanation for the difference, but some aspects in the setup of the encryption should be considered:

- The code follows the algorithm in making the generated private keys random. Because the target vectors and the lattices based on these vectors are random, it is natural to cause some differences between attacks, even though they are based on the same parameters. The article also presents results from 2 rounds of attacks with the same parameters to address this point. In the presented results the running time was different for the rounds, and one can also observe that the function sometimes executed faster after an increase in dimension or block size, like what can be seen with $N = 85 - 90$ in table 3.

- The form of the lattices for the attacks presented in this thesis are slightly different from those presented in the article. The value α in the lattice-matrix was replaced with 1. Even though the values of α presented in the article was close to 1, it can be hypothesized that the effect it had on the form of the lattice affected the running time.

The block size necessary to find a usable vector seem to be the requirement that most affected the time needed to break an encryption, as it has an exponential effect on the running time of the BKZ-algorithm. [5] This seem to have become a supportive layer of security for the NTRU system. Comparing the running times, as shown in table 8, the running time of the BKZ-function could be several hundred times faster on a modern computer, but when comparing the fastest attacks in the highest dimensions, the largest ratio found was 69.38. The block size necessary to succeed in an attack is decided by the keys generated from the algorithm, and increases with the dimension of the lattice. It is not affected by the processing power, so although the development of processing power in newer computers have a significant effect on the running time, it is still not enough to become unaffected by the exponential increase from the required block size.

While the differences noted above should be taken into consideration, it is still

interesting to note that the attempt at breaking the encryption generated at moderate level security did not succeed even after 7 days, close to the predicted time from the original article, which was 9 days.

In table 4, the outcome of $N = 80$ with block size 8 stands out. The BKZ-algorithm found a vector smaller than the target vector. The vector was not invertible, so it could not be used for decryption, and the code was reset to generate a new encryption. It was not checked at the time, but because the BKZ-algorithm sorts vectors starting with the smallest, it is possible that a vector that was appropriate for decryption could be found in another row of the reduced matrix.

When attempting to attack encryptions where $q = 256$, the BKZ-function were able to frequently find vectors smaller than q-vectors. Some of the keys from these vectors were invertible, but they failed to fully decrypt the message and was disregarded to find a closer approximation to the target vector. Even though these vectors did not succeed in decrypting the message perfectly, a partial decryption can still hold value in cryptanalysis and the frequent appearance of these vectors may have to be considered in regards to security of NTRUEncrypt in higher dimensions. However, none of the vectors in question from the tables above have a norm close to or lower than $q/4$, so the risk they impose should be negligible. [6]

Q	N	Running time, new (sec)	Running time, original (sec)	Ratio, running time	Block size, new	Block size, original
64	75	6.40	1604	250.63	8	6
64	80	11.96	3406	284.78	10	8
64	85	28.66	5168	180.32	14	16
64	90	94.71	18920	199.77	20	18
64	95	1144.13	62321	54.47	22	22
64	100	3218.81	183307	56.95	22	22
128	75	5.94	3026	509.43	6	8
128	80	16.68	5452	326.86	8	10
128	85	39.30	10689	271.98	12	12
128	90	59.78	20195	337.82	16	16
128	95	80.98	57087	704.95	18	20
128	100	8703.01	109706	12.61	24	20
256	108	2101.92	145634	69.38	22	22

Table 8: Comparison of running time and block size for a selection of attacks from table 1, 2, 3, 5 and tables in "NTRU: A ring-based public key cryptosystem." [5]

6 Conclusion

Attacks on NTRUEncrypt using a modern computer was executed and presented. A comparison was made to the data found in “”NTRU: A ring-based public key cryptosystem.” As expected, there was a significant reduction in running time when executing an attack on a modern computer, however the ratio between running times seemed to grow smaller as the dimension of the lattice increased. An attempt to break an encryption using parameters presented as moderate security was made, but was terminated after running for a week without completing, a time close to the expected running time in the article.

Even though NTRU has developed into newer and more advanced systems, such as NTRU Prime, it is interesting to see that even the starting point of these systems can still withstand more than brute-force attacks after 20 years. With the system being resistant to quantum-computers as well, one can understand that it gets attention in the current cryptoglogcial community.

References

[1] J. H. Silverman, J. Pipher, and J. Hoffstein, *An introduction to mathematical cryptography*, vol. 1. Springer, 2008.

[2] R. L. Rivest, "Cryptography," in *Algorithms and complexity*, pp. 717–755, Elsevier, 1990.

[3] E. M. Guizzo, *The essential message: Claude Shannon and the making of information theory*. PhD thesis, Massachusetts Institute of Technology, 2003.

[4] W. Easttom, *Modern Cryptography: Applied Mathematics for Encryption and Information Security*. Cham: Springer International Publishing AG, 2022.

[5] J. Hoffstein, J. Pipher, and J. H. Silverman, "Ntru: A ring-based public key cryptosystem," in *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*, pp. 267–288, Springer, 2006.

[6] D. Coppersmith and A. Shamir, "Lattice attacks on ntru," in *Advances in Cryptology—EUROCRYPT'97: International Conference on the Theory and Application of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings 16*, pp. 52–61, Springer, 1997.

[7] D. J. Bernstein, B. B. Brumley, M.-S. Chen, C. Chuengsatiansup, T. Lange, A. Marotzke, B.-Y. Peng, N. Tuveri, C. Vredensdaal, and B.-Y. Yang, "Ntru prime: round 3." `https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf`, 2020. Accessed: 2023-03-30.

[8] J. L. (https://math.stackexchange.com/users/11619/jyrki lahtonen), "Polynomial ring over finite field - inverting a polynomial non-prime." Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/2650663 (version: 2018-02-15).

7 Appendices

7.1 Appendix A

Code in Jupyter using SageMath to make a simple encryption and decryption with the NTRU system.

```
In [1]: import numpy as np
        import random
        import time
```

Figure 1: Imported libraries

```
In [2]: #Parameters
        p=3;q=64;N=75
        df = 15; dg= 15; dphi = 5
        assert np.gcd(p,q) == 1, "p and q are not coprime"
        #Defining the rings
        Integers.<z> = ZZ[]; QuotientRing = Integers.quotient(z^N-1)
        RingModp.<y> = Zmod(p)[];QRingModp = RingModp.quotient(y^N-1)
        RingModq.<x> = Zmod(q)[];QRingModq = RingModq.quotient(x^N-1)
        print(QuotientRing); print(QRingModp);print(QRingModq)
```

Figure 2: Defining parameters

```
Univariate Quotient Polynomial Ring in zbar over Integer Ring with modulus z^75 - 1
Univariate Quotient Polynomial Ring in ybar over Ring of integers modulo 3 with modulus y^75
+ 2
Univariate Quotient Polynomial Ring in xbar over Ring of integers modulo 64 with modulus x^7
5 + 63
```

Figure 3: Output from code in figure 2

In [3]:
```
#A function to generate a list that represents the coefficients of a random ternary polynomial.
def GenTernaryCoeff(d1,d2):
    #d1 1s, d2 (-1)s and the rest is 0
    Alist = [0]*(N-(d1+d2))+[1]*d1+[-1]*d2
    random.shuffle(Alist)
    return Alist
```

Figure 4: Defining a function to generate coefficients for a ternary polynomial

In [4]:
```
#A function to find the inverse in a quotient ring over integers modulo a power of 2
def FindInverseMod2q(Coeffs):
    Power = 2 #Staring with finding the inverse for mod 2
    #Z/2 and R/2
    RingMod2.<t> = Zmod(2)[]; QRingMod2 = RingMod2.quotient(t^N-1)
    #Returns false if there is no inverse (for any power of 2)
    if QRingMod2(Coeffs).is_unit() == False:
        return False
    #Finding the inverse mod 2
    polynomial = QuotientRing(Coeffs)
    Inverse = QRingMod2(Coeffs)**(-1)
    #Converts to R = Z/(x^N-1)
    if Power != q:
        Inverse = QuotientRing(Inverse.lift())
    while Power < q:
        #Finding the remainder in R
        Remainder = polynomial*Inverse - 1
        #Calculating the new inverse
        Inverse = Inverse*(1-Remainder)
        #Adjusting to the current power of 2
        Power = Power*Power
    Inverse = QRingModq(Inverse.lift()) #Converting to polynomial in R/q
    assert QRingModq(Coeffs)*Inverse == 1, "Something went wrong with inverse"
    return Inverse
```

Figure 5: Defining a function to find the inverse mod a power of 2

In [5]:
```
#Generating coefficients for the keys and defining the polynomials in relevant rings.
#Letting d2 = d1-1 for a key ensures that it can have an inverse.
gKeyCoeff = GenTernaryCoeff(dg,dg-1); print("Key g is: ", QuotientRing(gKeyCoeff))
FInvModq = False
#f must have an inverse both mod p and mod q
while FInvModq == False or QRingModp(fKeyCoeff).is_unit() == False:
    fKeyCoeff = GenTernaryCoeff(df,df-1); print("Key f is: ", QuotientRing(fKeyCoeff))
    #FindInverseMod2q returns false if there is no inverse
    FInvModq = FindInverseMod2q(fKeyCoeff); print("Inverse of f mod q is: ", FInvModq)
FInvModp = QRingModp(fKeyCoeff)**(-1); print("Inverse of f mod p is: ", FInvModp)
#hKeyModq is the public key
hKeyModq = FInvModq * QRingModq(gKeyCoeff); print("Key h is: ", hKeyModq)
```

Figure 6: Generating private keys f and g, and the public key h, and calculating the necessary inverses

In [6]:
```
#Encrypting
#Write the message as a list of coefficients valid in the message space.
Message = [-1,0,1,0,-1,0,1,0,-1,0,1]; print("The message is: ", Message)
#Generate a random ephemeral key for encryption
EphemeralCoeff = GenTernaryCoeff(dphi,dphi); print("The ephemeral key is: ", QRingModq(EphemeralCoeff))
#Encryption
EncryptedMessage = p*QRingModq(EphemeralCoeff)*hKeyModq + QRingModq(Message)
print("The encrypted message is: ", EncryptedMessage)
```

Figure 7: Encryption of a chosen message using the public key and a generated ephemeral key

In [7]:
```
#Decrypting
#Multiply with the f key mod q
DecryptStep = QRingModq(fKeyCoeff) * EncryptedMessage
#Centered lift
LiftedDecryptStep = [coeff.lift_centered() for coeff in (DecryptStep).lift()]
#Multiply with stored inverse of f key mod p
DecryptFinish = FInvModp * QRingModp(LiftedDecryptStep)
#Centered lift to get coefficients from the message space
DecryptedMessage = [coeff.lift_centered() for coeff in (DecryptFinish).lift()]
print("The decrypted message is: ", DecryptedMessage)
assert DecryptedMessage == Message, "The decryption is not the same"
```

```
The decrypted message is:  [-1, 0, 1, 0, -1, 0, 1, 0, -1, 0, 1]
```

Figure 8: Decryption of the encrypted message using the private key f and its inverse

7.2 Appendix B

Code in cocalc using SageMath to make a simple encryption and decryption with the Streamlined NTRU Prime system.

```
1
2  1  import numpy as np
3  2  import random
4
```

Figure 9: Imported libraries

```
5
6  1  #parameters
7  2  q=37;p=37;w=2;
8  3  assert q.is_prime(), "q"
9  4  #Defining the rings
10 5  Q.<x> = Zmod(q)[];Rq = Q.quotient(x^p-x-1);T.<t> = Zmod(3)[];Rt = T.quotient(t^p-t-1); Z.<z>=ZZ[]; R = Z.quotient(z^p-z-1);Rt;Rq;R
11 6  #x^p-x-1 must be irreducible for a prime p
12 7  while not (x^p-x-1).is_irreducible() or not p.is_prime():
13 8      p = p + 2
14 9      if p.is_prime():
15 10         Q.<x> = Zmod(q)[];Rq = Q.quotient(x^p-x-1);T.<t> = Zmod(3)[];Rt = T.quotient(t^p-t-1); Z.<z>=ZZ[]; R = Z.quotient(z^p-z-1);
16 11     assert p<200, "Over"
17 12 assert p.is_prime(), p
18 13 assert 2*p >= 3*w, "pw"; assert q >= 16*w+1, "qw";
19 14 q,p,w
20
    Univariate Quotient Polynomial Ring in tbar over Ring of integers modulo 3 with modulus t^37 + 2*t + 2
    Univariate Quotient Polynomial Ring in xbar over Ring of integers modulo 37 with modulus x^37 + 36*x + 36
    Univariate Quotient Polynomial Ring in zbar over Integer Ring with modulus z^37 - z - 1
    (37, 37, 2)
```

Figure 10: Defining parameters

```
#Generating coefficients for a random ternary polynomial with weight w.
def GenShortW(w,p):
    #Randomly selecting how many 1s of the weight are positive
    Rand = randint(0,w);
    Alist = [0]*(p-w)+[1]*Rand+[-1]*(w-Rand)
    #The placement of coefficients should also be random
    random.shuffle(Alist)
    return Alist
#Generating coefficients for a random ternary polynomial
def GenShort(p):
    Randp = randint(0,p);
    Randm = randint(0,p-Randp)
    Alist = [0]*(p-Randp-Randm)+[1]*Randp+[-1]*Randm
    random.shuffle(Alist)
    return Alist
```

Figure 11: Defining function to generate random coefficients for ternary polynomial

```
#Generating private key g and and its inverse mod 3
gz = R(GenShort(p))
gt = Rt(gz.lift())
while not gt.is_unit():
    gz = R.random_element();
    gt = Rt(gz.lift())
Gt = gt^(-1);
assert Gt*gt == 1, "Whoa inverted g"
#Generating private key f and public key h in R/q
fl = GenShortW(w,p)
fz = R(fl)
hq = Rq(gz.lift())*Rq(3*fz.lift())^(-1);
assert Rq(3*fz.lift())*hq == Rq(gz.lift()), "Whoa inverted f"
```

Figure 12: Generating private keys f and g, and the public key h, and calculating the necessary inverses

```
#A function to encrypt a polynomial representing a message
def Round(poly):
    #Converting to list
    Rpoly = R([coeff.lift_centered() for coeff in (poly).lift()])
    Rlist = Rpoly.list()
    #Round every element to its nearest multiple of 3
    for i in range(len(Rlist)):
        Rlist[i] = floor(Rlist[i]/3+0.5)*3
    return R(Rlist)
```

```

```

```
#A simple message with mostly 0s
m = [1,0,-1] + [0]*(p-3);mq = Rq(m)
#Using function to encrypt message
c = Round(hq*mq)
```

```

```

Figure 13: Encrypting a simple message in Streamlined NTRU Prime

```
#Multiplying encrypted message c with 3f
eq = Rq(fl)*Rq(3*c.lift())
#Centered lift
ez = [coeff.lift_centered() for coeff in (eq).lift()]
#Reducing modulo 3
et = Rt(ez)
#Multiplying by inverse of g and applying a centered lift
rr = R([coeff.lift_centered() for coeff in (Gt*et).lift()]); rr.list()
```

```
[1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 14: Decryption of the encrypted message using the private keys

7.3 Appendix C

Code in Jupyter using SageMath to execute and record running time of an attack for one dimension of an encryption from the NTRUEncrypt algorithm.

```
In [1]: import numpy as np
        import random
        import time
```

Figure 15: Imported libraries

```
In [2]: #A function to generate a list that represents the
        #coefficients of a random ternary polynomial.
        def GenTernaryCoeff(d1,d2):
            #d1 1s, d2 (-1)s and the rest is 0
            Alist = [0]*(N-(d1+d2))+[1]*d1+[-1]*d2
            random.shuffle(Alist)
            return Alist
        #A function to find the inverse in a quotient ring
```

Figure 16: Defining a function to generate coefficients for a ternary polynomial

```
#A function to find the inverse in a quotient ring over integers modulo a power of 2
def FindInverseMod2q(Coeffs, pwr2):
    Power = 2 #Strating with finding the inverse for mod 2
    #Z/2 and R/2
    RingMod2.<t> = Zmod(2)[]; QRingMod2 = RingMod2.quotient(t^N-1)
    #Returns falase if there is no inverse (for any power of 2)
    if QRingMod2(Coeffs).is_unit() == False:
        return False
    #Finding the inverse mod 2
    polynomial = QuotientRing(Coeffs)
    Inverse = QRingMod2(Coeffs)**(-1)
    #Converts to R = Z/(x^n-1)
    if Power != pwr2:
        Inverse = QuotientRing(Inverse.lift())
    while Power < pwr2:
        #Finding the remainder in R
        Remainder = polynomial*Inverse - 1
        #Calculating the new inverse
        Inverse = Inverse*(1-Remainder)
        #Adjusting to the current power of 2
        Power = Power*Power
    Inverse = QRingModq(Inverse.lift()) #Converting to polynomial in R/q
    assert QRingModq(Coeffs)*Inverse == 1, "Something went wrong with inverse"
    return Inverse
#Functions to express the cryptation keys as a lattice
```

Figure 17: Defining a function to find the inverse mod a power of 2

```
#Functions to express the cryptation keys as a lattice
#A function to shift all elements of a list to the right
def push_right(coeff):
    listcopy=copy(coeff)
    temp = listcopy[-1]
    for i in range(len(listcopy)-1,0,-1):
        listcopy[i]=listcopy[i-1]
    listcopy[0] = temp
    return copy(listcopy)
#A function to make the NTRU matrix of size (2N)^2
def Matrify(p,q,N,h):
    #Making lists of the elements that will fill the matrix
    P = copy(h); Q = [q]; I = [1]; O = [0];
    #Q and I have N elements with all elements 0 except the first
    for i in range(N-1):
        I.append(0); O.append(0); Q.append(0)
    AMatrix = []
    for i in range(N):
        #The upper left of the matrix is an identity matrix
        #The upper right is rotations of coefficients in the public key h
        AMatrix.append(I+P)
        P = push_right(P); I = push_right(I)
    for i in range(N):
        #The lower left of the matrix is a zero matrix
        #The lower right is q times the identity matrix
        AMatrix.append(O+Q)
        Q = push_right(Q)
    return matrix(AMatrix)
#A function to find the L^2 norm of a vector
```

Figure 18: Defining a function to generate a lattice-matrix from input values

```
#A function to find the L^2 norm of a vector
def FindNorm(Coefflist):
    Average = 0
    for i in range(2*N):
        Average += Coefflist[i]
    Average = Average/N
    Norm = 0
    for i in range(2*N):
        Norm += (Coefflist[i] - Average)**2
    Norm = np.sqrt(Norm)
    return Norm
```

Figure 19: Defining a function to calculate the L^2 norm of a vector

```
In [3]: #Parameters#########################################################
        p=3;q=128;N=75
        df = 61; dg= 61; dphi = 18
        ##########################################################################
        assert np.gcd(p,q) == 1, "p and q are not coprime"
        #Calculating norms of f and g, only dependend on df and dg
        fKeyNorm = float(sqrt(2*df-1-N**(-1)));
        gKeyNorm = float(sqrt(2*dg-1-N**(-1)));
        #Calculating ratio of length between
        #target vector and shortest expected vector
        ch = np.sqrt(2*np.pi*np.e*gKeyNorm*fKeyNorm/(N*q));
        #Adjusting df and dg to achieve the wanted ratio
        while 0.235 < ch:
            df -=1; dg -= 1
            fKeyNorm = float(sqrt(2*df-1-N**(-1)));
            gKeyNorm = float(sqrt(2*dg-1-N**(-1)));
            ch = np.sqrt(2*np.pi*np.e*gKeyNorm*fKeyNorm/(N*q));
        print("df and dg are: ", df, dg)
        print("Norm of f: ", fKeyNorm)
        print("Norm of g: ", gKeyNorm)
        print("ch is: ", ch)
        #Defining the rings
        Integers.<z> = ZZ[]; QuotientRing = Integers.quotient(z^N-1)
        RingModp.<y> = Zmod(p)[];QRingModp = RingModp.quotient(y^N-1)
        RingModq.<x> = Zmod(q)[];QRingModq = RingModq.quotient(x^N-1)
        print(QuotientRing); print(QRingModp);print(QRingModq)
```

Figure 20: Defining and adjusting parameters

```
df and dg are:  16 16
Norm of f:  5.5665668653728275
Norm of g:  5.5665668653728275
ch is:  0.2347950781088655
Univariate Quotient Polynomial Ring in zbar over Integer Ring with modulus z^75 - 1
Univariate Quotient Polynomial Ring in ybar over Ring of integers modulo 3 with modulus y^75 + 2
Univariate Quotient Polynomial Ring in xbar over Ring of integers modulo 128 with modulus x^75 + 127
```

Figure 21: Output from code in figure 20

```
In [4]: #Generating coefficients for the keys and defining the polynomials in relevant rings.
        #The g key will have the same length as the f key
        gKeyCoeff = GenTernaryCoeff(dg,dg-1); print("Key g is: ", QuotientRing(gKeyCoeff))
        FInvModq = False
        #FindInverseMod2q returns false if there is no inverse
        while FInvModq == False or QRingModp(fKeyCoeff).is_unit() == False:
            #Letting d2 = d1-1 for the f key ensures that it can have an inverse.
            fKeyCoeff = GenTernaryCoeff(df,df-1); print("Key f is: ", QuotientRing(fKeyCoeff))
            FInvModq = FindInverseMod2q(fKeyCoeff,q); print("Inverse of f mod q is: ", FInvModq)
        FInvModp = QRingModp(fKeyCoeff)**(-1); print("Inverse of f mod p is: ", FInvModp)
        hKeyModq = FInvModq * QRingModq(gKeyCoeff); print("Key h is: ", hKeyModq)
```

Figure 22: Generating private keys f and g, and the public key h, and calculating the necessary inverses

```
In [5]: #Encrypting
        #Write the message as a list of coefficients valid in the message space.
        Message = [-1,0,1,0,-1,0,1,0,-1,0,1]; print("The message is: ", Message)
        #Generate a random ephemeral key for encryption
        EphemeralCoeff = GenTernaryCoeff(dphi,dphi); print("The ephemeral key is: ", QRingModq(EphemeralCoeff))
        #Encryption
        EncryptedMessage = p*QRingModq(EphemeralCoeff)*hKeyModq + QRingModq(Message)
        print("The encrypted message is: ", EncryptedMessage)
```

Figure 23: Encryption of a chosen message using the public key and a generated ephemeral key

```
In [6]: #Converting the h key to a list
hKeyCoeff = Integers(hKeyModq.lift()).coefficients(sparse=false)
#Ensuring that the list is of appropriate length
while len(hKeyCoeff) < N:
    hKeyCoeff = hKeyCoeff + [0]
print(hKeyCoeff)
#Generating a matrix with the public key h, and the public parameters
NTRUMatrix = Matrify(p,q,N,hKeyCoeff); print("The NTRU Matrix: "); print(NTRUMatrix)
```

Figure 24: Generating a lattice-matrix from the public key and the public parameters

```
In [7]: Decryption = False
#Setting starting block-size
BlockSize = 2
while Decryption == False:
    #Recording start time and end time
    starttime = time.time()
    #Reducing the matrix with the QP1-version of the BKZ-function
    LLLReduced = NTRUMatrix.BKZ(algorithm='NTL',fp='qd1', block_size = BlockSize)
    endtime = time.time(); ReductionTime = endtime - starttime
    #Retrieving keys from the first row of the reduced matrix
    AlternatefCoeff = LLLReduced[0,:N].list()
    AlternategCoeff = LLLReduced[0,N:].list()
    #Calculating norms and ratio
    AlternateTotalNorm = FindNorm(AlternatefCoeff+AlternategCoeff)
    ActualTotalNorm = FindNorm(fKeyCoeff+gKeyCoeff)
    Ratio = AlternateTotalNorm/ActualTotalNorm
    #Starts decryption if the found key have an inverse
    if QRingModp(AlternatefCoeff).is_unit():
        AlternateFInvp = QRingModp(AlternatefCoeff)**(-1)
        #Decrypting
        #Multiply with the f key mod q
        AlternateDecryptStep = QRingModq(AlternatefCoeff) * EncryptedMessage
        #Centered lift
```

Figure 25: Applying the BKZ-function, retrieving the found vectors, and calculating data about the norms of the found and target vectors

```
if QRingModp(AlternatefCoeff).is_unit():
    AlternateFInvp = QRingModp(AlternatefCoeff)**(-1)
    #Decrypting
    #Multiply with the f key mod q
    AlternateDecryptStep = QRingModq(AlternatefCoeff) * EncryptedMessage
    #Centered lift
    LiftedAlternateDecryptStep = [coeff.lift_centered() for coeff in (AlternateDecryptStep).lift()]
    #Multiply with stored inverse of f key mod p
    AlternateDecrypt = AlternateFInvp * QRingModp(LiftedAlternateDecryptStep)
    #Centered lift to get coefficients from the message space
    AlternateDecryptedMessage = [coeff.lift_centered() for coeff in (AlternateDecrypt).lift()]
    assert AlternateDecryptedMessage == Message, "The decryption is not the same"
    Decryption = True
#Prints results of the attempt
print("--------------------------------------------------------------------------------")
print("p:", p, "q:", q, "N:", N)
print("df:", df, "dg:", dg, "|f|:", fKeyNorm, "|g|:", gKeyNorm, "ch:", ch)
print("Reduction time:", ReductionTime, "Block size:", BlockSize)
print("Norm found vector:", AlternateTotalNorm, "Norm actual vector:", ActualTotalNorm, "Ratio:", Ratio)
print("The decryption succeeded:", Decryption)
#Increases the block size
BlockSize += 2
#A new attempt is started if no inverse was found.
```

Figure 26: Decrypting with an invertible found key and printing results

```
--------------------------------------------------------------------------------
p: 3 q: 128 N: 75
df: 16 dg: 16 |f|: 5.5665668653728275 |g|: 5.5665668653728275 ch: 0.2347950781088655
Reduction time: 2.1138997077941895 Block size: 2
Norm found vector: 128.0 Norm actual vector: 7.874007874011811 Ratio: 16.256016256024385
The decryption succeeded: False
--------------------------------------------------------------------------------
p: 3 q: 128 N: 75
df: 16 dg: 16 |f|: 5.5665668653728275 |g|: 5.5665668653728275 ch: 0.2347950781088655
Reduction time: 4.34681248664856 Block size: 4
Norm found vector: 128.0 Norm actual vector: 7.874007874011811 Ratio: 16.256016256024385
The decryption succeeded: False
--------------------------------------------------------------------------------
p: 3 q: 128 N: 75
df: 16 dg: 16 |f|: 5.5665668653728275 |g|: 5.5665668653728275 ch: 0.2347950781088655
Reduction time: 5.941127061843872 Block size: 6
Norm found vector: 7.874007874011811 Norm actual vector: 7.874007874011811 Ratio: 1.0
The decryption succeeded: True
```

Figure 27: Output from code in figure 25 and 26

```
#Starts decryption if the found key have an inverse
if QRingModp(AlternatefCoeff).is_unit():
    AlternateFInvp = QRingModp(AlternatefCoeff)**(-1)
    #Decrypting
    #Multiply with the f key mod q
    AlternateDecryptStep = QRingModq(AlternatefCoeff) * EncryptedMessage
    #Centered lift
    LiftedAlternateDecryptStep = [coeff.lift_centered() for coeff in (AlternateDecryptStep).lift()]
    #Multiply with stored inverse of f key mod p
    AlternateDecrypt = AlternateFInvp * QRingModp(LiftedAlternateDecryptStep)
    #Centered lift to get coefficients from the message space
    AlternateDecryptedMessage = [coeff.lift_centered() for coeff in (AlternateDecrypt).lift()]
    #If the decryption succeeded, the code will finish
    if AlternateDecryptedMessage == Message:
        Decryption = True
#Prints results of the attempt
print("--------------------------------------------------------------------------------")
print("p:", p, "q:", q, "N:", N)
print("df:", df, "dg:", dg, "|f|:", fKeyNorm, "|g|:", gKeyNorm, "ch:", ch)
print("Reduction time:", ReductionTime, "Block size:", BlockSize)
print("Norm found vector:", AlternateTotalNorm, "Norm actual vector:", ActualTotalNorm, "Ratio:", Ratio)
print("The decryption succeeded:", Decryption)
#Increases the block size
BlockSize += 2
#A new attempt is started if a working key was not found.
```

Figure 28: Altered form of 26, adapted to consider unusable vectors with inverses