Contents

1 Introduction

2 Mathematical Concepts

2.1 Lattice Methods

Lattices have proven to be a powerful tool in modern cryptanalysis. In this section, we explore several methods that rely on lattice reduction — including techniques that can be applied to weaken RSA under specific conditions. We also take a closer look at the NTRU public key cryptosystem and explain how its security is based on lattice structures.

Definition: Let $v_1, \dots, v_n$ be linearly independent vectors in $\mathbb{R}^n$. This means that any real vector $v \in \mathbb{R}^n$ can be uniquely expressed as a linear combination:

$$v = a_1 v_1 + a_2 v_2 + \cdots + a_n v_n$$

where $a_1, \dots, a_n \in \mathbb{R}$. If we restrict the coefficients to integers, we obtain a lattice. Specifically, the lattice generated by $v_1, \dots, v_n$ is defined as:

$$L = \{ m_1 v_1 + \cdots + m_n v_n \,|\, m_1, \dots, m_n \in \mathbb{Z} \}$$

The set $\{v_1, \dots, v_n\}$ is called a basis of the lattice. A lattice can have infinitely many distinct bases. For example, let $\{v_1, v_2\}$ be a basis in $\mathbb{R}^2$, and let $k \in \mathbb{Z}$. Define new vectors $w_1 = v_1 + k v_2$ and $w_2 = v_2$. Then $\{w_1, w_2\}$ is also a basis of the same lattice.

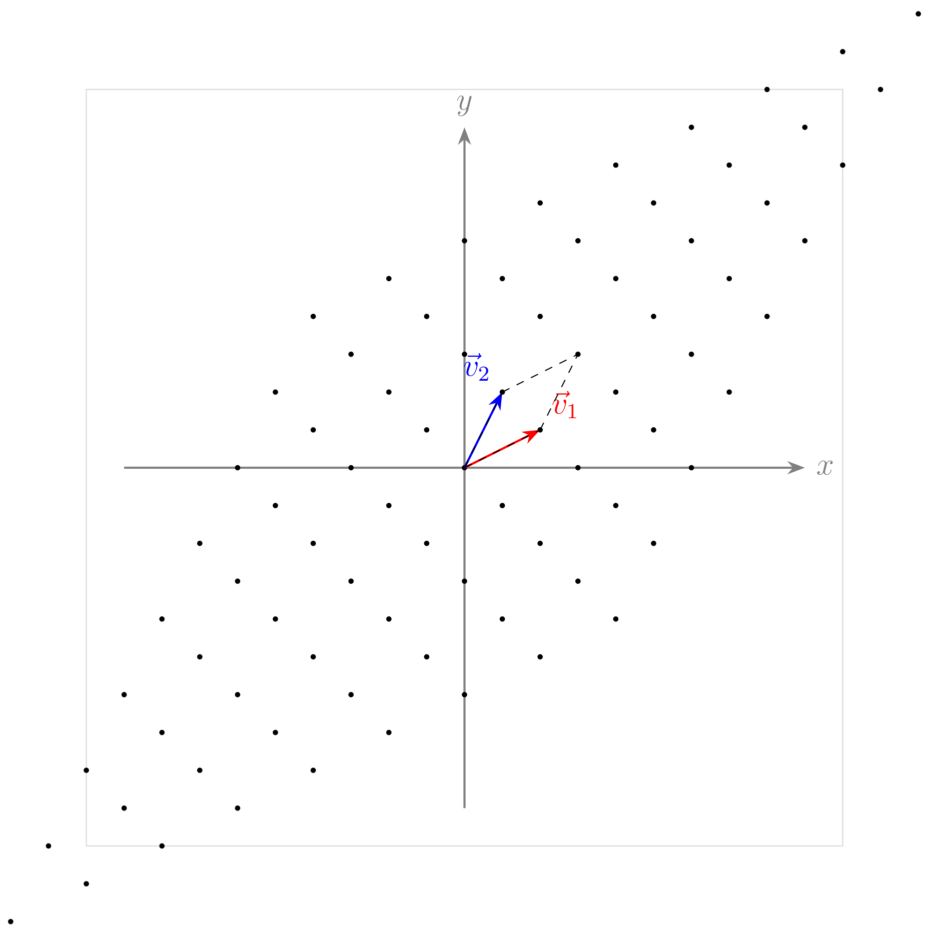

Figure 1. A 2D lattice generated by basis vectors $\vec{v}_1$ and $\vec{v}_2$.

Any integer combination of v_1 and v_2, like $m_1v_1 + m_2v_2$, can also be written using $w_1 = v_1 + kv_2$ and $w_2 = v_2$. To do this, we just substitute $v_1 = w_1 - kw_2$, which gives:

$$m_1v_1 + m_2v_2 = m_1(w_1 - kw_2) + m_2w_2 = m_1w_1 + (m_2 - km_1)w_2$$

So, any vector written using v_1, v_2 can also be written using w_1, w_2, meaning both pairs generate the same lattice.

Example. *Let $\vec{v}_1 = (1, 0)$ and $\vec{v}_2 = (0, 1)$. The lattice generated by these vectors is the set of all integer coordinate pairs (x, y). This means any lattice point is an integer linear combination of $\vec{v}_1$ and $\vec{v}_2$.*

Alternative bases for the same lattice are possible. For example, the set

$$\{(1, 5),\ (0, 1)\}$$

also forms a basis, as does

$$\{(5, 16),\ (6, 19)\}.$$

In general, if a matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

has determinant ± 1, then the pair of vectors (a, b), (c, d) also forms a basis of the same lattice.

The length of a vector $\vec{v} = (x_1, \ldots, x_n)$ is defined as:

$$\|\vec{v}\| = \sqrt{x_1^2 + \cdots + x_n^2}.$$

Many lattice problems reduce to finding a short nonzero vector. In particular, the **Shortest Vector Problem (SVP)** is known to be computationally hard, especially in high dimensions. In the next section, we will look at techniques that are effective in low-dimensional cases.

2.2 Basis Reduction 2D

Let $\vec{v}_1$ and $\vec{v}_2$ be a basis of a two-dimensional lattice. Our goal is to replace this basis with a shorter one, known as a *reduced basis.*

To start, we check the lengths of the vectors. If $\|\vec{v}_1\| > \|\vec{v}_2\|$, we swap them, so that we can assume $\|\vec{v}_1\| \leq \|\vec{v}_2\|$.

Ideally, we would like to replace $\vec{v}_2$ with a vector that is orthogonal to $\vec{v}_1$, similar to what is done in the Gram–Schmidt process in linear algebra.

In the classical Gram–Schmidt procedure, a set of linearly independent vectors $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ is transformed into an orthogonal set $\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_n$ by successively removing projections. For the second vector, this gives:

$$\vec{u}_2 = \vec{v}_2 - \frac{\vec{v}_1 \cdot \vec{v}_2}{\vec{v}_1 \cdot \vec{v}_1}\vec{v}_1$$

Here, $\vec{u}_2$ is orthogonal to $\vec{v}_1$, since the projection of $\vec{v}_2$ onto $\vec{v}_1$ is subtracted.

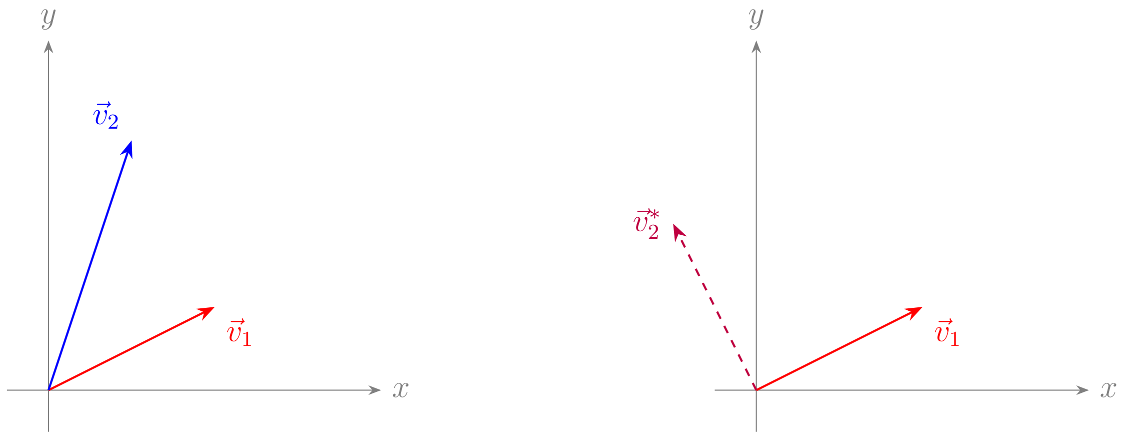

Figure 2. A comparison of original and orthogonalized basis vectors. The left diagram shows the original basis $\vec{v}_1 = (2, 1)$ and $\vec{v}_2 = (1, 3)$, while the right diagram illustrates how Gram–Schmidt orthogonalization replaces $\vec{v}_2$ with $\vec{v}_2^*$, making it orthogonal to $\vec{v}_1$.

In standard Gram–Schmidt orthogonalization, the resulting vectors are not guaranteed to lie within the original lattice, since orthogonal projections may produce non-integer components. To adapt the process for lattice-based algorithms, we modify the orthogonalization step by rounding the projection coefficient to the nearest integer:

$$\vec{v}_2^* = \vec{v}_2 - \left\lfloor \frac{\langle \vec{v}_2, \vec{v}_1 \rangle}{\langle \vec{v}_1, \vec{v}_1 \rangle} \right\rceil \vec{v}_1$$

Here, $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. This adjustment ensures that all resulting vectors remain within the lattice, preserving its discrete structure while achieving partial orthogonality.

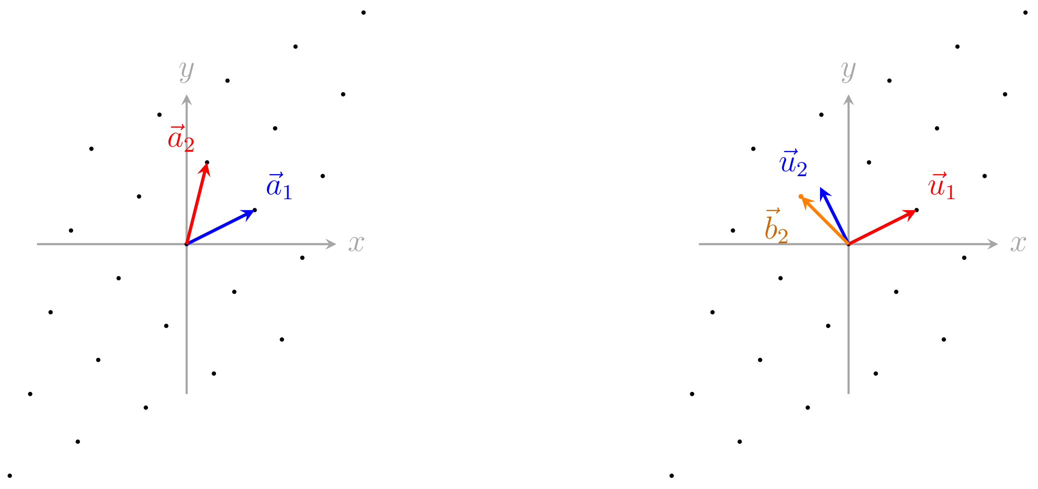

Figure 3. Modified Gram–Schmidt orthogonalization on a lattice basis. Left: original vectors $\vec{a}_1$, $\vec{a}_2$. Right: standard orthogonal vectors $\vec{u}_1$, $\vec{u}_2$, and the lattice-preserving modified vector $\vec{b}_2 = \vec{b}_2 - \text{round}(\mu)\vec{b}_1$.

Note: In lattice-based algorithms (e.g., LLL), we typically aim to find basis vectors forming angles strictly less than 90°, while preserving integrality. Modified orthogonalization helps achieve this within the lattice structure.

Example. *Consider the basis vectors* $\boldsymbol{v}_1 = (31, 59)$ *and* $\boldsymbol{v}_2 = (37, 70)$*. Since*

$$\|\boldsymbol{v}_1\| < \|\boldsymbol{v}_2\|,$$

no swap is required. Compute the projection coefficient:

$$\frac{\boldsymbol{v}_1 \cdot \boldsymbol{v}_2}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1} = \frac{5277}{4442}.$$

Setting $t = 1$*, the updated basis becomes*

$$\boldsymbol{v}_1' = \boldsymbol{v}_1 = (31, 59), \qquad \boldsymbol{v}_2' = \boldsymbol{v}_2 - \boldsymbol{v}_1 = (6, 11).$$

We swap the vectors to maintain order:

$$\boldsymbol{v}_1'' = (6, 11), \qquad \boldsymbol{v}_2'' = (31, 59).$$

Now,

$$\frac{\boldsymbol{v}_1'' \cdot \boldsymbol{v}_2''}{\boldsymbol{v}_1'' \cdot \boldsymbol{v}_1''} = \frac{835}{157},$$

so $t = 5$*, and*

$$\boldsymbol{v}_2^{(3)} = \boldsymbol{v}_2'' - 5\boldsymbol{v}_1'' = (1, 4), \qquad \boldsymbol{v}_1^{(3)} = (6, 11).$$

Renaming:

$$\boldsymbol{v}_1^{(3)} = (1, 4), \qquad \boldsymbol{v}_2^{(3)} = (6, 11).$$

Then,

$$\frac{\boldsymbol{v}_1^{(3)} \cdot \boldsymbol{v}_2^{(3)}}{\boldsymbol{v}_1^{(3)} \cdot \boldsymbol{v}_1^{(3)}} = \frac{50}{17},$$

which gives $t = 3$*. Perform one more reduction:*

$$\boldsymbol{v}_1^* = \boldsymbol{v}_2^{(3)} - 3\boldsymbol{v}_1^{(3)} = (3, -1), \qquad \boldsymbol{v}_2^* = (1, 4).$$

Finally, since $\|\boldsymbol{v}_1^*\| \le \|\boldsymbol{v}_2^*\|$ *and*

$$\frac{\boldsymbol{v}_1^* \cdot \boldsymbol{v}_2^*}{\boldsymbol{v}_1^* \cdot \boldsymbol{v}_1^*} = -\frac{1}{10},$$

the basis $\{\boldsymbol{v}_1^*, \boldsymbol{v}_2^*\}$ *is LLL-reduced.*

Theorem 1. *Let* $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ *be a basis of a two-dimensional lattice in* $\mathbb{R}^2$*. The following procedure produces a reduced basis:*

1. *If* $\|\boldsymbol{v}_1\| > \|\boldsymbol{v}_2\|$*, swap* $\boldsymbol{v}_1$ *and* $\boldsymbol{v}_2$ *so that* $\|\boldsymbol{v}_1\| \le \|\boldsymbol{v}_2\|$*.*
2. *Let* t *be the nearest integer to* $\dfrac{\boldsymbol{v}_1 \cdot \boldsymbol{v}_2}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1}$*.*
3. *If* $t = 0$*, terminate. Otherwise, update* $\boldsymbol{v}_2 \leftarrow \boldsymbol{v}_2 - t\boldsymbol{v}_1$ *and repeat from step 1.*

This algorithm always terminates in finitely many steps and yields a reduced basis $\{\boldsymbol{v}_1', \boldsymbol{v}_2'\}$*. Moreover,* $\boldsymbol{v}_1'$ *is a shortest nonzero vector in the lattice.*

A Python implementation of this reduction method is available on GitHub: https://github.com/SanyaKor/Cryptanalysis/blob/main/lattice_methods/basis_reduction_2d.py

Proof. To establish termination, consider a lattice basis $\{\boldsymbol{v}_1, \boldsymbol{v}_2\} \subset \mathbb{R}^2$, and define

$$\mu = \frac{\boldsymbol{v}_1 \cdot \boldsymbol{v}_2}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1}.$$

Let us define a new vector $\boldsymbol{v}_2^* = \boldsymbol{v}_2 - \mu \boldsymbol{v}_1$, so that

$$\boldsymbol{v}_2 = \boldsymbol{v}_2^* + \mu \boldsymbol{v}_1.$$

Now, select an integer t closest to μ, and define the updated vector

$$\boldsymbol{v}_2' = \boldsymbol{v}_2 - t\boldsymbol{v}_1 = \boldsymbol{v}_2^* + (\mu - t)\boldsymbol{v}_1.$$

Since $\boldsymbol{v}_1 \perp \boldsymbol{v}_2^*$, the squared norm is

$$\|\boldsymbol{v}_2'\|^2 = \|\boldsymbol{v}_2^*\|^2 + (\mu - t)^2 \|\boldsymbol{v}_1\|^2.$$

If $t \neq 0$, then $|\mu - t| \leq \frac{1}{2}$, which implies

$$\|\boldsymbol{v}_2'\|^2 = \|\boldsymbol{v}_2^*\|^2 + (\mu - t)^2 \|\boldsymbol{v}_1\|^2 < \|\boldsymbol{v}_2^*\|^2 + \mu^2 \|\boldsymbol{v}_1\|^2 = \|\boldsymbol{v}_2\|^2.$$

Thus, every update strictly decreases $\|\boldsymbol{v}_2\|$. But there are only finitely many lattice vectors of bounded length, so this process must terminate.

To prove minimality of $\boldsymbol{v}_1$, let $\boldsymbol{w} = a\boldsymbol{v}_1 + b\boldsymbol{v}_2$ be any nonzero lattice vector, where $a, b \in \mathbb{Z}$. Then:

$$\begin{aligned} \|\boldsymbol{w}\|^2 &= (a\boldsymbol{v}_1 + b\boldsymbol{v}_2) \cdot (a\boldsymbol{v}_1 + b\boldsymbol{v}_2) \\ &= a^2 \|\boldsymbol{v}_1\|^2 + b^2 \|\boldsymbol{v}_2\|^2 + 2ab(\boldsymbol{v}_1 \cdot \boldsymbol{v}_2). \end{aligned}$$

Since the basis is reduced, we have

$$-\frac{1}{2}\|\boldsymbol{v}_1\|^2 \leq \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \leq \frac{1}{2}\|\boldsymbol{v}_1\|^2,$$

and hence

$$\|\boldsymbol{w}\|^2 \geq a^2 \|\boldsymbol{v}_1\|^2 - |ab| \|\boldsymbol{v}_1\|^2 + b^2 \|\boldsymbol{v}_2\|^2.$$

Using $\|\boldsymbol{v}_2\|^2 \geq \|\boldsymbol{v}_1\|^2$, we obtain

$$\|\boldsymbol{w}\|^2 \geq (a^2 - |ab| + b^2) \|\boldsymbol{v}_1\|^2.$$

Now note that $a^2 - |ab| + b^2 \geq 1$ for all integers a, b not both zero. Hence,

$$\|\boldsymbol{w}\|^2 \geq \|\boldsymbol{v}_1\|^2,$$

As soon as

$$\boldsymbol{w} = a\boldsymbol{v}_1 + b\boldsymbol{v}_2 \Rightarrow \|a\boldsymbol{v}_1 + b\boldsymbol{v}_2\|^2 \geq \|\boldsymbol{v}_1\|^2$$

which proves that $\boldsymbol{v}_1$ is the shortest nonzero lattice vector.

The two-dimensional case serves as a simple yet insightful example of lattice basis reduction. To extend these ideas to higher dimensions, we introduce the Lenstra–Lenstra–Lovász (LLL) algorithm—a general and efficient method for reducing arbitrary lattice bases while preserving polynomial complexity.

2.3 The LLL algorithm

Lattice reduction in two dimensions provides useful intuition, but generalizing these ideas to higher dimensions is significantly more complex. To address this, the LLL algorithm was introduced by A. Lenstra, H. Lenstra, and L. Lovász. It extends the principles of two-dimensional reduction to arbitrary dimension, offering an efficient way to find relatively short, nearly orthogonal basis vectors.

Theorem 2. *Let L be an n-dimensional lattice generated by vectors $\mathbf{v}_1, \dots, \mathbf{v}_n$ in $\mathbb{R}^n$. Define the determinant of the lattice as*

$$D = |\det(\mathbf{v}_1, \dots, \mathbf{v}_n)|.$$

Let λ be the length of the shortest nonzero vector in L. Then the LLL algorithm produces a basis $\{\mathbf{b}_1, \dots, \mathbf{b}_n\}$ satisfying:

1. $\|\mathbf{b}_1\| \leq 2^{(n-1)/4} D^{1/n}$,
2. $\|\mathbf{b}_1\| \leq 2^{(n-1)/2} \lambda$,
3. $\|\mathbf{b}_1\| \cdot \|\mathbf{b}_2\| \cdots \|\mathbf{b}_n\| \leq 2^{n(n-1)/4} D$.

- **Condition (2):** The first basis vector is close in length to the shortest nonzero vector in the lattice. This means it gives a good approximation of the shortest possible vector.
- **Condition (3):** The basis vectors are nearly orthogonal. In practice, this means that the product of their lengths is close to the volume D of the fundamental parallelepiped, so the basis is well-balanced and spread out.

Example. *Let's consider the lattice generated by the vectors $(31, 59)$ and $(37, 70)$, which was previously used in the two-dimensional reduction. The LLL algorithm gives the same reduced basis:*

$$b_1 = (3, -1), \quad b_2 = (1, 4).$$

We compute the determinant as $D = 13$, and the shortest vector length as $\lambda = \sqrt{10}$ (e.g., from $\|(3, -1)\|$).

We verify the three LLL conditions:

1. $\|b_1\| = \sqrt{10} \leq 2^{1/4}\sqrt{13}$
2. $\|b_1\| = \sqrt{10} \leq 2^{1/2}\sqrt{10}$
3. $\|b_1\| \cdot \|b_2\| = \sqrt{10} \cdot \sqrt{17} \leq 2^{1/2} \cdot 13$

Thus, the LLL conditions hold for this reduced basis.

The LLL algorithm below is based on Wikipedia and implemented by the author: GitHub.

Algorithm 1 Lenstra–Lenstra–Lovász (LLL) Lattice Basis Reduction

Input: A basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\} \subset \mathbb{Z}^m$, reduction parameter $\delta \in (1/4, 1)$ (typically $\delta = 3/4$)
Output: An LLL-reduced basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$

```
1: Compute the Gram-Schmidt orthogonalization $\{\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*\}$ of $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$
2: for $i = 1$ to $n$ do
3:     for $j = 1$ to $i - 1$ do
4:         $\mu_{i,j} \leftarrow \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$
5: $k \leftarrow 2$
6: while $k \leq n$ do
7:     for $j = k - 1$ to 1 by $-1$ do
8:         if $|\mu_{k,j}| > \frac{1}{2}$ then
9:             $\mathbf{b}_k \leftarrow \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \cdot \mathbf{b}_j$
10:            Recompute Gram-Schmidt orthogonalization and $\mu_{i,j}$ as needed
11:    if $\|\mathbf{b}_k^*\|^2 \geq (\delta - \mu_{k,k-1}^2)\|\mathbf{b}_{k-1}^*\|^2$ then
12:        $k \leftarrow k + 1$
13:    else
14:        Swap $\mathbf{b}_k \leftrightarrow \mathbf{b}_{k-1}$
15:        Recompute Gram-Schmidt orthogonalization and $\mu_{i,j}$ as needed
16:        $k \leftarrow \max(k - 1, 2)$
17: return $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$                ▷ The LLL-reduced basis
```

Complexity. The running time of the LLL algorithm is bounded by a constant times $n^6 \log^3 B$, where n is the lattice dimension and B bounds the length of the initial basis vectors. In practice, the algorithm performs much faster than this worst-case estimate. This suggests that LLL is efficient for small to moderate dimensions but may become impractical as n grows large.

2.4 LLL Attack on RSA

We now consider a scenario demonstrating how the LLL algorithm can be applied to attack RSA under specific weak key conditions.

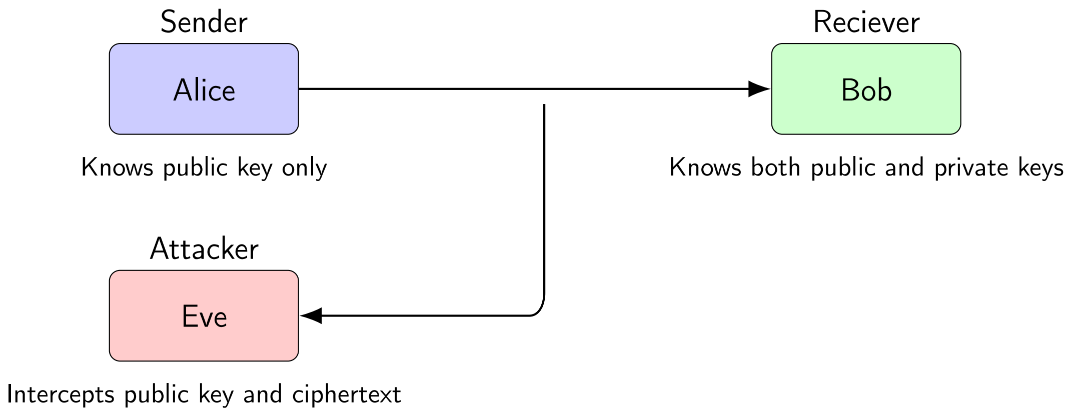
Sender
Alice
Knows public key only
Reciever
Bob
Knows both public and private keys
Attacker
Eve
Intercepts public key and ciphertext

Figure 4. Visualization of the LLL-based attack on RSA , where Eve intercepts the message sent from Alice to Bob.

Alice wants to send Bob a message of the form

*The answer is ***

or

*The password for your new account is ********.*

In these cases, the message is of the form

$$m = B + x, \quad \text{where } B \text{ is fixed and } |x| \leq Y$$

for some integer Y. We'll present an attack that works when the encryption exponent is small.

Suppose Bob has public RSA key $(n, e) = (n, 3)$. Then the ciphertext is

$$c \equiv (B + x)^3 \pmod{n}.$$

We assume that Eve knows B, Y, and n, so she only needs to find x. She forms the polynomial

$$f(T) = (B+T)^3 - c = T^3 + 3BT^2 + 3B^2T + B^3 - c \equiv T^3 + a_2T^2 + a_1T + a_0 \pmod{n}.$$

Eve is looking for $|x| \leq Y$ such that $f(x) \equiv 0 \pmod{n}$. In other words, she is looking for a small solution to a polynomial congruence $f(T) \equiv 0 \pmod{n}$.

Eve applies the *LLL* algorithm to the lattice generated by the vectors

$$\begin{aligned} v_1 &= (n, 0, 0, 0), \\ v_2 &= (0, Yn, 0, 0), \\ v_3 &= (0, 0, Y^2n, 0), \\ v_4 &= (a_0, a_1Y, a_2Y^2, Y^3). \end{aligned}$$

This yields a new basis $b_1, \dots, b_4$, but we need only b_1. The theorem in Subsection 2.3 tells us that

$$\|b_1\| \leq 2^{3/4} \det(v_1, \dots, v_4)^{1/4} = 2^{3/4}(n^3Y^6)^{1/4} = 2^{3/4}n^{3/4}Y^{3/2}. \tag{17.3}$$

We can write

$$b_1 = c_1v_1 + \cdots + c_4v_4 = (e_0, Ye_1, Y^2e_2, Y^3e_3)$$

with integers c_i and with

$$\begin{aligned} e_0 &= c_1n + c_4a_0, \\ e_1 &= c_2n + c_4a_1, \\ e_2 &= c_3n + c_4a_2, \\ e_3 &= c_4. \end{aligned}$$

We observe that the coefficients e_i of the polynomial $g(T)$ satisfy

$$e_i \equiv c_4a_i \pmod{n}, \quad \text{for } i = 0, 1, 2.$$

Now, define the polynomial

$$g(T) = e_3T^3 + e_2T^2 + e_1T + e_0.$$

Since x is a root of $f(T)$ modulo n, and $g(T) \equiv c_4 f(T) \mod n$, we have:

$$g(x) \equiv 0 \pmod{n}.$$

Assume that

$$Y < 2^{-7/6}n^{1/6}. \tag{17.4}$$

We now estimate the size of $g(x)$:

$$\begin{aligned} |g(x)| &\le |e_0| + |e_1x| + |e_2x^2| + |e_3x^3| \\ &\le |e_0| + |e_1|Y + |e_2|Y^2 + |e_3|Y^3 \\ &= \langle (1,1,1,1), (|e_0|, |e_1Y|, |e_2Y^2|, |e_3Y^3|) \rangle \\ &\le \|(1,1,1,1)\| \cdot \|(|e_0|, |e_1Y|, |e_2Y^2|, |e_3Y^3|)\| \\ &= 2\|b_1\|, \end{aligned}$$

where we used the Cauchy–Schwarz inequality in the last step.

Using the bound from inequality (17.3) and our assumption (17.4), we get

$$\|b_1\| \le 2^{3/4}n^{3/4}Y^{3/2} < 2^{3/4}n^{3/4}\left(2^{-7/6}n^{1/6}\right)^{3/2} = \frac{n}{2}.$$

Therefore,

$$|g(x)| < n.$$

Given method transforms the modular equation $f(T) \equiv 0 \pmod{n}$ into an exact polynomial equation $g(T) = 0$, which can be solved numerically. At most three candidates for x are tested to recover the plaintext. More generally, small roots of modular equations can be found using LLL on a lattice of dimension $d+1$. Coppersmith's method extends this to polynomials of degree d, allowing recovery of x when $|x| \le n^{1/d}$, in polynomial time.

This attack method is based on the description from the book *Introduction to Cryptography with Coding Theory* by Trappe and Washington (2nd Edition, Pearson, 2006). An example implementation of this attack has been developed and published by the author on GitHub: github.com/SanyaKor/Cryptanalysis/blob/main/notebooks/lll_attack.ipynb.

2.5 NTRU

Lattice-based cryptography becomes especially powerful in high-dimensional settings. While traditional reduction techniques like the LLL algorithm lose their effectiveness as the dimension n increases (typically beyond $n \geq 100$), this limitation opens the door for new constructions that rely on the hardness of certain lattice problems.

One of the most prominent lattice-based cryptosystems is NTRU, a public-key encryption scheme known for its efficiency and resistance to quantum attacks. Although its original formulation did not explicitly reference lattices, it can be naturally interpreted through a lattice framework—a perspective we will explore in this chapter.

Preliminaries for NTRU

Before presenting the structure of the NTRU cryptosystem, we introduce a few algebraic preliminaries.

We work with polynomials of degree less than N, with coefficients in the ring $\mathbb{Z}_q$. Let

$$f(x) = \sum_{i=0}^{N-1} a_i x^i, \qquad g(x) = \sum_{i=0}^{N-1} b_i x^i$$

be two such polynomials. Their ordinary (non-reduced) product over $\mathbb{Z}$ is given by:

$$(f * g)(x) = \sum_{k=0}^{2N-2} \left(\sum_{i=0}^{k} a_i b_{k-i} \right) x^k.$$

However, NTRU operates in the quotient ring $\mathbb{Z}_q[x]/(x^N - 1)$, where coefficients are reduced modulo q, and the exponents wrap around modulo N. In this ring, polynomial multiplication becomes

$$(f * g)(x) = \sum_{i=0}^{N-1} c_i x^i, \qquad \text{where } c_i = \sum_{j+k \equiv i \pmod{N}} a_j b_k \pmod{q}.$$

This cyclic convolution is fundamental to the design and efficiency of the NTRU cryptosystem.

Example. *Let $N = 3$, and consider the polynomials*

$$f(x) = x^2 + 7x + 9, \qquad g(x) = 3x^2 + 2x + 5.$$

We compute their cyclic convolution product $f * g$ in $\mathbb{Z}[x]/(x^3 - 1)$. The coefficient c_1 of x is

$$c_1 = a_0 b_1 + a_1 b_0 + a_2 b_2 = 9 \cdot 2 + 7 \cdot 5 + 1 \cdot 3 = 56,$$

and the full product is

$$f * g = 46x^2 + 56x + 68.$$

An implementation of this polynomial multiplication is available on GitHub: `github.com/SanyaKor/Cryptanalysis/blob/main/lattice_methods/ntru.py`

Addition. Given two polynomials

$$f(x) = \sum_{i=0}^{N-1} a_i x^i, \quad g(x) = \sum_{i=0}^{N-1} b_i x^i,$$

their sum in $\mathbb{Z}_q[x]/(x^N - 1)$ is defined coefficient-wise:

$$(f+g)(x) = \sum_{i=0}^{N-1} (a_i + b_i \bmod q)\, x^i.$$

This operation is straightforward and preserves the ring structure.

NTRU works with a specific subset of polynomials that have small integer coefficients. To formalize this, we define a family of sets:

$$\mathcal{L}(j,k) = \begin{cases} \text{the set of polynomials of degree } < N \text{ with} \\ j \text{ coefficients equal to } +1,\ k \text{ coefficients equal to } -1, \\ \text{and all other coefficients equal to } 0. \end{cases}$$

The remaining coefficients are set to zero.

We are now ready to describe the NTRU cryptosystem. Suppose Alice wishes to send a message to Bob. To begin, Bob must generate and publish a public key. He selects three integers N, p, q, subject to the constraints:

$$\gcd(p,q) = 1, \quad \text{and} \quad p \ll q.$$

Recommended parameters for moderate security include:

$$(N, p, q) = (107, 3, 64),$$

and for higher security:

$$(N, p, q) = (503, 3, 256).$$

Case	**Level**	N	p	q	**Time (seconds)**
A	Moderate	107	3	64	780,230 (9 days)
B	High	167	3	128	$1.198 \cdot 10^{10}$ (380 years)
C	Highest	503	3	256	$1.969 \cdot 10^{35}$ ($6.2 \cdot 10^{27}$ years)

NTRU Parameter Sets and Estimated Attack Times. Based on data from `https://www.ntru.org/`. Tests were performed on a 200 MHz Pentium Pro processor.

To initiate the cryptosystem, Bob begins by selecting two small polynomials f and g, typically with coefficients in $\{-1, 0, 1\}$. These polynomials are chosen such that f is invertible modulo both p and q. This means there exist polynomials F_p and F_q, each of degree less than N, such that

$$F_p * f \equiv 1 \pmod{p}, \qquad F_q * f \equiv 1 \pmod{q}.$$

Using these, Bob computes the public key polynomial

$$h \equiv F_q * g \pmod{q},$$

and publishes the tuple (N, p, q, h) as his public key.

The private key is the polynomial f, which should be kept secret. Although F_p can be efficiently recomputed from f, it is recommended to store it for faster decryption. As for g, note that since $g \equiv f * h \pmod{q}$, all required information is contained in f and h, so storing g is unnecessary.

Encryption and Decryption in NTRU

Once the keys are established, Alice can encrypt a message for Bob. She encodes her message as a polynomial m of degree less than N, with coefficients from $\{-1, 0, 1\}$, assuming $p = 3$. Then she chooses a small random polynomial ϕ (its precise constraints will be detailed later) and computes the ciphertext as

$$c \equiv p\phi * h + m \pmod{q}.$$

This ciphertext c is sent to Bob.

To decrypt the message, Bob first computes

$$a \equiv f * c \pmod{q}.$$

If all coefficients of a have absolute value less than $q/2$, he can recover the original message by reducing modulo p:

$$m \equiv F_p * a \pmod{p}.$$

Although decryption is not always guaranteed to succeed, appropriate parameter choices yield very low failure probabilities (e.g., less than 5×10^{-5}).

Here's a sketch of why decryption usually works:

$$a \equiv f * c \equiv f * (p\phi * h + m) \equiv f * p\phi * F_q * g + f * m \equiv p\phi * g + f * m \pmod{q}.$$

Given that ϕ, g, f, m all have small coefficients and $p \ll q$, it is likely that the result does not wrap modulo q, so we can treat the expression as:

$$a = p\phi * g + f * m.$$

Applying F_p, we have:

$$F_p * a = pF_p * \phi * g + F_p * f * m \equiv 0 + 1 * m \equiv m \pmod{p},$$

which successfully recovers the message m.

A complete implementation of the NTRU cryptosystem, including key generation, encryption, and decryption, is available on GitHub: `github.com/SanyaKor/Cryptanalysis/blob/main/lattice_methods/ntru.py`

An example demonstrating how to use the NTRU scheme in practice can be found at: `notebooks/usage_examples.ipynb`

Advantages and Disadvantages of NTRU

Advantages:

- *Efficiency:* NTRU offers fast key generation, encryption, and decryption compared to many other public-key schemes.
- *Small Key Sizes:* Public and private keys are relatively small, making NTRU attractive for constrained environments.
- *Post-Quantum Security:* NTRU is believed to be resistant to attacks by quantum computers, unlike RSA or ECC.

Disadvantages:

- *Decryption Failures:* With improper parameters, decryption can fail. Careful tuning is required.
- *Parameter Sensitivity:* Security and correctness heavily depend on choosing parameters properly.
- *Lattice Attacks:* Though generally secure, NTRU remains susceptible to advanced lattice-based attacks if parameters are too weak.

2.6 Attack on NTRU

Let $h = h_{N-1}X^{N-1} + \cdots + h_0$. Form the $N \times N$ matrix

$$H = \begin{pmatrix} h_0 & h_1 & \cdots & h_{N-1} \\ h_{N-1} & h_0 & \cdots & h_{N-2} \\ \vdots & \vdots & \ddots & \vdots \\ h_1 & h_2 & \cdots & h_0 \end{pmatrix}.$$

If we represent the polynomials $f = f_{N-1}X^{N-1} + \cdots + f_0$ and $g = g_{N-1}X^{N-1} + \cdots + g_0$ by their corresponding row vectors:

$$\vec{f} = (f_0, \ldots, f_{N-1}), \quad \vec{g} = (g_0, \ldots, g_{N-1}),$$

then multiplication in $\mathbb{Z}_q[x]/(x^N - 1)$ can be viewed as matrix-vector multiplication $H \cdot \vec{f}$.

Here, H is a circulant matrix generated from the coefficients of h. This form is particularly useful because circulant matrices encode convolution-like operations. When working modulo $X^N - 1$, the multiplication of polynomials behaves like cyclic convolution. Representing this operation via matrix multiplication simplifies both notation and implementation, especially in lattice-based cryptographic schemes like NTRU.

We observe that

$$\vec{f}H \equiv \vec{g} \pmod{q}.$$

To express this relationship as a lattice problem, define the identity matrix I and the scaled identity matrix qI as:

$$I = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}, \qquad qI = \begin{pmatrix} q & 0 & \cdots & 0 \\ 0 & q & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & q \end{pmatrix}.$$

Now form the $2N \times 2N$ matrix M by:

$$M = \begin{pmatrix} I & H \\ 0 & qI \end{pmatrix}.$$

Let y be an unknown polynomial of degree less than N, with coefficient vector $\vec{y}$. Then we can express the relation

$$g = f * h + qy$$

as

$$(\vec{f}, \vec{y})M = (\vec{f}, \vec{g}).$$

This means the vector $(\vec{f}, \vec{g})$ belongs to the lattice L generated by the rows of M. Because f and g have small coefficients, the corresponding vector is short. An attacker can apply lattice reduction techniques (like the LLL algorithm) to search for short vectors in L, and potentially recover f and g, thus compromising the cryptosystem.

3 Code

This section provides an overview of the source code modules responsible for implementing the core functionality of the project.It includes core algorithms for lattice manipulation, polynomial arithmetic, cryptographic operations, and supporting utilities such as testing and verification routines.

Each file is presented with its key functions, their parameters, return types, and the full implementation as extracted directly from the Python source. The structure is modular, with each component focusing on a specific aspect of the overall system.

Note: This documentation includes only the main functions. Some helper or internal code is not shown here. To explore the full project, including all source files and examples, visit the GitHub repository: `https://github.com/SanyaKor/Cryptanalysis`

3.1 Lattice Methods

This section contains core functions for lattice-based computations, including basis reduction, coefficient normalization, and algebraic operations over polynomial rings. These methods form the foundation of the cryptographic and algorithmic procedures implemented in this project.

All functions are implemented in Python and designed to be modular and reusable across different schemes.

3.1.1 basis_reduction_2d

Function: `reduce_2d_basis`

Description:

Performs Gauss-style reduction of a 2D lattice basis.

This function reduces a two-dimensional lattice basis using a simplified form of Gauss' lattice basis reduction algorithm. The process iteratively subtracts integer multiples of vectors to produce a shorter, more orthogonal basis. **Parameters:**

`basis1`	`numpy.ndarray` — First basis vector.
`basis2`	`numpy.ndarray` — Second basis vector.
`verbose`	`bool` — If True, returns a log of each reduction step.

Returns:

`return`	`list` — If `verbose` is False, returns `[b1, b2]` (the reduced basis). If `verbose` is True, returns a list of dictionaries with step-by-step logs: • `step` — step index or "→ shortest" • `b1`, `b2` — current reduced basis vectors (NumPy arrays).

Note: The algorithm stops when the projection coefficient `t` becomes zero. This is a simplified form of Gauss' lattice basis reduction in 2D.

Source Code:

```python
def reduce_2d_basis(basis1, basis2, verbose=False):
    data = []

    steps = 0

    #TODO linear ind check
    # if np.linalg.matrix_rank(np.column_stack((basis1, basis2))) < 2:
    #     raise ValueError("Input vectors are linearly dependent.")

    while True:

        if np.linalg.norm(basis2) < np.linalg.norm(basis1):
            basis1, basis2 = basis2, basis1
            continue


        t = round(np.dot(basis1, basis2) / np.dot(basis1, basis1))

        data.append({
            'step': steps,
            'b1': basis1.copy(),
            'b2': basis2.copy(),
        })

        steps += 1


        if t == 0:
            break

        basis2 = basis2 - t * basis1

    shortest = basis1 if np.linalg.norm(basis1) <= np.linalg.norm(basis2) else basis2

    ###TODO reducing the basis not includes short vector, might have to remove
    data.append({
        'step': ' shortest',
        'b1': shortest if np.array_equal(shortest, basis1) else '',
        'b2': shortest if np.array_equal(shortest, basis2) else ''
    })

    return data if verbose else [basis1, basis2]
```

3.1.2 lll

Function: `lll_reduce`

Description:

Performs LLL (Lenstra–Lenstra–Lovász) lattice basis reduction.

This function applies the classical LLL algorithm to reduce a given lattice basis to a shorter and nearly orthogonal form.

Parameters:

`basis`	list[`numpy.ndarray`] — A list of NumPy vectors representing the lattice basis.
`delta`	float — Lovász parameter, typically in the range (0.5, 1). Default is 0.75.
`verbose`	bool — If True, enables step-by-step debug output (currently unused).

Returns:

`return`	list[`numpy.ndarray`] — A list of NumPy vectors representing the LLL-reduced lattice basis.

Note: This function assumes all basis vectors are linearly independent.

Warning: No validation is performed on the input; ensure basis vectors are valid.

See also: `lattice_methods.utils.gram_schmidt` for orthogonalization.

```python
def lll_reduce(basis, delta=0.75, verbose=False):

    ##TODO verbose ..

    basis = [b.copy() for b in basis]
    n = len(basis)
    k = 1


    while k < n:

        ortho, mu = gram_schmidt(basis)

        for j in range(k - 1, -1, -1):  # j < k

            if abs(mu[k, j]) > 0.5:
                r = round(mu[k, j])
                basis[k] -= r * basis[j]
                ortho, mu = gram_schmidt(basis)

        norm_sq_prev = np.dot(ortho[k - 1], ortho[k - 1])
        norm_sq_curr = np.dot(ortho[k], ortho[k])

        lhs = delta * norm_sq_prev
        rhs = norm_sq_curr + mu[k, k - 1]**2 * norm_sq_prev

        if lhs > rhs:
            basis[k], basis[k - 1] = basis[k - 1], basis[k].copy()

            k = max(k - 1, 1)
        else:
            k += 1

    return basis
```

3.1.3 ntru

Function: `poly_inv_mod_ring`

Description:

Computes the inverse of a polynomial modulo (pow(x,N) - 1) and q.

This function attempts to find the inverse of a given polynomial ‘polynomial_f‘ in the ring of polynomials modulo (pow(x,N) - 1) with coefficients reduced modulo ‘q‘. It works both when ‘q‘ is prime and composite by setting the appropriate polynomial domain.

Parameters:

`polynomial_f`	sympy.Poly — The polynomial to invert (as a SymPy Poly object).
`N`	int — The degree defining the modulus polynomial pow(x,N) - 1.
`q`	int — The modulus for coefficient arithmetic.

Returns:

`return`	list[int] or None — List of coefficients of the inverse polynomial if it exists; otherwise, None.

Source Code:

```
def poly_inv_mod_ring(polynomial_f, N, q):

    #f = Poly(f_coeffs, x, domain=GF(q))
    if(isprime(q)):
        mod_poly = Poly(x**N - 1, x, domain=GF(q))
    else:
        mod_poly = Poly(x ** N - 1, x, domain=ZZ).trunc(q)

    polynomial_f = polynomial_f.set_domain(mod_poly.domain)

    if gcd(polynomial_f, mod_poly).degree() != 0:
        return None

    try:
        inv = invert(polynomial_f, mod_poly)
        return inv.all_coeffs()
    except Exception:
        return None
```

Function: `poly_mult_mod_ring`

Description:

Performs multiplication of two polynomials modulo (pow(x,N) - 1) and coefficient modulus q.

This function computes the product of two polynomials represented by coefficient lists ‘p1‘ and ‘p2‘. The multiplication is done modulo the polynomial (pow(x,N) - 1), which means the coefficients are reduced with wrap-around at degree N, and all coefficients are taken modulo ‘q‘.

Parameters:

`p1`	list[int] — Coefficients of the first polynomial (highest degree first).
`p2`	list[int] — Coefficients of the second polynomial (highest degree first).
`N`	int — Degree of the modulus polynomial (pow(x,N) - 1).
`q`	int — Modulus for coefficient arithmetic.

Returns:

`return`	list[int] — Coefficients of the resulting polynomial after modular multiplication,

Source Code:

```python
def poly_mult_mod_ring(p1, p2, N, q):
    p1 = p1[::-1]
    p2 = p2[::-1]

    length = len(p1) + len(p2) - 1
    result = [0] * length
    for i in range(len(p1)):
        for j in range(len(p2)):
            idx = (i + j)
            result[idx] += p1[i] * p2[j]

    for i in range(len(result)):
        if(i > N-1):
            difference = i % N
            result[difference] += result[i]
            result[i] = 0

    for i in range(len(result)-1, -1, -1):
        if(result[i]!=0 or len(result) <= N):
            break

        result.pop(i)

    return [c % q for c in result[::-1]]
```

Function: `ntru_generate_keys`

Description:

Generates public and private keys for the NTRU cryptosystem.

This function computes the NTRU key pair based on parameters N, p, q and the private polynomials ‘polynomial_f‘ and ‘polynomial_g‘. It returns the public key and the inverse of ‘polynomial_f‘ modulo q, which serves as part of the private key.

Parameters:

`N`	int — Degree of the polynomials and ring dimension.
`p`	int — Small modulus parameter for message space.
`q`	int — Large modulus parameter for polynomial arithmetic.
`polynomial_g`	sympy.Poly — Polynomial used in key generation (SymPy Poly).
`polynomial_f`	sympy.Poly — Private polynomial used for key generation (SymPy Poly).

Returns:

`return`	tuple[list, list] — Tuple ‘(pub_key, prv_key)‘ where

Source Code:

```
def ntru_generate_keys(N : int, p: int, q : int, polynomial_g :
   Poly, polynomial_f : Poly):

    if(gcd(p, q) != 1 or p >= q):
        print("ERROR SMTH WRONG WITH p, q ")
        return

    if(isprime(q)):
        poly_f_over_q = Poly(polynomial_f, x, domain=GF(q))
    else:
        poly_f_over_q = Poly(polynomial_f, x, domain=ZZ).trunc(q)

    poly_f_over_p = Poly(polynomial_f, x, domain=GF(p))

    Fp = poly_inv_mod_ring(poly_f_over_p, N, p)
    Fq = poly_inv_mod_ring(poly_f_over_q, N, q)

    if Fp is None or Fq is None:
        print("ERROR SMTH WRONG WITH POLYNOMIALS Fq Fp")
        return

    Fqp = [p * x for x in Fq]
    h = poly_mult_mod_ring(Fqp, polynomial_g.all_coeffs(), N, q)

    pub_key = [N, p, q, h]
    prv_key = [polynomial_f, Fp]

    return pub_key, prv_key
```

Function: `ntru_encryption`

Description:

Encrypts a message polynomial using the NTRU public key.

This function performs NTRU encryption by computing the ciphertext polynomial as the sum of the product of the random polynomial ‘polynomial_phi‘ with the public key polynomial ‘h‘, plus the message polynomial ‘polynomial_m‘, all modulo ‘q‘.

Parameters:

`pubkey`	list — Public key represented as a list ‘[N, p, q, h]‘.
`polynomial_phi`	sympy.Poly — Random polynomial used for encryption (SymPy Poly).
`polynomial_m`	sympy.Poly — Message polynomial to encrypt (SymPy Poly).

Returns:

`return`	list[int] — Ciphertext polynomial coefficients modulo q.

Source Code:

```
def ntru_encryption(pubkey, polynomial_phi: Poly, polynomial_m:
   Poly):
    N, p, q, h = pubkey

    phi_coeffs = polynomial_phi.all_coeffs()
    m_coeffs = polynomial_m.all_coeffs()
```

```python

    c = poly_mult_mod_ring(phi_coeffs, h, N, q)
    ciphertext = poly_add_mod_ring(c, m_coeffs, q)

    return ciphertext
```

Function: `ntru_decryption`

Description:

Decrypts a ciphertext polynomial using the NTRU private key.

This function performs NTRU decryption by multiplying the ciphertext with the private polynomial ‘polynomial_f‘ modulo (pow(x,N) - 1, q), centering the coefficients if needed, and then multiplying by the inverse of ‘polynomial_f‘ modulo p to recover the original message polynomial.

Parameters:

`pubkey`	list — Public key represented as a list ‘[N, p, q, h]‘.
`prvkey`	list — Private key represented as a list ‘[polynomial_f, Fp]‘,
`ciphertext`	list[int] — Ciphertext polynomial coefficients.

Returns:

`return`	sympy.Poly — Decrypted message polynomial over GF(p).

Source Code:

```python
def ntru_decryption(pubkey, prvkey, ciphertext):
    [polynomial_f, Fp] = prvkey
    N, p, q, h = pubkey

    f_coeffs = polynomial_f.all_coeffs()
    a = poly_mult_mod_ring(f_coeffs, ciphertext, N, q)

    cond = check_coeff_range(a, (-q/2, q/2))
    if not cond:
        a = center_poly_coeffs(a, q)

    Fpa = poly_mult_mod_ring(Fp, a, N, p)
    #print(Poly(Fpa, x, domain=GF(p)))

    return Poly(Fpa, x, domain=GF(p))
```

3.2 Tests

This section provides the core test functions used to verify the behavior of the implemented lattice algorithms. Each function performs automated checks on correctness and expected improvements after reduction.

For complete source code and additional test cases, see the full repository on GitHub. `Cryptanalysis/tests` on GitHub

3.2.1 tests_br2d

Function: `tests_br2d`

Description:

Performs batch testing of 2D lattice basis reduction.

This function takes a list of 2D lattice bases, applies the 'reduce_2d_basis' algorithm to each pair, and verifies two conditions for each reduction: 1. The reduced basis is equivalent to the original basis. 2. The shorter vector in the reduced basis is no longer than in the original.

A test is marked as passed only if both conditions hold.

Parameters:

`basis_list`	list[tuple[np.ndarray, np.ndarray]] — List of 2D lattice bases. Each element is a pair of vectors (b1, b2), where b1 and b2 are NumPy arrays.
`verbose`	bool — Whether to print detailed output for each test.

Returns:

`return`	list[dict[str, Any]] — List of test results. Each result is a dict with reduced vectors and a pass/fail flag.

Source Code:

```python
def tests_br2d(basis_list, verbose=False):
    tests_amount = len(basis_list)
    tests_passed = 0

    results = []

    for i, (b1, b2) in enumerate(basis_list):

        b1_reduced, b2_reduced = reduce_2d_basis(b1, b2)

        same = are_bases_equivalent([b1, b2], [b1_reduced,
           b2_reduced])

        original_len = min(np.linalg.norm(b1), np.linalg.norm(b2))
        reduced_len = min(np.linalg.norm(b1_reduced), np.linalg.
           norm(b2_reduced))
        improved = reduced_len <= original_len

        if same and improved:
            tests_passed += 1
            if verbose:
                print(f" Test {i + 1}: PASSED")
                print(f"Initial basis: b1 = {b1}, b2 = {b2}")
                print(f"Reduced basis: b1 = {b1_reduced}, b2 = {
                    b2_reduced}")

            results.append({
                "b1": b1_reduced,
                "b2": b2_reduced,
                "result": 1
            })

        else:

            if verbose:
                print(f" Test {i + 1} FAILED: ")
                print(f"b1 = {b1}, b2 = {b2}")

            results.append({
                    "b1": b1_reduced,
                    "b2": b2_reduced,
                    "result": 0
            })
    if verbose:
        print(f"\n {tests_amount}/{tests_passed} tests passed.")

    return results
```

3.2.2 tests_lll

Function: `tests_brlll`

Description:

Performs batch testing of LLL lattice basis reduction.

This function takes a list of lattice bases, applies the LLL reduction algorithm to each, and verifies two key properties: 1. The reduced basis is equivalent to the original one. 2. The shortest vector in the reduced basis is no longer than in the original.

Each test is counted as passed if both properties hold.

Parameters:

`basis_list`	list[list[np.ndarray]] — List of lattice bases to test. Each basis is a list of NumPy arrays.
`verbose`	bool — Whether to print step-by-step output for each test.

Returns:

`return`	list[dict[str, Any]] — List of results per test. Each result is a dict with the reduced basis and pass/fail flag.

Source Code:

```python
def tests_brlll(basis_list, verbose=False):
    tests_amount = len(basis_list)
    tests_passed = 0

    results = []

    for i, basis in enumerate(basis_list):
        reduced = lll_reduce(basis)

        same = are_bases_equivalent(basis, reduced)

        original_len = min(np.linalg.norm(v) for v in basis)
        reduced_len = min(np.linalg.norm(v) for v in reduced)
        improved = reduced_len <= original_len

        result = int(same and improved)
        if result:
            tests_passed += 1

        if verbose:
            print(f"{'' if result else ''} Test {i + 1}: {'PASSED' if result else 'FAILED'}")
            print("Initial basis:")
            for v in basis:
                print(" ", v)
            print("Reduced basis:")
            for v in reduced:
                print(" ", v)
            print()

        results.append({
            "basis": [v.tolist() for v in reduced],
            "result": result
        })

    if verbose:
        print(f"\n {tests_passed}/{tests_amount} tests passed.")

    return results
```

4 Notebooks

4.1 Exercises

This section contains a collection of exercises related to lattice-based cryptography, adapted from the textbook *Introduction to Cryptography with Coding Theory* by W. Trappe and L. C. Washington (2nd Edition, Pearson, 2006).

The problems focus on topics such as NTRU, modular arithmetic, and lattice reduction.

4.1.1 Exercise 1

Find a reduced basis and a shortest nonzero vector in the lattice generated by the vectors (58,19), (168,55).

```
[7]: import pandas as pd
     from lattice_methods import reduce_2d_basis
     import numpy as np

     b1 = np.array([58, 19])
     b2 = np.array([168, 55])

     data = reduce_2d_basis(b1, b2, verbose = True)
     table = pd.DataFrame(data)
     display(table.style.hide(axis="index"))
```

```
| step       | b1       | b2         |
|:-----------|:---------|:-----------|
| 0          | [58 19]  | [168  55]  |
| 1          | [-6 -2]  | [58 19]    |
| 2          | [-2 -1]  | [-6 -2]    |
| 3          | [0 1]    | [-2 -1]    |
| 4          | [0 1]    | [-2  0]    |
```

4.1.2 Exercise 2

(a) Find a reduced basis for the lattice generated by the vectors (53,88), (107,205).

(b) Find the vector in the lattice of part (a) that is closest to the vector (151,33).

```
[8]: b1 = np.array([53, 88])
     b2 = np.array([107, 205])

     data = reduce_2d_basis(b1, b2, verbose = True)
     b1_reduced, b2_reduced = data[-2]["b1"], data[-2]["b2"]
     print(f"Reduced basis : {b1_reduced}, {b2_reduced}")


     # TODO (b)
```

```
Reduced basis : [ 1 29], [50  1]
```

4.1.3 Exercise 4

Let $\{v_1, v_2\}$ be a basis of a lattice. Let a, b, c, d be integers such that $ad - bc = \pm 1$, and define:

$$w_1 = av_1 + bv_2, \quad w_2 = cv_1 + dv_2$$

(a) Show that:

$$v_1 = \pm(dw_1 - bw_2), \quad v_2 = \pm(-cw_1 + aw_2)$$

(b) Show that $\{w_1, w_2\}$ is also a basis of the lattice.

Solution

(a) We define the change of basis as:

$$\begin{aligned} w_1 &= av_1 + bv_2 \\ w_2 &= cv_1 + dv_2 \end{aligned}$$

In matrix form:

$$\vec{w} = A \cdot \vec{v}$$

where

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad \vec{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}, \quad \vec{w} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

and

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

As stated in the problem, $ad - bc = \pm 1 \Rightarrow \det(A) = \pm 1$.

Since $\det(A) \neq 0$, we can write:

$$\vec{v} = A^{-1} \cdot \vec{w}$$

The inverse of matrix A is:

$$A^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Hence:

$$\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

Which gives:

$$v_1 = \pm(dw_1 - bw_2), \quad v_2 = \pm(-cw_1 + aw_2)$$

(b) We are given:

$$w_1 = av_1 + bv_2, \quad w_2 = cv_1 + dv_2$$

Since $ad - bc = \pm 1$, the transformation is invertible over integers.

We can solve for v_1, v_2 as:

$$v_1 = dw_1 - bw_2, \quad v_2 = -cw_1 + aw_2$$

This shows that:

- $v_1, v_2 \in L(w_1, w_2)$
- $w_1, w_2 \in L(v_1, v_2)$

Therefore:

$$L(v_1, v_2) = L(w_1, w_2)$$

$\Rightarrow \{w_1, w_2\}$ is also a basis of the same lattice.

4.1.4 Exercise 5

Let N be a positive integer.

(a) Show that if $j + k \equiv i \pmod{N}$, then $X^{j+k} - X^i$ is a multiple of $X^N - 1$.

(b) Let $0 \le i < N$. Let $a_0, \ldots, a_{N-1}, b_0, \ldots, b_{N-1}$ be integers, and define:

$$c_i = \sum_{j+k \equiv i \pmod{N}} a_j b_k$$

where the sum is taken over all pairs j, k such that $j + k \equiv i \mod N$. Show that:

$$c_i X^i - \sum_{j+k \equiv i \pmod{N}} a_j b_k X^{j+k}$$

is a multiple of $X^N - 1$.

(c) Let f and g be polynomials of degree less than N. Let $f \cdot g$ be the usual product of f and g, and let $f * g$ be the **cyclic convolution** of f and g as defined in Section 17.4. Show that:

$$f \cdot g - f * g$$

is a multiple of $X^N - 1$.

Definition: Cyclic Convolution $f * g$

Let f and g be polynomials of degree less than N:

$$f = a_{N-1}X^{N-1} + \cdots + a_0, \quad g = b_{N-1}X^{N-1} + \cdots + b_0$$

Then their **cyclic convolution** is defined as:

$$h = f * g = c_{N-1}X^{N-1} + \cdots + c_0$$

where the coefficients c_i are given by:

$$c_i = \sum_{j+k \equiv i \pmod{N}} a_j b_k$$

That is, the sum is taken over all index pairs j, k such that $j + k \equiv i \mod N$.

Solution

(a) It is given that $j + k \equiv i \pmod{N}$, which means:

$$j + k \equiv i \pmod{N} \Rightarrow (j + k) \bmod N = i \bmod N$$

$\Rightarrow$ there should exist some integer remainder $m \in \mathbb{Z}$ such that $j + k = i + mN$

$$\Rightarrow X^{j+k} \equiv X^{i+mN}$$

We work in the ring: $\mathbb{Z}[X]/(X^N - 1)$

In this ring, we have:

$$X^N - 1 = 0 \quad \Rightarrow \quad X^N \equiv 1 \mod (X^N - 1)$$

Therefore:

$$X^{i+mN} = X^i \cdot (X^N)^m \equiv X^i \cdot 1^m = X^i \mod (X^N - 1)$$

Since $X^{j+k} = X^{i+mN}$, we conclude:

$$X^{j+k} - X^i \equiv 0 \mod (X^N - 1)$$

(b) By definition:

$$c_i = \sum_{j+k\equiv i} a_j b_k \Rightarrow c_i X^i = \sum_{j+k\equiv i} a_j b_k X^i$$

Now consider the expression:

$$c_i X^i - \sum_{j+k\equiv i} a_j b_k X^{j+k}$$

Substitute the expression for $c_i X^i$:

$$= \sum_{j+k\equiv i} a_j b_k X^i - \sum_{j+k\equiv i} a_j b_k X^{j+k}$$

Since the sums are over the same index set, we combine:

$$= \sum_{j+k\equiv i} a_j b_k (X^i - X^{j+k})$$

From part (a), we know that if $j+k\equiv i \mod N$, then:

$$X^{j+k} \equiv X^i \mod (X^N-1) \Rightarrow X^i - X^{j+k} \equiv 0 \mod (X^N-1)$$

So for each term in the sum, we have:

$$a_j b_k (X^i - X^{j+k}) \equiv 0 \mod (X^N-1)$$

Therefore, the whole sum is congruent to 0:

$$\sum_{j+k\equiv i} a_j b_k (X^i - X^{j+k}) \equiv 0 \mod (X^N-1)$$

Hence,

$$c_i X^i - \sum_{j+k\equiv i} a_j b_k X^{j+k} \equiv 0 \mod (X^N-1)$$

So the expression is a multiple of X^N-1, as required.

(c) Let:

$$f(X) = \sum_{j=0}^{N-1} a_j X^j, \quad g(X) = \sum_{k=0}^{N-1} b_k X^k$$

Then the **usual product** is:

$$f \cdot g = \sum_{m=0}^{2N-2} \left(\sum_{j+k=m} a_j b_k \right) X^m$$

The **cyclic convolution** is:

$$f * g = \sum_{i=0}^{N-1} \left(\sum_{j+k \equiv i \pmod N} a_j b_k \right) X^i$$

In this ring, we have:

$$X^m \equiv X^{m \bmod N} \mod (X^N - 1)$$

So in this system, all terms in fg with degrees $\geq N$ get **wrapped around** (reduced modulo N).

This means that fg becomes:

$$f \cdot g \mod (X^N - 1) = \sum_{i=0}^{N-1} \left(\sum_{j+k \equiv i \pmod N} a_j b_k \right) X^i = f * g$$

Thus:

$$f \cdot g \equiv f * g \mod (X^N - 1) \quad \Rightarrow \quad f \cdot g - f * g \equiv 0 \mod (X^N - 1)$$

So:

$$f \cdot g - f * g \text{ is a multiple of } X^N - 1$$

4.2 Usage Examples

The full notebook with code and visual output is available at:
`notebooks/usage_examples.ipynb` on GitHub

This section presents selected examples extracted from interactive Jupyter notebooks. They demonstrate how to use the core functions defined in the codebase, providing practical context and visual insight into the algorithms in action.

These examples are included to illustrate real usage patterns without replicating the full notebooks.

Note: For complete code listings and further experiments, refer to the corresponding notebooks in the GitHub repository.

Example 1: The following example is taken from the `usage_examples` module and demonstrates how to apply the function `reduce_2d_basis` in a practical setting.

Reduce a 2D Basis and higher

Function: `lll_reduce`

Implements the **Lenstra–Lenstra–Lovász (LLL)** lattice basis reduction algorithm for integer bases in arbitrary dimension. Applies size reduction and swaps based on the Lovász condition to produce shorter, nearly orthogonal vectors.

Parameters

Name	Type	Description
`basis`	`List[np.ndarray]`	List of linearly independent integer vectors (dimension `n`)
`delta`	`float`	Lovász parameter, typically between 0.5 and 1. Default is `0.75`
`verbose`	`bool`	If `True`, prints the internal steps of reduction. Default is `False`

Returns

- `List[np.ndarray]`: The reduced basis as a list of vectors in the same dimension as the input.

---

[2]:
```python
data = lll_reduce([b1,b2], verbose=False)
print(data)
```

```
[array([0, 1]), array([-2,  0])]
```

Example 2:

Performs iterative 2D lattice basis reduction using projection and subtraction (similar to Gram-Schmidt). Returns a list of intermediate steps for inspection or visualization.

Parameters

Name	Type	Description
`basis1`	`np.ndarray`	First 2D basis vector (shape `(2,)`)
`basis2`	`np.ndarray`	Second 2D basis vector (shape `(2,)`)
`verbose`	`bool`	If `True`, prints step-by-step details to the console. Default is `False`

Returns A `List[Dict]` of reduction steps. Each step contains: - `'step'`: step index (starting from 0) - `'b1'`: current state of the first basis vector - `'b2'`: current state of the second basis vector

---

[1]:
```python
from lattice_methods import reduce_2d_basis
from lattice_methods import lll_reduce
import numpy as np
import pandas as pd

# 2d vector example
b1 = np.array([58, 19])
b2 = np.array([168, 55])

#
data = reduce_2d_basis(b1, b2, verbose=True)
table = pd.DataFrame.from_dict(data)
display(table.style.hide(axis="index"))
```

```
| step       | b1       | b2         |
|:-----------|:---------|:-----------|
| 0          | [58 19]  | [168  55]  |
| 1          | [-6 -2]  | [58 19]    |
| 2          | [-2 -1]  | [-6 -2]    |
| 3          | [0 1]    | [-2 -1]    |
| 4          | [0 1]    | [-2  0]    |
```

4.3 Tests

The full notebook with code and visual output is available at:
`notebooks/tests.ipynb` on GitHub

This notebook provides interactive tests and visual demonstrations of the core algorithms implemented in the `lattice_methods` module, including basis reduction, polynomial transformations, and related lattice operations.

- reduce_2d_basis: View source
- lll_reduce: View source Based on Wikipedia
- ntru: View source

Example 1

This is an example of testing the `reduce_2d_basis` function using `tests_br2d`, which verifies the correctness of 2D lattice basis reduction. The test checks whether the output basis spans the same lattice and whether the reduction improves vector lengths.

[2]:
```
from tests import tests_br2d
from tests import generate_random_bases
from tests import tests_brlll
from lattice_methods import are_bases_equal_2d


sample = generate_random_bases(10, 2)
tests_br2d(sample, True);
```

```
 Test 1: PASSED
Initial basis: b1 = [-19 -42], b2 = [47 44]
Reduced basis: b1 = [28  2], b2 = [  9 -40]
 Test 2: PASSED
Initial basis: b1 = [ 19 -49], b2 = [-25  49]
Reduced basis: b1 = [-6  0], b2 = [  1 -49]
 Test 3: PASSED
Initial basis: b1 = [-50 -38], b2 = [12 47]
Reduced basis: b1 = [-38   9], b2 = [12 47]
 Test 4: PASSED
Initial basis: b1 = [ 50 -21], b2 = [-10  27]
Reduced basis: b1 = [-10  27], b2 = [40  6]
 Test 5: PASSED
Initial basis: b1 = [50 47], b2 = [-37  41]
Reduced basis: b1 = [-37  41], b2 = [50 47]
 Test 6: PASSED
Initial basis: b1 = [-12  12], b2 = [36 13]
Reduced basis: b1 = [-12  12], b2 = [24 25]
 Test 7: PASSED
Initial basis: b1 = [33 28], b2 = [-30 -34]
```

```
Reduced basis: b1 = [ 3 -6], b2 = [39 16]
 Test 8: PASSED
Initial basis: b1 = [23 49], b2 = [-17 -17]
Reduced basis: b1 = [-11  15], b2 = [-17 -17]
 Test 9: PASSED
Initial basis: b1 = [ 13 -23], b2 = [ 29 -44]
Reduced basis: b1 = [3 2], b2 = [ 16 -21]
 Test 10: PASSED
Initial basis: b1 = [ 49 -13], b2 = [-28  20]
Reduced basis: b1 = [21  7], b2 = [-7 27]

 10/10 tests passed.
```

Example 2

This is a test of the NTRU encryption scheme, including key generation, lattice construction, and basis reduction for correctness verification.

Example taken from the book: ***Introduction to Cryptography with Coding Theory*** **Authors:** William Trappe, Lawrence C. Washington **Chapter:** 17 **Edition:** 2nd Edition **Publisher:** Pearson, 2006

---

[14]:
```
x = symbols('x')

N = 5
p = 3
q = 16

#### a_0 x^n + a_1 x^n-1.....
phi = [1,-1]
m = [1, -1, 1]
g = [0, 1, 0, -1, 0]
f = [1, 0, 0, 1, -1]


poly_f = Poly(f, x)
poly_g = Poly(g, x)
poly_m = Poly(m, x)
poly_phi = Poly(phi, x)

pub_key, prv_key = ntru_generate_keys(N, p, q, poly_g, poly_f)
ciphertext = ntru_encryption(pub_key, poly_phi, poly_m)
poly_d = ntru_decryption(pub_key, prv_key, ciphertext)

[f, Fp] = prv_key
[N,p, q, h] = pub_key
```

```
poly_m = Poly(poly_m, x, domain=GF(p))

print(" Original Message Polynomial:")
print(f"    m(x) = {poly_m}\n")

print(" Decrypted Message Polynomial:")
print(f"    m'(x) = {poly_d}")
print("=" * 80)
```

```
 Original Message Polynomial:
    m(x) = Poly(x**2 - x + 1, x, modulus=3)

 Decrypted Message Polynomial:
    m'(x) = Poly(x**2 - x + 1, x, modulus=3)
================================================================================
```

5 Conclusion

This project explored key techniques in lattice-based cryptography. We started with two-dimensional basis reduction and then applied the LLL algorithm to higher-dimensional problems. Using these tools, we analyzed two attacks: one on RSA with partial message leakage, and another on NTRU, where short vectors reveal secret keys. The practical examples showed how lattice reduction helps uncover hidden information when parameters are weak. Through both theory and implementation, we saw how powerful and versatile lattice methods are in modern cryptography.

6 References

- **Source Code and Notebooks**
  Complete codebase and usage examples available at:
  `https://github.com/SanyaKor/Cryptanalysis`
- **Introduction to Cryptography with Coding Theory**
  W. Trappe, L. C. Washington — 2nd Edition, Pearson, 2006.
- **NTRU: A Ring-Based Public Key Cryptosystem**
  J. Hoffstein, J. Pipher, J. H. Silverman, 1998.
  `https://ntru.org`
- **Lenstra–Lenstra–Lovász lattice basis reduction algorithm**
  Wikipedia Article.
  Wikipedia
- **Public Key Cryptography – NTRU**
  Sz. Tengely — Example values used in this notebook based on:
  `https://shrek.unideb.hu/~tengely/crypto/section-8.html`
- **Applied Cryptanalysis: Breaking Ciphers in the Real World**
  M. Stamp, R. M. Low — Wiley-IEEE Press, 2007. Chapter 6.7.