SSP: An Alternative Perspective to NTRU

Caleb Seow[1], Ng Zi Yang[2], Choo Jia Guang[3], Lim Wen Cong Isla[3]
[1] Catholic High School, 9 Bishan Street 22, Singapore 579767
[2] Temasek Junior College, 22 Bedok South Road, Singapore 469278
[3] DSO National Laboratories, 12 Science Park Drive, Singapore 118225

Abstract

NTRU is a post-quantum public key cryptography scheme based on operations in polynomial quotient rings reduced 2 different moduli. Current attacks (described in this paper) are based off finding the private key as a short vector in a well-constructed lattice. This problem, known as the shortest vector problem (SVP), is known to be a hard problem in general.

This paper presents another attack based not on lattice reduction, but on finding subsets that sum to certain targets, known as the Subset Sum Problem (SSP). While the SSP is NP-Complete, it is a lot better studied than the SVP, and has algorithms with in pseudo-polynomial runtimes. It is our hope that this different perspective provides more insight into NTRU.

Introduction

NTRU was proposed by Hoffstein, Pipher and Silverman and presented in Crypto '96 [1]. Since then, many different variants of NTRU have been made with small tweaks [2]. This paper looks at one particular variant – NTRU-HPS. After a short description of the scheme, we look at some naïve brute force attacks, then at the current approach to attacks via lattice reduction algorithms, before presenting our own attack based on the Subset Sum Problem. Afterwards, we sketch a possible dynamic programming solution for illustration. Note that the algorithm is for understanding, and that faster algorithms to solve the Subset Sum Problem exist.

The NTRU Cryptographic System

In NTRU, we pick integers $(n, p, q) \geq 1$ such that $\gcd(n, q) = \gcd(p, q) = 1$ and let R and R_q be polynomial quotient rings

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

An element $f \in R$ can also be expressed as a vector: $f = \sum_{i=0}^{n-1} f_i x^i = [f_0, f_1, \ldots, f_{n-1}]$.

Next, we define the product of polynomials in R as

$$a(x) \star b(x) = c(x), \text{ where } [x^k]c(x) = \sum_{i+j \equiv k \,(\mathrm{mod}\, q)} a_i \, b_j \, .$$

This is in fact just regular polynomial multiplication, where each term x^{n+k} is reduced to x^k. This operation is the same in R_q , except that coefficients of the product are reduced modulo q .

For NTRU-HPS, we define T as the set of polynomials in R with coefficients -1,0,or 1. We also define $T(d)$ as the set of polynomials in R with $d/2$ coefficients as 1 , $d/2$ coefficients as -1 and

remaining coefficients as 0 . These polynomials are termed ternary, meaning that they only contain 3 different coefficients. In this case, these coefficients are centered around 0 .

In NTRU, we have public perimeters (n, p, q) and private keys $f \in T$ and $g \in T(q/8 - 2)$. For cleaner notation, we let $d = q/16 - 1$ from here on out. For 128-bit security (commonly used today), NIST recommends parameters $(n, p, q) = (509,3,2048)$. We will be using the approximation $d \approx n/4$. For more details in implementation, see [2]. The underlying mathematical problem for NTRU is: Given $h \in R_q$, find f and g such that $f \star h \equiv pg \pmod{q}$, where p is usually 3 . We also note that for a key (f, g), any rotation of that key $(x^k \star f, x^k \star g)$ will also be a viable key. While there might exist alternative (f', g') that fulfil the equation above, the chances of this occurring are astronomically small [3, Ch. 6.10.2].

A Brute Force Attack

We begin by looking at a naïve brute force attack. Since f is ternary, it can be thought of as a string of 1 s, -1 s and 0 s. Since f is random, the number of 1 s, -1 s and 0 s is approximately $n/3$. This brute force approach tries every possible combination. Mathematically put, we will be finding the solution set of $f_0^2 + f_1^2 + \cdots + f_{n-1}^2 \approx 2n/3$. Geometrically, this solves for integer points on an n - sphere of radius approximately $\sqrt{2n/3}$. However, as each f_k can take on 3 different values, we have a time complexity on the order of $O(3^n)$.

In addition, note that we can arbitrarily pick the value of g_0 (as there will exist some cycle of f that will result in that corresponding g). Setting $g_0 = 0$ gives

$$p \cdot g_0 = (h_0 f_0 + h_1 f_{n-1} + \cdots + h_{n-1} f_1) = 0.$$

Notice that pg_0 can thus be expressed as the dot product of h with a cycle of the reverse of f . More specifically, if J is an $n \times n$ exchange matrix, then $pg_0 = \langle h, x \star Jf \rangle = 0$. This implies h and $x \star Jf$ are orthogonal vectors. Geometrically speaking, we now know f lies on the intersection of the surface of an n - sphere and some specific $n - 1$ subspace (An illustration is given in Annex A). Algebraically, this gives us another equation, decreasing the number of possible keys f' by 3 . Since there are approximately $n/3 \cdot 0$ s that can be considered pg_0, we get an attack of $O(3^n/n)$ attempts.

We can extend this using the probabilistic argument that for large n , there is a high chance that there exist 2 (or more) arbitrary terms separated by a chosen distance (i.e. that there exists some $pg_i = \alpha$ and $pg_{i+k} = \beta$ for known $\alpha, \beta \in \{-3, 0, 3\}$, $0 < k \leq n - 1$). This gives us an additional equation to work with, decreasing the number of combinations by 3. This argument can be extended so long as the probability of such a case not occurring remains very low.

Lattice Reduction Attacks

Before diving into the attack, we will give some background on lattices. We will assume readers are familiar with the definitions of vector spaces, span, independence and vector bases.

By definition, n linearly independent vectors will span an n - dimensional vector space. Consider a set of independent vectors $V = \{v_1, \dots, v_n\}$ where $v_i \in \mathbb{Z}^n$. We define the lattice spanned by V, $L(V)$, to be the set of all vectors that can be formed by a linear combination of vectors in V with integer coefficients: $L(V) = \{a_1 v_1 + \cdots + a_n v_n : a_i \in \mathbb{Z}\}$. For the purposes of this paper, we will assume that all lattices have integer coordinates. Similar to vector bases, V' is a basis for $L(V)$ if and only if $L(V) = L(V')$. For more details, see [3, Ch. 6.4].

Suppose that $\{v_1, \dots, v_n\}$ is a basis for L and $\{w_1, \dots, w_n\}$ are vectors in L. In other words, each vector $w_i = a_{i,1} v_1 + \cdots + a_{i,n} v_n$. Let V and W be column vectors $(v_1, \dots, v_n)$ and $(w_1, \dots, w_n)$ respectively. Then $W = \mathcal{M}V$, where $\mathcal{M}$ is the matrix below.

$$\mathcal{M} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

Then $\{w_1, \dots, w_n\}$ is also a basis for L if and only if $\det(\mathcal{M}) = \pm 1$ [3, Ch. 6.4]. Intuitively, this means that the volume of the parallelepipeds formed by both bases, and thus the average distance between lattice points, are the same.

Lattice reduction is the process of finding a shorter and more orthogonal basis (non-zero) vectors to a lattice.

In open literature, the most efficient attacks against NTRU have been with lattice reduction algorithms [1]. That is, constructing the $2n$ - dimensional lattice L^{NTRU}, whose vectors form rows of the matrix $\mathcal{M}^{NTRU}$ as shown below, we can show that the vector $(f, pg) = (f_0, \dots, f_{n-1}, pg_0, \dots, pg_{n-1})$ lies in L^{NTRU}.

$$\mathcal{M}^{NTRU} = \left(\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & h_0 & h_1 & \cdots & h_{n-1} \\ 0 & 1 & \cdots & 0 & h_{n-1} & h_0 & \cdots & h_{n-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & h_1 & h_2 & \cdots & h_0 \\ \hline 0 & 0 & \cdots & 0 & q & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & q & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & q \end{array}\right)$$

Note that $\mathcal{M}^{NTRU}$ is composed of 4 $n \times n$ square matrices. Since $\mathcal{M}^{NTRU}$ is a triangular matrix, $\det(\mathcal{M}^{NTRU}) = q^n$. The upper left block is the identity matrix; the upper right block is made of cycles of the coefficients of h; the lower left block is a zero matrix; and the lower right block is q times the identity matrix. As such, it is often (and will, for this paper) be abbreviated to

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

In this paper, the rows of the matrix are the vectors; other literature might show the transpose of this matrix, with its columns as vectors.

Now, we will show that $(f,\ pg)$ is a vector in L^{NTRU}. Since $f \star h \equiv pg \pmod q$, for some polynomial $u(x) \in R$, $f \star h = pg + qu$. Thus, we find that $(f, -u)\mathcal{M}^{NTRU} = (f, pg)$ showing that it is in L^{NTRU}[3, Ch. 6.11]. Thinking about the full $2n \times 2n$ matrix multiplication, we can observe that $[x^k]u$ corresponds with the number of q s we "mod" off $[x^k](f \star h)$ to obtain $p \cdot [x^k]g$. We will explore this approach later on in section 4 .

However, it is not automatically clear why should find shorter bases. Firstly, we will prove that (f, pg) is highly likely to be one of the shortest vectors in L^{NTRU}. In the following paragraphs after, we prove that (f, pg) is a basis vector (and thus can be attacked by lattice reduction).

The Gaussian Heuristic for a lattice L of n dimensions is a prediction of the length of the shortest nonzero vector in a randomly chosen lattice [3, Ch. 6.5]. It is given by

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

We get the Gaussian Heuristic for L^{NTRU} to be

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

From above, we get

$$\big||f|\big| \approx \sqrt{\frac{n}{3}\cdot 1^2 + \frac{n}{3}\cdot(-1)^2} = \sqrt{\frac{2n}{3}} \text{ and } \big||g|\big| \approx \sqrt{\frac{n}{4}\cdot 1^2 + \frac{n}{4}\cdot(-1)^2} = \sqrt{\frac{n}{2}}.$$

And hence, we estimate the length of (f, pg) to be

$$\big||(f, pg)|\big| = \sqrt{\frac{2n}{3}} + p \cdot \sqrt{\frac{n}{2}}.$$

Using the sample perimeters $n = 509$, $p = 3$, and $q = 2048$, we get

$$\big||(f, 3g)|\big| \approx 66.280 \ll \sigma(L^{NTRU}) = 698.76\ (5sf).$$

In fact, the existence of such a short vector (f, pg) would be incredibly surprising had we not constructed L^{NTRU} such that (f, pg) is a lattice point. Thus, it is highly likely (f, pg) is the shortest vector in L^{NTRU}.

We define the i -th minima for a lattice, $\lambda_i(L)$ for the i -th shortest vector in that lattice.

So how do we know that (f, pg) is even a basis vector in L^{NTRU}?

We prove in Annex B that any vector $w = a_1v_1 + \cdots + a_nv_n$ with $\gcd(a_1, \ldots, a_n) = 1$ can be a basis vector of some basis. λ_1 definitely has coefficients that fit this criterion; else, there exist a shorter vector where each coefficient is divided by the $\gcd(a_i)$.

Now we can finally explore lattice reduction algorithms.

We begin with $\mathbb{Z}$ (1 dimension). Since 1 vector is trivial, we give ourselves 2 linearly dependent vectors v_1, v_2. Without loss of generality, we suppose $\big||v_1|\big| \geq \big||v_2|\big|$. A straightforward approach would be to take the longer vector, and subtract off the shorter vector once (i.e. $v_1 - v_2$). Then we repeat this so long as $\big||v_1|\big|$ is greater than 0 . Since v_1 and v_2 are 1 - dimensional, we can treat

them as numbers. Hence, letting $||v_1|| = k_1$ and $||v_2|| = k_2$, the step above is equivalent to taking $k_1 \pmod{k_2}$. If the new $v_1' = k_1 \pmod{k_2} < v_2$, we swap the vector names and repeat the first step. We can then loop this process until $v_1^k = v_2^k$.

This process is similar to the Euclidean algorithm for finding the greatest common factor. At the heart of this algorithm is the fact that $\gcd(\alpha, \beta) = \gcd(\alpha, \beta \bmod \alpha)$. In fact, by Bézout's Identity, given 2 numbers, $\alpha, \beta \in \mathbb{Z}$, there exists $x, y \in \mathbb{Z}$ such that $\alpha x + \beta y = \gcd(\alpha, \beta)$. Furthermore, the solution to any linear combination of α, β is a multiple of $\gcd(\alpha, \beta)$. Hence, this tells us that the shortest vector formed by v_1, v_2 has a magnitude $\gcd(k_1, k_2)$, and can be found in polynomial time.

In 2 dimensions, with independent vectors v_1, v_2 and $||v_1|| \geq ||v_2||$, we hope to have find an analogous idea of having the longer vector "mod" the smaller one. One thing way we can do this is by subtracting from the longer vector its projection onto the shorter vector. In other words,

$$v_1' = v_1 - \mu_{1,2} \cdot v_2, \text{ where } \mu_{1,2} = \frac{\langle v_1, v_2 \rangle}{\langle v_2, v_2 \rangle} = ||v_1|| \cos\theta \ .$$

Note that $\mu_{1,2}$ is the component of v_1 parallel to v_2 , and that in general, $\mu_{i,j}$ can be read as "the projection of v_i on v_j".

Since we subtract off the part parallel to v_2, note that $v_1 - \mu_{1,2} \cdot v_2$ is orthogonal to v_2. Any further subtractions will increase the magnitude of the vectors. However, this is cheating since $v_1 - \mu_{1,2} \cdot v_2$ is not usually a lattice point. Instead, we take $\mu_{1,2}' = \lfloor \mu_{1,2} \rceil = \lfloor\, ||v_1|| \cos\theta \,\rceil$, finding the integer multiple of v_2 that makes v_1' shortest. With this, we arrive at the basis of the Lagrange-Gauss algorithm [4, Ch. 17.1]. The algorithm's pseudocode given in Annex C.

However, in higher dimensions, it becomes unclear which vectors should be subtracted from which in order to obtain the shortest vector. In particular, greedy pairwise reductions may fail to fully reduce a lattice.

In 1982, Lenstra, Lenstra and Lovasz published the LLL algorithm [5], a revolutionary algorithm for lattice reduction. More specifically, the LLL algorithm terminates in polynomial time, and guarantees finding a vector v_i such that $||v_i|| \leq 2^{\frac{n-1}{2}} \cdot ||\lambda_1(L)||$. This is based off 2 conditions.

Before going into these conditions, we define the Gram-Schmidt vector v_i^* as the projection of $v_i$onto the subspace orthogonal to the span of $\{v_1, \dots, v_{i-1}\}$. Additionally, all Gram-Schmidt vectors are to be orthogonal to each other. Rigorously, we get

$$v_i^* = v_i - \sum_{j=1}^{i-1} \mu_{i,j} \cdot v_j^* \text{ , where, similar to above, } \mu_{i,j} = \frac{\langle v_i, v_j \rangle}{\langle v_j, v_j \rangle}.$$

Note that $v_1^* = v_1$ by definition. Then for a lattice L with a basis $B = \{v_1, \dots, v_n\}$ (with some order), we can get a corresponding Gram-Schmidt basis $B^* = \{v_1^*, \dots, v_n^*\}$. Note that B^* (and so the LLL algorithm below) is dependent on the ordering of B .

Since $\mu_{i,j}$ is, more likely than not, non-integral, B^* is not a lattice basis for L , but rather a basis only for the vector subspace spanned by B . It is interesting to note that $\det(L) = \prod v_i^*$. Intuitively, this is because all v_i^* are orthogonal and thus form a parallelepiped in the form of a hyper-cuboid. For more details, see [3, Ch. 6.12].

Now onto the conditions for LLL. A basis is said to be LLL-reduced if it satisfies 2 conditions:

1. Size Condition:

$$|\mu_{i,j}| \leq \frac{1}{2}, \text{ for all } 1 \leq j < i \leq n.$$

2. Lovász Condition:

$$||v_i^*||^2 \geq (\delta - \mu_{i,i-1}^2)||v_{i-1}^*||^2, \text{ for all } 1 < i \leq n, \text{ for some constant } \delta < 1.$$

The Size Condition ensures that vectors cannot be made shorter by some sort of pairwise subtraction.

The Lovász Condition is slightly more complicated. Note that $v_i^*$is orthogonal to v_{i-1}. Then we can rewrite the condition as:

$$\left||v_i^* + \mu_{i,i-1}v_{i-1}\right|^2 \geq \delta\left||v_{i-1}^*\right|^2$$

$$\Rightarrow || \text{ Projection of } v_i^* \text{ onto the orthogonal span of } \{v_1, \dots, v_{i-2}\}||^2$$
$$\geq \delta || \text{ Projection of } v_{i-1}^* \text{ onto the orthogonal span of } \{v_1, \dots, v_{i-2}\}||^2$$

Keeping $\delta < 1$ ensures that the LLL algorithm terminates in polynomial time; δ is often set at $\frac{3}{4}$. However, the effects of δ are quite unpredictable and a larger δ does not necessarily imply a better LLL-reduced basis [3, Ch. 6.12]. An implementation of the LLL algorithm on Python can be found in Annex D.

4 A Subset Sum Attack

The Subset Sum Problem (SSP) is a famous NP-Complete problem: Given a set (or multiset, since repeats are allowed) of integers S, can we determine if some subset of S sums up to some arbitrary k? This decision problem is NP-Complete. Furthermore, if such a subset does exist, can we generate all valid subsets?

It is our hope that since SVP is a hard problem, while SSP is NP-Complete, that perhaps some insight can be found in translating from finding the shortest vector into solving the subset sum problem.

In Section 3, we noted that $[x^k](f \star h) - [x^k]u = p \cdot [x^k]g$. By definition (see Section 1), we also know that $[x^k]g = \sum_{i+j \equiv k \pmod q} f_i h_j$. These 2 equations give us a different perspective to look at the problem.

From here, we denote F as the reverse of f (ie. $F = Jf$, where J is an exchange matrix). Then, we get

$$[x^k]g = \sum_{i+j \equiv k \pmod q} f_i h_j = \langle x^k \star F, h \rangle,$$

where $\langle a, b \rangle$ is the dot product of a and b.

At the same time, since f, and so F, is ternary, $\langle F, h \rangle$ is equivalent to either choosing some element in h, the negative of that element, or not choosing that element at all. More specifically,

if $F_i = 1$, then we "choose" h_i. If $F_i = -1$, we "choose" $-h_i$. And if $F_i = 0$, we do not "choose" h_i. Afterwards, we sum up all of the "chosen" terms as M. If q divides M with a remainder of $-p, 0, p$, we say that M "passes". Our aim is to find an F' such that for every cycle of F', its corresponding M passes. Then that corresponding f' will suffice as a key.

One important note is that while some versions of SSP include negative numbers in S, the fastest (albeit exponential-time) algorithms only consider positive terms. To overcome this problem, instead of finding a subset of the terms of h, we define the ordered set

$$h^C = \{q - h_i : 0 \leq i < n\}.$$

Then, choosing the negative of some element h_i is equivalent to choosing h_i^C in modulo q. So, we consider the subsets present in the ordered multiset

$$H = h \cup h^C = \{h_0, \dots h_{n-1}, h_0^C, \dots h_{n-1}^C\}$$

instead (technically speaking, h is not a set, but this notation treats its coefficients as one). Then we only need to choose either 0 or 1 for each element in H.

Running a SSP oracle here will return us a binary vector of $2n$ dimensions which we will call F_2'. Treating H as a vector (it is ordered), $\langle H, F_2' \rangle \bmod q$ will give some $-p, 0, p$. We obtain the n-dimensional ternary vector F' by taking $[x^i]F' = [x^i]F_2' - [x^{n+i}]F_2'$ for $0 \leq i < n$.

What we can do is to generate a list of all possible F' that pass, L_0. Then depending on which is computationally faster, we can do two things. Firstly, we can do the multiplication $h \star f'$and check for correctness ($f'$is the reverse of F'). If some corresponding $pg'$is found, $f'$is a key. Secondly, we can find a second list of possible F' that pass for the set H cycled by i, L_i (that is, treating H as a polynomial, $\langle x^i \star H, F' \rangle \bmod q = -p, 0, p$). Only common elements in L_0 and L_i need to be checked by multiplication; alternatively, any common element in $L_0, \dots, L_{n-1}$ will serve as a key.

Brute forcing for F_2' will be on the order of $O(2^{2n})$ since for each of the $2n$ elements we choose either 0 or 1. However, SSP is a highly studied problem and several ways have been discovered to decrease its runtime. In the following section, we provide a simple sketch of an algorithm based on a dynamic programming. In section 6, we will discuss ways to decrease computations.

5 A Sketch of a Python Dynamic Programming Algorithm Attack

This algorithm (Annex E) comprises of 2 segments. Let $\#(H)$ be the number of elements in the multiset H and $X_{sum} = k \cdot q + r$ for $0 \leq k \leq n$ and $r \in \{-3, 0, 3\}$. The first segment fills out a table called dp which has $\#(H)$ rows and $X_{sum} + 1$ columns using dynamic programming. Given a set S comprising of $n + 1$ elements (due to zero indexing on Python):

$$S = \{x_0, x_1, \dots, x_n\}.$$

$dp[i][j]$ is True if a sum j can be formed from a subset of S,

$$S_i = \{x_k : 0 \leq k \leq i\}, \text{ where } 0 \leq i \leq n$$

Otherwise, $dp[i][j]$ is False.

Before the program starts, all elements in the table dp are initialised as False. Afterwards, the program starts from $dp[0][0]$, increments through all possible j s before incrementing i once and repeats, until the whole table is filled out. The bottom right corner of dp , which is when the first segment terminates would be $dp[\#(H) - 1][X_{sum}]$ (due to zero indexing). The logic of the program is shown and explained in the left-side flow chart below (Fig. 1).

The information contained within dp will be important for the decision making and logic of the second segment of the algorithm, which will recursively traverse dp to find all possible subsets of H . First, the program checks whether or not $dp[\#(H) - 1][X_{sum}]$ is True. If it is False, there are no possible subsets and the program terminates immediately.

Now, we introduce a recursive function printSubsetsRec$(H, i, sumto, s)$. H is defined above; i is the index of the element it is currently examining; $sumto$ is the "distance" remaining to reach X_{sum} after subtracting the value of the elements it has considered already; and s is a list of indices of the elements that are considered as part of the sum.

After filling up the table dp , we run printSubsetsRec$(H, \#(H) - 1, X_{sum}, [\,])$, where [ ] is an empty list. Every other recursive call of printSubsetsRec( ) will then be derived from this parent function. The logic of the program is shown and explained in the right-side flow chart below (Fig. 2).

For both sections of the code, it will run through $3n$ possible values of X_{sum} (i.e. $-p/0/p + kq$ for $0 \leq k \leq n$). This is to lift the modulo q coefficients into the real numbers[1], while noting that the maximum bound of the sum of all terms in H is $n \cdot q$.

When the program returns the indices of a subset of H that sums to X_{sum}, we obtained a ternary vector (F') by the process described in section 4 .

Some conditions have also added on to the core algorithm to filter the answers obtained and reduce the number of paths the algorithm needs to go through before pruning a branch.

Every time we add a new element to s , check if $(i + n)$ mod $(\#H)$ not in s. In other words, check if the i - th element already has its complement in the set. If True, reject adding i to s . This ensures that we find unique ternary vectors for each value of X_{sum}.

[1] In other words, to bring the mod world into the real number world!

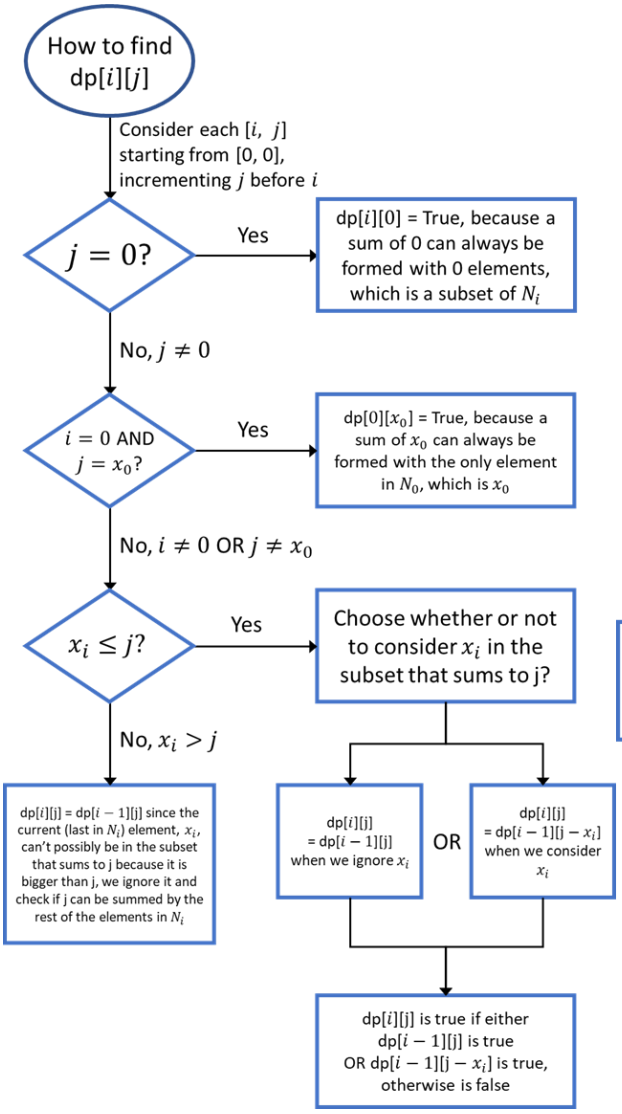

Fig. 1 (left): The flowchart that outlines and explains the main logic for assigning the values of $dp[i][j]$ in dp .

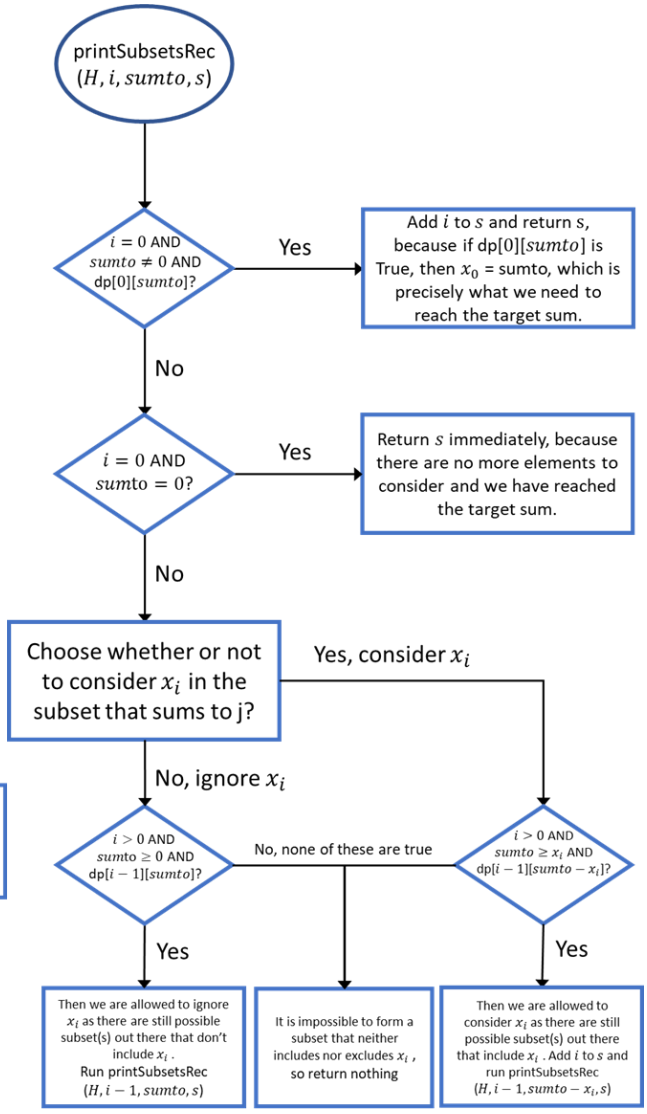

Fig. 2 (right): The flowchart that outlines and explains the main logic for the printSubsetsRec( ) function.

Discussion

Note that the algorithm and ideas above are not fully explored. In this section, we will briefly mention some possible addition exploration paths.

Our target sum ranges $-p/0/p + kq$ for $0 \leq k \leq n$. However, we can probably guess that the key will not exist in the subsets (if any) for $k = 0$, and similarly, for $k = n$. In fact, as the coefficients of h are, as far as we can tell, very close to random, the greatest number of subsets of H can be found when $k \approx n/2$. With that in mind, we can suspect the distribution of actual keys f would be greater for certain values of k . Hence, we can reduce the values k we need to check while still being able to find a key with high probability. Also, we do not actually need to run the program for all the targets $p, 0, -p$. Instead, just choosing one of the 3 targets would suffice for us to find a key.

This second exploration is of much greater significance. While the subset sum program is running, we should also take note of certain F' (or f') with short (albeit slightly longer) corresponding g'. We suspect that a small tweak of the Coppersmith attack [6] can be applied. As mentioned in [6], we can then construct a linear system of equations to recover the message with several longer g' (and error correcting codes). In other words, our algorithm need not find exact F', which will significantly increase the proportion of useful subsets. Doing a probability analysis on the average length of g' can give us an insight into the rate of success of finding useful F'.

Acknowledgements

We would like to thank our mentors, Isla and Jia Guang, for their constant guidance and patience. From helping us get familiar to NTRU, to the countless discussions over our new ideas, to finally proof reading our paper, we are truly grateful to them.

Many thanks to Ang Yong En, who served as both our student and friend, for his invaluable help and comments.

References

[1] Hoffstein, J., Pipher, J., & Silverman, J. H. (1998, June). NTRU: A ring-based public key cryptosystem. In International algorithmic number theory symposium (pp. 267-288). Springer, Berlin, Heidelberg.

[2] Chen, C., Danba, O., Hoffstein, J., Hülsing, A., Rijneveld, J., Schanck, J. M., ... & Zhang, Z. (2019). Algorithm specifications and supporting documentation. Brown University and Onboard security company, Wilmington USA.

[2] Chen, C., Danba, O., Hoffstein, J., Hülsing, A., Rijneveld, J., Schanck, J. M., ... & Zhang, Z. (2019). Algorithm specifications and supporting documentation. Brown University and Onboard security company, Wilmington USA.

[3] Hoffstein, J., Pipher, J., Silverman, J. H., & Silverman, J. H. (2008). An introduction to mathematical cryptography (Vol. 1). New York: springer.

[4] Galbraith, S. D. (2012). Mathematics of public key cryptography. Cambridge University Press.

[5] Lenstra, A. K., Lenstra, H. W., & Lovász, L. (1982). Factoring polynomials with rational coefficients. Mathematische annalen, 261(ARTICLE), 515-534.

[6] Coppersmith, D., & Shamir, A. (1997, May). Lattice attacks on NTRU. In International conference on the theory and applications of cryptographic techniques (pp. 52-61). Springer, Berlin, Heidelberg.

Annex A

Note that the points of intersection are not f ; but rather if $h = [2,4]$ or $h = [2,4,1]$, then the intersections represent the possible solutions vectors $[x, y] = [x^i \star Jf]$ and $[x, y, z] = [x^i \star Jf]$ respectively for some $0 \leq i < n$. Then $p \cdot g_{(i-1 \bmod n)} = \langle h, [x, y] \rangle \equiv 0 \bmod q$ and $p \cdot g_{i-1} = \langle h, [x, y, z] \rangle = 0$ respectively.

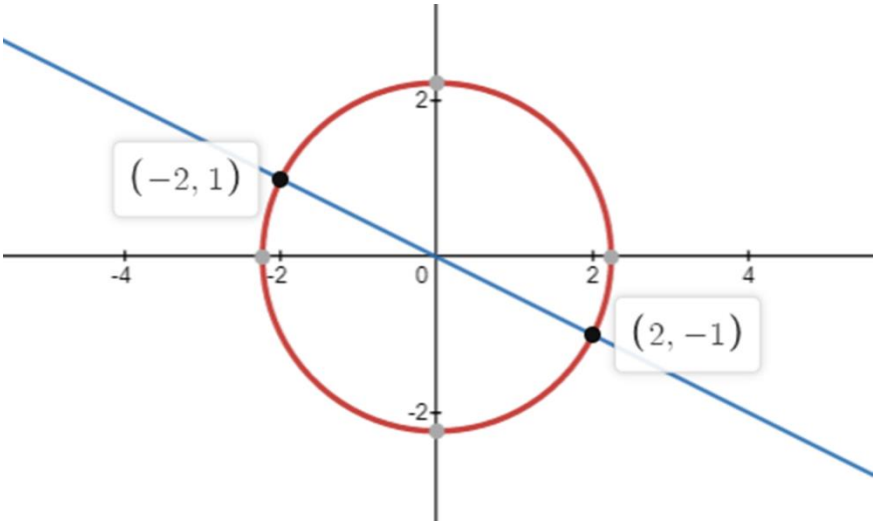

Fig. 3 (above):

$x^2 + y^2 = 5$ and $\langle [2,4], [x, y] \rangle = 2x + 4y = 0$ on Desmos.

Instead of finding all integer solutions on the circle, we cut down to finding integer solutions at the intersection of graphs (number of solutions reduced by 1 dimension).

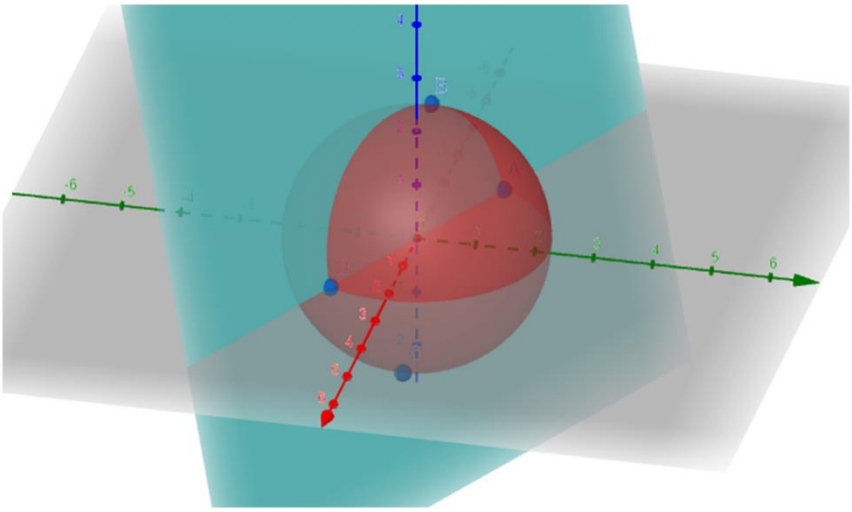

Fig. 4 (above):

$x^2 + y^2 + z^2 = 5$ and $2x + 4y + z = 0$ on GeoGebra (all integer solutions in dark blue).

Again, the second equation reduces the number of solutions by 1 dimension.

Annex B

A lattice L is spanned by a set of basis vectors, $\{v_1, \dots, v_n\}$. We define $\mathcal{M}^L$ to be the lattice's corresponding matrix, with columns as vectors. By definition, all lattice vectors are in the form of $w = \sum_{i=1}^{n} a_i v_i$. Below, we prove that if $\gcd(a_1, \dots, a_n) = 1$, then that vector is a basis vector in some basis that spans the lattice.

We know that

$$\gcd(a_1, \dots, a_n) = \gcd\big(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_{j-1}, a_{j+1}, \dots a_n, \gcd(a_i, a_j)\big),$$

$$\text{for } i, j \in \{1, \dots, n\}$$

This reduces finding the greatest common divisor of multiple elements to finding the greatest common divisor (gcd) of some 2 elements repeatedly. As seen in the Euclidean Algorithm, we know that $\gcd(\alpha, \beta) = \gcd(\alpha, \beta \bmod \alpha)$. Since we can express a gcd of multiple elements as gcd of pairs of numbers, we know that there exists at least one a_γ such that

$$a_\gamma \bmod a_1, \dots, a_{\gamma-1}, a_{\gamma+1}, \dots, a_n = 1,$$

which implies

$$1 = a_\gamma - k_1 a_1 - \dots - k_{\gamma-1} a_{\gamma-1} - k_{\gamma+1} a_{\gamma+1} - \dots - k_n a_n \text{ for some } k_i.$$

What we want to do is find some "path" from the row vector $(a_1, \dots, a_n)$ to the almost zero vector with a 1 in the i - th position, $(0, \dots, 1, \dots, 0)$. These coefficients are symbolic of a larger $n \times n$ matrix with the columns $a_i v_i$. Since we can find 1 as some linear combination of a_i, we know that we can achieve this "path" via addition (and subtraction) only; hence we can express this "path" as a transformation matrix (or as a composition of multiple simpler transformation matrices).

More specifically, if one of the steps in this "path" is to take $a_i - k_j a_j$, then this can be denoted by the $n \times n$ matrix $\mathcal{M}$ with 1 s on the main diagonal and $\mathcal{M}_{j,i} = -k_i$. Since multiplying a matrix by the identity matrix keeps the matrix the same, adding $\mathcal{M}_{j,i} = -k_i$ keeps all rows of the product the same except for the i - th row. This is equivalent to taking the entire j - th row and subtracting it off the i - th row (all other terms of the j - th row are 0).

After at most $(n-1)$ steps to make one of the entries of the row vector 1 , and another at most $(n-1)$ steps to make all other entries 0 , we have an upper bound of $2(n-1)$ steps, each represented by their own matrix.

We will be using the notation $\mathcal{M}^{it}$ to represent the i - th step of this transformation. The small "t" added is to indicated "transformation" and to ensure that this notation does not get confused with matrix exponentiation.

Keep in mind that matrix multiplication is not commutative. We let

$$\mathcal{M}^{1t} \dots \mathcal{M}^{nt} = \prod \mathcal{M}^{it}.$$

Then note that $(a_1, \dots, a_n)^T \prod \mathcal{M}^{it}$ gives the desire row vector of all 0 s and one 1 . Without loss of generality, we assume that the γ - th entry is 1 .

Then taking $\mathcal{M}^L \left(\left(\prod \mathcal{M}^{it} \right)^T \right)^{-1}$ will give us a lattice with the γ -th column as the chosen vector $w = \sum_{i=1}^{n} a_i v_i$.

Now, we just need to prove that what we get is a still a basis. Since all $\mathcal{M}^{it}$ are triangular matrices, $\det(\mathcal{M}^{kt}) = \pm 1$ for each k . Then since $\det(AB) = \det(A)\det(B)$, it follows that

$$\det\left(\prod \mathcal{M}^{it}\right) = \prod \det\left(\mathcal{M}^{it}\right) = \pm 1.$$

Hence $\mathcal{M}^L \left(\left(\prod \mathcal{M}^{it} \right)^T \right)^{-1}$ is also basis for the lattice L , containing w as one of its basis vectors.

QED.

Annex C

Lagrange-Gauss Algorithm:
Input: v_1, v_2 with $||v_1|| \geq ||v_2||$
Output: v_1, v_2, a new basis, with v_2 short

1. Take $v_1 = v_1 - \lfloor \mu_{1,2} \rceil v_2$, where $\mu_{i,j} = \frac{\langle v_i, v_j \rangle}{\langle v_j, v_j \rangle} = ||v_i|| \cos\theta$
2. If $||v_2|| > ||v_1||$:
    a. Swap v_1 and v_2
    b. Repeat from 1.
3. Else:
    a. Return v_1, v_2, with v_2 short

Annex D

LLL Algorithm:

```
# insert number of dimensions here
n =
# insert input matrix as a list of lists here, where each vector is an list
in the bigger list
v =

v_gs = [[0 for i in range(n)] for i in range(n)]
for i in range(n):
    for j in range(n):
        v_gs[i][j] = v[i][j]

v_rs = [[0 for i in range(n)] for i in range(n)]
for i in range(n):
    for j in range(n):
        v_rs[i][j] = v[i][j]

def dot(V1,V2):
    V12 = 0
    for i in range(n):
        V12 += V1[i] * V2[i]
    return V12

def sqmag(V):
    sqmagV = dot(V,V)
    return sqmagV

def proj(V2,V1):
    proj2on1 = dot(V1,V2)/sqmag(V1)
    return(proj2on1)

# gram schmidt vectors
def gsv(kv_gs,kv):
    for i in range(1,n):
        for j in range(i):
            for h in range(n):
                kv_gs[i][h] -= proj(kv[i],kv_gs[j]) * kv_gs[j][h]
    return kv_gs
```

```
# reduction step
def reductionstep(kv_rs,kv,kv_gs):
    for i in range(1,n):
        for j in range(i):
```

```
            for h in range(n):
                kv_rs[i][h] -= round(proj(kv[i],kv_gs[j]) + 0.001) *
kv_rs[j][h]
    return kv_rs

# updating v
def updatev(kv_rs):
    v = [[0 for i in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(n):
            v[i][j] = kv_rs[i][j]
    return v

k = 1
Counter = 0

# LLL
while k < n:
    print(k)
    v_gs = gsv(v_gs,v)
    reductionstep(v_rs,v,v_gs)

    # lovasz condition and swap
    if sqmag(v_gs[k]) >= (0.75 - (proj(v[k],v_gs[k-1]))**2) * sqmag(v_gs[k-
1]):
        k += 1
    else:
        middleman = v_rs[k]
        v_rs[k] = v_rs[k-1]
        v_rs[k-1] = middleman
        k = max(k-1,1)

    v = updatev(v_rs)
    v_gs = updatev(v_rs)
    Counter += 1

print('Counter = '+ str(Counter))
print('Reduced Matrix:')
for v in v_rs:
    print(str(v) + ',')
```

Annex E

SSP Algorithm:

```
# A Python program to count all subsets with given sum.
# dp[i][j] is going to store True if sum j is
# possible with Hay elements from 0 to i.
dp = [[]]

def dot(v1,v2):
     V12 = 0
     for i in range(n):
         V12 += v1[i] * v2[i]
     return V12

def display(v):
    reverse_f_prime = [0 for i in range(n)]
    for value in v:
        if value >= n:
            reverse_f_prime[value - n] -= 1
        else:
            reverse_f_prime[value] += 1

    # L_0 will contain all "candidates" of f_prime that need to be tested
    L_0.append(reverse_f_prime)

    # Prints out the solution if found, in our case we know the key
already
    # and assuming no other solutions exist
    if reverse_f_prime == reverse_f:
        print(reverse_f_prime)

# A recursive function to print all subsets with the
# help of dp[][]. list s[] stores current subset.
def printSubsetsRec(H, i, sumto, s):
    # If we reached end and sum is non-zero. We print
    # s[] only if H[0] is equal to sum OR dp[0][sum]
    # is True.
    if i == 0 and sumto != 0 and H[i] == sumto and ((i + n) % q) not in s
and i >= 0:
        s.append(i)
        display(s)
        s = []
        return
```

```
    # If sum becomes 0
    if (i == 0 and sumto == 0 and i >= 0):
        display(s)
```

```
        s = []
        return

    # If given sum can be achieved after ignoring
    # current element.
    if i > 0 and sumto >= 0:
        if (dp[i-1][sumto]):
            # Create a new list to store path
            b = []
            b.extend(s)
            printSubsetsRec(H, i-1, sumto, b)

    # If given sum can be achieved after considering
    # current element.
    if (sumto >= H[i] and dp[i-1][sumto-H[i]] and (i + n) % q) not in s and
i > 0:
        s.append(i)
        printSubsetsRec(H, i-1, sumto-H[i], s)

# Prints all subsets of H[0..H_length-1] with sum 0.
def printAllSubsets(H, H_length, sumto):
    if (H_length == 0 or sumto < 0):
        return

    # Sum 0 can always be achieved with 0 elements
    global dp
    dp = [[False for i in range(sumto+1)] for j in range(H_length)]

    for i in range(H_length):
        dp[i][0] = True

    # Sum H[0] can be achieved with single element
    if (H[0] <= sumto):
        dp[0][H[0]] = True

    # Fill rest of the entries in dp[][]
    for i in range(1, H_length):
        for j in range(0, sumto + 1):
            if (H[i] <= j):
                dp[i][j] = (dp[i-1][j] or dp[i-1][j-H[i]])
            else:
                dp[i][j] = dp[i - 1][j]
```

```
    if (dp[H_length-1][sumto] == False):
        print("There are no subsets with sum ", sumto)
        return

    # Now recursively traverse dp[][] to find all
    # paths from dp[H_length-1][sum]
```

```
    s = []
    printSubsetsRec(H, H_length-1, sumto, s)

k = 9
q = 128
h = [59,88,122,41,7,116,33,121,99,42,73,26,64,59,37,103,41,36,113]
h_comp = [q - i for i in h]
H = h + h_comp
n = len(h)
H_length = len(H)

L_0 = []

# This key and h were generated using a code and we read them in reverse
# so note that they are reversed
reverse_f = [0,0,1,0,0,-1,1,-1,1,0,-1,1,0,-1,-1,1,1,-1,-1]

# Considering all possibilities in mod world
for k in range(0,n+1):
    for adder in range(-3, 4, 3):
        sumto = k * q + adder
        print("sumto is now: " + str(sumto))
        # The program breaks if sumto is negative, so we must prevent that
        if sumto < 0:
            sumto = 0
        printAllSubsets(H,H_length,sumto)
print("bye")

# This code is adapted from Lovely Jain
```