Heuristically Secure Threshold Lattice-Based Cryptography Schemes

James D Dalton

A thesis submitted to the Graduate Faculty of

JAMES MADISON UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Department of Computer Science

May 2021

Table of Contents

List of Definitions

List of Examples

List of Algorithms

Abstract

In public-key encryption, a long-term private key can be an easy target for hacking and deserves extra protection. One way to enhance its security is to share the long-term private key among multiple (say n) distributed servers; any threshold number $(t, t \leq n)$ of these servers are needed to collectively use the shared private key without reconstructing it. As a result, an attacker who has compromised less than t servers will still not be able to reconstruct the shared private key.

In this thesis, we studied threshold decryption schemes for lattice-based public-key encryption, which is one of the most promising post-quantum public-key encryption schemes. We developed threshold decryption schemes for Stinson's, the standard NTRU, and NTRU with Ring Learning with Errors (R-LWE) cryptosystems. Prototype implementations were developed for validating the functionality of these threshold decryption schemes. Our designs achieve heuristic security, and its security is supported by mechanisms similar to that of R-LWE.

Chapter 1

Introduction

In the setting of public-key cryptosystem, a user, Bob, first generates a public/private key pair and then distributes his public key to other people. Bob's private key must be kept confidential and if necessary, online for decrypting incoming ciphertext or digitally signing messages on the fly. Such an online, long-term secret is an attractive target for hacking and given enough time and efforts, its compromise seems inevitable.

One way to mitigate the threat against the online, long-term private key is to share it among multiple (say n, n is an integer) distributed servers in such a way that any threshold number (say t, $t \leq n$) can work together to collectively use the shared private key without actually reconstructing it in the process. The collective computation by the willing participating servers can be either decryption or digital signing. This cryptographic scheme is called threshold decryption or threshold digital signing correspondingly. Together they are called threshold cryptography [8, 7, 9, 12, 14, 13, 32].

It is worth noting that threshold cryptography is different from threshold secret sharing [30, 3] in important ways. In both, a long-term secret is shared among n parties in a way that any t or more of them are capable of working together to reconstruct the shared secret. In threshold secret sharing, the shared secret is indeed reconstructed in its use and the reconstruction point will be a single point of attack. In threshold cryptography, however, the shared secret is never reconstructed and thus there is no single point of attack. In some sense, threshold cryptography is security extension of threshold secret sharing.

Problem Statement

Most existing popular public-key encryption schemes, including RSA [29], ElGamal [11], Elliptic-curve [21, 24], are vulnerable to attacks from general quantum computers running appropriate quantum algorithms [31, 25].

One of the most promising public-key cryptosystems that will be secure in post-quantum era is based on lattices [18, 26, 23, 28, 34]. Example lattice-based cryptosystems include NTRUEncrypt [18], Stinson's NTRU scheme [34], learning with error (LWE) [27], and

NTRU with R-LWE [33].

How can we add threshold mechanisms to these lattice-based cryptosystems?

Overview

The remaining of this thesis is organized as follows. In Chapter 2, we shall review the basic concepts and building blocks of lattice-based cryptosystems. Our review will come with small concrete examples. In Chapter 3, we shall present our threshold lattice cryptosystems, including threshold Stinson's NTRU, threshold NTRUEncrypt, and threshold NTRU with R-LWE. Concrete examples will be used in the description of these threshold schemes.

Security analysis of these schemes and related work will be provided in 4. Concluding remarks of this thesis will be given in Chapter 5.

Chapter 2

Building Blocks

Rings

A ring is a set with two operations, addition and multiplication, that has special properties. The set of integers ($\mathbb{Z}$) is a ring, as is $\mathbb{Z}[x]$, the set of polynomials with integer coefficients.

Definition 2.1 (Ring)**.** *The set R is a ring if it has binary operations for addition and multiplication defined that meet the following axioms.*

- *Addition is associative.*

$$(a+b)+c = a+(b+c)$$

- *Addition is commutative.*

$$a+b = b+a$$

- *An additive identity exists.*

$$a+0 = a$$

- *An additive inverse exists.*

$$a+(-a) = 0$$

- *Multiplication is associative.*

$$(ab)c = a(bc)$$

- *A multiplicative identity exists.*

$$1a = a$$

- *Multiplication is distributive with respect to addition.*

$$a(b+c) = ab+ac$$

$$(a+b)c = ac+bc$$

Definition 2.2 (Ideal Ring). *Let R be a ring. I is an ideal ring if for any $x, y \in I$ and $r \in R$*

$$x + y \in I$$

$$rx \in I$$

Definition 2.3 (Quotient Ring). *Let R be a ring and I be an ideal of that ring. The quotient ring is R/I.*

If $\mathbb{Z}[x]$ is a ring, n is an integer and $x^n - 1$ is an ideal of that ring, the quotient ring is $\mathbb{Z}[x]/(x^n - 1)$. Another way to view it is the polynomials in the quotient ring are the polynomials in R mod I.

Gaussian Distributions

A Gaussian distribution is also known as a normal distribution or bell curve. A value sampled from a Gaussian distribution is expected to be within one standard deviation from the mean 68% of the time. It will be within two standard deviations 95% of the time and 99.7% of the time will be within three standard deviations.

Lattices

A vector is an ordered tuple of values where the values are real numbers ($\mathbb{R}$). Let n be an integer, $\mathbb{R}^n$ is a vector of n real numbers. A vector written horizontally is called a row vector, written vertically it is a column vector. Two vectors can be added and multiplied. Addition is performed by adding the matching indices

$$\begin{aligned} a &= \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \\ b &= \begin{bmatrix} b_1 & b_2 & \dots & b_n \end{bmatrix} \\ a + b &= \begin{bmatrix} a_1 + b_1 & a_2 + b_2 & \dots & a_n + b_n \end{bmatrix} \end{aligned}$$

Scalar or dot product multiplication is performed by multiplying the matching indices and summing the results.

$$a \cdot b = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

Definition 2.4 (Linearly Independent)**.** *A set of vectors $(v_1, v_2, \ldots v_n)$ is considered linearly independent if for all scalar values $a_1, a_2, \ldots a_n$ the equation*

$$a_1v_1 + a_2v_2 + \ldots + a_nv_n = 0$$

only if $a_1, a_2, \ldots, a_n = 0$.

Definition 2.5 (Basis)**.** *A basis of $\mathbb{R}^n$ is a set of linearly independent vectors where any vector in $\mathbb{R}^n$ can be expressed as a linear combination of the basis vectors.*

The vectors $v_1 = \begin{bmatrix} 1 & 0 \end{bmatrix}$ and $v_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$ are a basis for $\mathbb{R}^2$ since any vector $\begin{bmatrix} a & b \end{bmatrix} \in \mathbb{R}^2$ can be written as $av_1 + bv_2$. The basis can be written as a matrix where each vector is a row

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Definition 2.6 (Determinant)**.** *The determinant of a matrix is a scalar measurement of a square matrix. The determinate of a matrix (M) is denoted by det(M), $|M|$ or $||M||$. The determinate is calculated as the sum of the products of the permutation of row and column values. The permutations take one and only one value from each row and column.*

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$|A| = aei + bgf + cdh - ceg - bdi - afh$$

A matrix with a determinate of ± 1 is called a unimodular matrix. Unimodular operations on a basis matrix produce another basis matrix. The unimodular operations are

- Multiply any row by -1.
- Interchange any two rows.
- Add an integral multiple of any row to another row.

Definition 2.7 (Lattice)**.** *A lattice is the set of all liner combinations of basis vectors with integral coefficients.*

What makes lattices of interest in cryptography are two problems, the Shortest Vector (SVP) and Closest Vector (CVP) problems.

Definition 2.8 (Shortest Vector Problem). *Given a basis for a lattice $\mathcal{L}$ in $\mathbb{R}^n$. Find a vector $v \in \mathcal{L}$ where v is not all 0 such that $||v||$ is minimized. The vector v is the shortest vector in $\mathcal{L}$*

Definition 2.9 (Closest Vector Problem). *Given a basis for a lattice $\mathcal{L}$ in $\mathbb{R}^n$ and a vector $w \in \mathbb{R}^n$ that is not in $\mathcal{L}$. Find a vector $v \in \mathcal{L}$ such that $||w - v||$ is minimized. The vector v is the closest vector to w in $\mathcal{L}$.*

The CVP is known to be NP-hard, and SVP is NP-hard under certain conditions when the vectors are randomized and independent with uniform distributions chosen over all vectors in $\mathbb{Z}^n$ [2].

LLL

The Lenstra-Lenstra-Lovàsz (LLL) algorithm 2.1 is a lattice basis reduction algorithm [22] that can be used to solve the SVP. The LLL algorithm performs a series of reduce and exchange steps until it meets the definition of an LLL-reduced lattice. The definition allows for getting the exact shortest vector or an approximation of it. Getting an approximation is guaranteed to complete in polynomial time, however, getting the exact vector is not guaranteed to complete.

In the reduction step it makes use of the Gram-Schmidt process in reducing the matrix.

Definition 2.10 (Gram Matrix). *The Gram matrix $\Delta(L)$ of a lattice L is the matrix where the (i, j) entry is the scalar product of the i-th and j-th basis vectors.*

Definition 2.11 (Gram-Schmidt Process). *The Gram-Schmidt process takes a basis and generates an orthogonal basis.*

- *Let $v_1, v_2, \ldots, v_n$ form a basis.*
- *Let the orthogonal basis be $v_1^*, v_2^*, \ldots, v_n^*$.*
- *Calculate v_i^* by*

$$v_1^* = v_1$$

For $2 \leq i \leq n$:

$$\mu_{ij} = \frac{v_i \cdot v_j^*}{v_j^* v_j^*}$$

$$v_i* = v_i - \sum_{j=1}^{i-1} u_{ij} \cdot v_i^*$$

The Gram-Schmidt coefficient is μ_{ij}.

Definition 2.12 (LLL Reduced)**.** *A basis for a lattice is LLL-reduced with parameter α if*

- $\frac{1}{4} < \alpha \leq 1$
- $|\mu_{ij}| \leq \frac{1}{2}$ *for* $1 \leq j < i \leq n$
- $|v_I^* + \mu_{i,i-1} v_{i-1}^* \geq \alpha |v_{v-1}^*|^2$ *for* $2 \leq i \leq n$

The standard value for α is $\frac{3}{4}$. The value of α determines how reduced the basis is, the higher the value the more reduced a basis produced. For values of $\alpha < 1$, the algorithm is guaranteed to complete in polynomial time. A value of 1 will produce the shortest vector but is not guaranteed to complete.

Learning With Errors

Given a system of linear equations of n variables, it is possible to find solutions efficiently. Injecting a bit of randomness to the equations makes it more difficult to solve. Based on this, Regev introduced Learning With Errors (LWE) [27].

Definition 2.13 (Learning With Errors)**.** *Let $n \geq 1$ and p be prime. The secret $s \in \mathbb{Z}_p^n$. Choose a_i independently and uniformly over $\mathbb{Z}_p^n$. With the probability distribution $\mathcal{X} : \mathbb{Z}_p \to \mathbb{R}^+$ on $\mathbb{Z}_p$. Chose $e_i \in \mathbb{Z}_p$ independently according to $\mathcal{X}$.*

$$b_i = \langle s \cdot a_i \rangle + e_i$$

Given a_i and b_i, determine s.

The value of p, n and $\mathcal{X}$ must be chosen correctly. If not, security may be compromised, or decryption may not be possible. The value of p must be significantly greater than n and $\mathcal{X}$ should be a Gaussian distribution. Regev proved that for certain choices of p and $\mathcal{X}$ the average-case solution is based on the worst-case lattice problem, Gap Shortest Vector, which is similar to the SVP. This implies a quantum solution to the problem.

Algorithm 2.1 LLL

```
procedure REDUCE($k, l$)
    if $|\mu_{kl}| > \frac{1}{2}$ then
        $y_k \leftarrow y_k - \lceil \mu_{kl} \rfloor y_l$
        for $j \leftarrow (1, 2, \ldots, l-1)$ do
            $\mu_{kj} \leftarrow \mu_{kj} - \lceil \mu_{kl} \rfloor \mu_{lj}$
        $\mu_{kl} \leftarrow \mu_{kl} - \lceil \mu_{kl} \rfloor$
procedure EXCHANGE($k$)
    $y_k, y_{k-1} \leftarrow y_{k-1}, y_k$
    $v \leftarrow \mu_{k,k-1}$; $\delta \leftarrow \gamma_k + v^2 \gamma_{k-1}$
    $\mu_{k,k-1} \leftarrow v\gamma_{k-1}/\delta$
    $\gamma_k \leftarrow \gamma_k \gamma_{k-1}/\delta$; $\gamma_{k-1} \leftarrow \delta$
    for $j \leftarrow (1, 2, \ldots, k-2)$ do
        $\mu_{kj}, \mu_{k-1,j} \leftarrow \mu_{k-1,j}, \mu_{kj}$
    for $i \leftarrow (k+1, \ldots, n)$ do
        $\xi \leftarrow \mu_{ik}$; $\mu_{ik} \leftarrow \mu_{i,k-1} - v\mu_{ik}$
        $\mu_{i,k-1} \leftarrow \mu_{k,k-1}\mu_{ik} + \xi$
Input: a basis $y_1, y_2, \ldots, y_n$ of the lattice $\mathcal{L} \subset \mathbb{R}^n$ and $\alpha \in \mathbb{R}$ where $\frac{1}{4} < \alpha < 1$
for $i \leftarrow (1, 2, \ldots, n)$ do
    $y_i^* \leftarrow y_i$
    for $j \leftarrow (1, 2, \ldots, i-1)$ do
        $\mu_{ij} \leftarrow (y_i \cdot y_i^*)/\gamma_j$
        $\gamma_i \leftarrow y_i^* \cdot y_i^*$
    $k \leftarrow 2$
    while $k \leq n$ do
        REDUCE($k, k-1$)
        if $\gamma_k \geq (\alpha - \mu_{k,k-1}^2)\gamma_{k-1}$ then
            for $l \leftarrow (k-2, \ldots, 1)$ do
                REDUCE($k, l$)
            $k \leftarrow k+1$
        else
            EXCHANGE($k$)
            if $k > 2$ then
                $k \leftarrow k-1$
```

Example 2.1 (LLL).

k	*Action*	y	μ	y^*	γ
1	*Init*	$\begin{bmatrix}1&1&1\\-1&0&2\\3&5&6\end{bmatrix}$	$\begin{bmatrix}1&0&0\\\frac{1}{3}&1&0\\\frac{14}{3}&\frac{13}{14}&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\-\frac{4}{3}&-\frac{1}{3}&\frac{5}{3}\\-\frac{6}{14}&\frac{9}{14}&-\frac{3}{14}\end{bmatrix}$	$\begin{bmatrix}3\\\frac{14}{3}\\\frac{9}{14}\end{bmatrix}$
2	$\lvert\mu_{21}\rvert = \frac{1}{3} < \frac{1}{2}$[2.1.2] $\frac{14}{3} > \frac{8}{3}$[2.1.25] *Reduce*	$\begin{bmatrix}1&1&1\\-1&0&2\\4&5&4\end{bmatrix}$	$\begin{bmatrix}1&0&0\\\frac{1}{3}&1&0\\\frac{14}{3}&\frac{13}{14}&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\-\frac{4}{3}&-\frac{1}{3}&\frac{5}{3}\\-\frac{6}{14}&\frac{9}{14}&-\frac{3}{14}\end{bmatrix}$	$\begin{bmatrix}3\\\frac{14}{3}\\\frac{9}{14}\end{bmatrix}$
3	$\lvert\mu_{32}\rvert = \frac{13}{14} > \frac{1}{2}$[2.1.2] *Reduce*	$\begin{bmatrix}1&1&1\\-1&0&2\\4&5&4\end{bmatrix}$	$\begin{bmatrix}1&0&0\\\frac{1}{3}&1&0\\\frac{13}{3}&-\frac{1}{14}&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\-\frac{4}{3}&-\frac{1}{3}&\frac{5}{3}\\-\frac{6}{14}&\frac{9}{14}&-\frac{3}{14}\end{bmatrix}$	$\begin{bmatrix}3\\\frac{14}{3}\\\frac{9}{14}\end{bmatrix}$
	$\frac{9}{14} < \frac{910}{196}$[2.1.25] *Exchange*	$\begin{bmatrix}1&1&1\\4&5&4\\-1&0&2\end{bmatrix}$	$\begin{bmatrix}1&0&0\\\frac{13}{3}&1&0\\\frac{1}{3}&-\frac{1}{14}&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\-\frac{1}{3}&\frac{2}{3}&-\frac{1}{3}\\\frac{44}{42}&\frac{1}{42}&-\frac{82}{42}\end{bmatrix}$	$\begin{bmatrix}3\\\frac{2}{3}\\\frac{8661}{1764}\end{bmatrix}$
2	$\lvert\mu_{21}\rvert = \frac{13}{3} > \frac{1}{2}$[2.1.2] *Reduce*	$\begin{bmatrix}1&1&1\\0&1&0\\-1&0&2\end{bmatrix}$	$\begin{bmatrix}1&0&0\\-\frac{1}{3}&1&0\\\frac{1}{3}&-\frac{1}{14}&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\-\frac{1}{3}&\frac{2}{3}&-\frac{1}{3}\\\frac{44}{42}&\frac{1}{42}&-\frac{82}{42}\end{bmatrix}$	$\begin{bmatrix}3\\\frac{2}{3}\\\frac{8661}{1764}\end{bmatrix}$
	$\frac{2}{3} < \frac{8}{3}$[2.1.25] *Exchange*	$\begin{bmatrix}0&1&0\\1&1&1\\-1&0&2\end{bmatrix}$	$\begin{bmatrix}1&0&0\\1&1&0\\0&1&1\end{bmatrix}$	$\begin{bmatrix}0&1&0\\1&0&1\\-2&-1&1\end{bmatrix}$	$\begin{bmatrix}1\\2\\6\end{bmatrix}$
2	$\lvert\mu_{21}\rvert = 1 > \frac{1}{2}$[2.1.2] *Reduce*	$\begin{bmatrix}0&1&0\\1&0&1\\-1&0&2\end{bmatrix}$	$\begin{bmatrix}1&0&0\\0&1&0\\0&1&1\end{bmatrix}$	$\begin{bmatrix}0&1&0\\1&0&1\\-2&-1&1\end{bmatrix}$	$\begin{bmatrix}1\\2\\6\end{bmatrix}$
	$2 < 1$[2.1.25] *Exchange*	$\begin{bmatrix}0&1&0\\1&0&1\\-1&0&2\end{bmatrix}$	$\begin{bmatrix}1&0&0\\0&1&0\\0&1&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\1&0&1\\-2&-1&1\end{bmatrix}$	$\begin{bmatrix}1\\2\\6\end{bmatrix}$
3	$\lvert\mu_{32}\rvert = 1 > \frac{1}{2}$[2.1.2] *Reduce*	$\begin{bmatrix}0&1&0\\1&0&1\\-2&0&1\end{bmatrix}$	$\begin{bmatrix}1&0&0\\0&1&0\\0&-1&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\1&0&1\\-2&1&1\end{bmatrix}$	$\begin{bmatrix}1\\2\\6\end{bmatrix}$
	$6 > 4$[2.1.25] *Reduce*	$\begin{bmatrix}0&1&0\\1&0&1\\-1&0&2\end{bmatrix}$	$\begin{bmatrix}1&0&0\\0&1&0\\0&-1&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\1&0&1\\-2&1&1\end{bmatrix}$	$\begin{bmatrix}1\\2\\6\end{bmatrix}$
4	$k > n$[2.1.23]	$\begin{bmatrix}0&1&0\\1&0&1\\-1&0&2\end{bmatrix}$	$\begin{bmatrix}1&0&0\\0&1&0\\0&1&1\end{bmatrix}$	$\begin{bmatrix}1&1&1\\1&0&1\\-2&-1&1\end{bmatrix}$	$\begin{bmatrix}1\\2\\6\end{bmatrix}$

Ring Learning With Errors

Ring Learning With Errors (R-LWE) [23] is similar to LWE, but uses polynomial rings.

Definition 2.14 (Ring Learning With Errors)**.** *Let n be the degree of the polynomials, $q = 1 \mod 2n$ where the coefficients of the polynomial are mod q. Let $\Phi(x)$ be an irreducible polynomial. All polynomials used are from the finite quotient ring $\mathbb{Z}_q(x)/\Phi(x)$. The secret s is a small unknown polynomial. Let $a_i(x)$ be a set of random known polynomials and $e_i(x)$ be a set of small random unknown polynomials.*

$$b_i(x) = \langle a_i(x) \cdot s(x) \rangle + e_i(x)$$

Given a_i and b_i determine s.

For the correct values of $\Phi(x)$, n and q Regev proved the average-case solution is also based on the Gap Shortest Vector problem.

Standard NTRU

NTRU was first introduced at the rump session of Crypto'96 [15] by Hoffstein, Pipher and Silverman. It uses polynomial rings for construction and lattices for the security proof. It was presented as being quantum safe. At EUROCRYPT'97, the original version was proven to not be as secure as claimed [6]. At ANTS'98 an updated version was presented that addressed problems in the original. Another update was published in 2009 [18] and is part of the IEEE P1363 Standard [1]. Although the standard has gone inactive, there has been academic work on standard NTRU in recent years. Two NTRU variants of interest are one that was introduced at Eurocrypt 2011 [33] which added R-LWE, and one introduced by Stinson and Paterson in *Cryptography: Theory and Practice* [34].

All three use a technique called center mod or mods.

Definition 2.15 (Center Mod or Mods)**.** *Take the normal modulus and map it to the interval $[-q/2, q/2]$. If the result of the modulus is greater than $q/2$ subtract q from modulus result to get the centered mod.*

Definition 2.16 (Standard NTRU)**.** *Standard NTRU has three public parameters, N a positive integer, is the degree of polynomials, q is a large modulus and p a small modulus.*

The values of q and p should be co-prime and $q >> p$. The standard sets $p = 3$. The polynomial ring is defined as $\mathcal{R} = \mathbb{Z}[X]/(X^N - 1)$.

- *NTRU.KeyGen: let f and g be two small random polynomials with coefficients drawn from $\{-1, 0, 1\}$.*

  Let f_q be f inverted in $\mathcal{R}_q$ and f_p be f inverted in $\mathcal{R}_p$.

  If f is not invertible or g is not invertible in $\mathcal{R}_q$, choose new polynomials.

  The private key is the pair (f, f_p).

  The public key is $h = p * g * f_q \pmod{\text{s}\, q}$

- *NTRU.Encrypt: Let r be a small random polynomial. Let m be the messages. The ciphertext y is calculated as*

$$y = r * h + m \pmod{q}.$$

- *NTRU.Decrypt:*

$$a = f * y \ (\mathrm{mods}\, q)$$
$$m = f_p * a \ (\mathrm{mods}\, p)$$

A 2017 paper [19], describes choosing the parameters for security and to avoid decryption failures. For certain combinations of parameters, decryption failure is possible and weakens the security of NTRU [20] with those parameters. Using the parameter sets from the previously mentioned paper can reduce the chance of failure and maintain a given security level.

Bits of Security	N	q
128	439	2048
256 (optimized for key size)	1087	2048
256 (optimized for encryption/decryption)	1499	2048
256	1171	2048

Example 2.2 (Standard NTRU).

Parameters

$$N = 11,\ p = 3 \text{ and } q = 32$$

Calculate the keys

$$f = -x^{10} + x^9 + x^6 - x^4 + x^2 + x - 1$$
$$g = -x^{10} - x^8 + x^5 + x^3 + x^2 - 1$$
$$f_p = 2x^9 + x^8 + 2x^7 + x^5 + 2x^4 + 2x^3 + 2x + 1$$
$$f_q = 30x^{10} + 18x9 + 20x^8 + 22x^7 + 16x^6 + 15x^5 + 4x^4 + 16x^3 + 9x + 5$$
$$h = 16x^{10} - 13x^9 + 12x^8 - 13x^7 + 15x^6 - 8x^5 + 12x^4 - 12x^3 - 10x^2 - 7x + 8$$

Encryption

$$m = x^{10} + x^9 - x^8 - x^4 + x^3 - 1$$
$$r = -x^7 - x^5 + x^4 + x^3 + x^2 - 1$$
$$y = 19x^{10} + 6x^9 + 25x^8 + 7x^7 + 30x^6 + 16x^5 + 14x^4 + 24x^3 + 26x^2 + 11x + 14$$

Decryption

$$a = 7x^{10} - 3x^9 + 5x^8 + 7x^7 + 6x^6 + 7x^5 + 10x^4 - 11x^3 - 10x^2 - 7x + 3$$
$$m = x^{10} + x^9 - x^8 - x^4 + x^3 - 1$$

Definition 2.17 (Stinson's NTRU). *Stinson's version uses the same definition for the public parameters, with the same recommendations.*

- *SNTRU.KeyGen: Let F and G be polynomials of degree N with coefficients from* $\{-1, 0, 1\}$

  $f = pF + 1$

  $g = pG$

  Let f^{-1} be f inverted in $\mathcal{R}_q$

  The private key is f.

  The public key is $h = f^{-1}g \pmod{\text{mods}\, q}$

  The values of F, G, and g are not needed after key generation, but should be kept secret.

- *SNTRU.Encrypt: Let r be a small random polynomial. Let m be the messages. The ciphertext y is calculated as*

$$y = r * h + m \ (\text{mods}\, q)$$

- *SNTRU.Decrypt:*

$$a = f * y \;(\operatorname{mods} q)$$

$$m = a \;(\operatorname{mods} p)$$

Having $f = pF + 1$ allows the decryption process to be more efficient.

Example 2.3 (Stinson's NTRU).

Parameters

$$N = 23,\ p = 3 \text{ and } q = 31$$

Calculate the keys

$$\begin{aligned}
F &= x^{18} - x^9 + x^8 - x^4 - x^2 \\
f &= 3x^{18} - 3x^9 + 3x^8 - 3x^4 - 3x^2 + 1 \\
G &= x^{17} + x^{12} + x^9 + x^3 - x \\
g &= 3x^{17} + 3x^{12} + 3x^9 + 3x^3 - 3x \\
h &= -13x^{22} - 15x^{21} + 12x^{19} - 14x^{18} + 8x^{16} - 14x^{15} - 6x^{14} + 14x^{13} - 3x^{12} + 7x^{11} - 5x^{10} \\
&\quad - 14x^9 + 3x^8 + 10x^7 + 5x^6 - 8x^5 + 4x^2 + x + 8
\end{aligned}$$

Encryption

$$\begin{aligned}
m &= x^{15} - x^{12} + x^7 - 1 \\
r &= x^{19} + x^{10} + x^6 - x^2 \\
y &= 5x^{22} - 15x^{21} + 4x^{20} + 8x^{19} + 10x^{18} - 15x^{17} + 6x^{16} + 8x^{15} - 8x^{14} + 3x^{13} - 10x^{12} \\
&\quad - 7x^{11} - x^{10} - 9x^9 + 12x^8 - 14x^7 + 15x^6 - 10x^5 + 15x^4 - 14x^3 - 5x^2 - 15x - 3
\end{aligned}$$

Decryption

$$\begin{aligned}
a &= 6x^{22} + 3x^{21} - 6x^{20} - 3x^{19} - 3x^{17} + 7x^{15} + 6x^{13} - x^{12} - 9x^{11} + 3x^{10} + 3x^9 \\
&\quad - 5x^7 + 6x^4 + 3x^3 + 6x^2 - 3x + 5 \\
m &= x^{15} - x^{12} + x^7 - 1
\end{aligned}$$

NTRU with R-LWE is a provably secure version. Changes are made to the algorithm that make it compatible with R-LWE, which leads to its proof of security.

Definition 2.18 (NTRU With R-LWE). *Parameters for NTRU with R-LWE or provably secure NTRU (pNE) have some overlap with the other two but adds additional public parameters.*

- *Public Parameters [5]:*

  N - a power of 2 greater than 8.

  q - a large prime. To guarantee decryption $q \in [dn^6 ln(n), 2dn^6 ln(n)]$ where $d > 512$ and $q = 1 \mod 2n$.

  p - 2

  α - A small standard deviation used to generate coefficients for the error polynomials. $\alpha = \sqrt{2n/\pi}$

  σ - A larger standard deviation used to generate coefficients for the keys. $\sigma = 2n\sqrt{ln(8nq)q}$

  $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ - The change to $X^N + 1$ is from R-LWE, and being an irreducible polynomial is part of what makes it provably secure.

- *pNE.KeyGen: Let F and g be polynomials of degree N with coefficients sampled from a discrete Gaussian distribution with standard deviation σ.*

  $f = pF + 1$

  Let f^{-1} be f inverted in $\mathcal{R}_q$, if not resample F.

  Let g be a polynomial invertible in $\mathcal{R}_q$

  The private key is f.

  The public key is $h = pf^{-1}g \pmod{\text{s}\, q}$

  The values of F, and g are not needed after key generation but should be kept secret.

- *pNE.Encrypt: Let r and e be a small random polynomial whose coefficients are sampled form a discrete Gaussian distribution with standard deviation α. Let m be the messages. The ciphertext y is calculated as*

$$y = r * h + p * e + m \ (\operatorname{mods} q)$$

  Including p+e in the encryption is also from R-LWE and contributes to the provability of the security.

- *pNE.Decrypt:*

$$a = f * y \ (\operatorname{mods} q)$$

$$m = a \ (\operatorname{mods} p)$$

Example 2.4 (pNE).

Parameters

$$N = 16$$

$$q = 37$$

$$\alpha = 1$$

$$\sigma = 1$$

Calculate the keys

$$\begin{aligned}
F &= x^{15} - x^{13} - x^{12} - x^9 - x^8 - x^7 - x^5 - x^4 - x^2 \\
f &= 2x^{15} - 2x^{13} - 2x^{12} - 2x^9 - 2x^8 - 2x^7 - 2x^5 - 2x^4 - 2x^2 + 1 \\
g &= x^{15} + 2x^{14} - x^{13} - x^{11} - x^{10} + x^9 - 2x^7 + x^6 + x^5 - x^2 + x \\
h &= -12x^{15} + 3x^{14} + 2x^{12} - 15x^{11} - 14x^{10} + 8x^9 + 17x^8 + 15x^7 + 3x^6 + 7x^5 + 17x^4 \\
&\quad + 13x^3 - 18x^2 - 10x + 8
\end{aligned}$$

Encryption

$$\begin{aligned}
m &= x^{15} + x^{12} + x^7 + 1 \\
r &= -x^{15} - x^{13} + x^{12} + x^{11} - x^{10} - x^9 - x^4 + 2x^3 - x^2 - x \\
e &= -x^{15} + x^{12} - 2x^{11} + 2x^{10} - x^9 + x^8 - x^7 - x^5 + 2 \\
y &= 8x^{15} - 12x^{14} - 12x^{13} - 15x^{12} + 2x^{10} + 4x^9 - 8x^7 - 8/x^5 - 18x^4 + 14x^3 - 18x^2 \\
&\quad - 5x - 5
\end{aligned}$$

Decryption

$$\begin{aligned}
a &= 5x^{15} - 6x^{14} + 2x^{13} - 3x^{12} - 12x^{10} - 4x^9 + 6x^8 - 15x^7 + 4x^6 - 12x^5 - 10x^4 + 10x^3 \\
&\quad - 14x^2 - 10x + 9 \\
m &= x^{15} + x^{12} + x^7 + 1
\end{aligned}$$

NTRUSign

In 2001 some of the same people involved in standard NTRU published NTRUSign [17] also based on polynomials and lattices. Unfortunately, it was quickly discovered that it could be broken with far too few transcripts. An update was published in 2003 [16], but that was also shown to broken with too few transcripts [10]. The 2003 version of the algorithm is included for completeness.

Definition 2.19 (NTRUSign)**.** *NTRUSign is a digital signature algorithm with four public parameters, N the degree of the polynomials, q the coefficients modulus, $\mathcal{N}$ a norm bound used to verify the signature and β a balancing factor where $0 < \beta \leq 1$.*

- *NTRUSign.KeyGen: The private keys are two small random polynomials, f and g, that are invertible in $\mathcal{R}_q$. For the public key, first find two polynomials F and G such that*

  $$f * G - g * F = q.$$

  The public key is

  $$h = F * f_q \pmod{q} = G * g_q \pmod{q}$$

- *NTRUSign.Sign: To sign a document map it to a vector $m \in [0, q)^N$ using an agreed upon hash function. Then set*

  $$(x, y) = (0, m) \begin{pmatrix} G & -F \\ -g & f \end{pmatrix} /q = \left(\frac{-m * g}{q}, \frac{m * f}{q} \right)$$

  Then let

  $$\epsilon = -x \text{ and } \epsilon` = -y.$$

  The signature s is calculated as

  $$s = \epsilon f + \epsilon` g$$

- *NTRUSign.Verify: To verify map the document to a vector m the same as it was for signing. Then calculate*

  $$t = s * h \mod q.$$

  Then calculate the norm

  $$v = \min_{k_1, k_2 \in \mathcal{R}} (||s + k_1 q||^2 + \beta^2 ||(t - m) + k_2 q||^2)^{1/2}.$$

  If $v \leq \mathcal{N}$ the signature is verified.

Example 2.5 (NTRUSign).

Parameters

$$N = 11, q = 32, \beta = 0.38 \text{ and } \mathcal{N} = 200$$

Key Generation

$$\begin{aligned}
f &= -x^{10} + x^8 + x^7 + x^5 - x^4 - x^2 + 1 \\
g &= x^{10} + x^7 + x^6 - x^5 - x^4 - x^2 + x \\
F &= -3x^{10} - x^8 + x^7 + x^6 - 3x^5 - rs^4 - x^3 - x^2 - 4x - 1 \\
G &= 2x^{10} + x^8 - x^7 + 4x^6 + x^5 + 3x^4 + 3x^2 - x + 4 \\
f_q &= 17x^{10} + 9x^9 + 3x^8 + 8x^7 + 12x^6 + 2x^5 + 4x^4 + 18x^3 + 20x^2 + 26x + 10 \\
g_q &= 15x^{10} + 12x^9 + x_4^8x^7 26x^6 + 25x^5 + 18x^4 + 10x^3 + 21x^2 + 17x + 12 \\
h &= 14x^{10} + 28x^9 + 18x^8 + 8x^7 + 18x^6 + 21x^5 + 9x^4 + 14x^3 + 14x^3 + 24x^2 + 14x + 8
\end{aligned}$$

Signing

$$\begin{aligned}
m &= 160x^{10} + 112x^9 + 32x^8 + 192x^7 + 80x^6 + 128x^5 + 224x^4 + 224x^2 + 144x^2 + 144x + 32 \\
x &= -4x^{10} + 4x^9 - 12x^8 + 2x^7 + 3x^6 - 13x^5 + 7x^4 - 5x^3 + 10x^2 + 9x - 4 \\
s &= 3x^{10} + 2x^9 + 2x^8 - x^7 - x^6 - x^5 + x^3 - 2x^2
\end{aligned}$$

Verification

$$\begin{aligned}
t &= -4x^{10} - 7x^9 - 3x^8 - 5x^7 - 4x^6 - 12x^5 - 11x^4 - x^3 - 8x^2 - 9x \\
v &= 178 \\
178 &\leq 200
\end{aligned}$$

Linear Secret Sharing Scheme

To define linear secret sharing schemes (LSSS), it is first necessary to define some of the access structures used with LSSS. Access structures are used to define the share matrix that is used to share the private key.

Definition 2.20 (Monotone Access Structure)**.** *Let $P = \{P_1, \cdots, P_N\}$ be a set of participants. A collection $\mathbb{A} \subseteq \mathcal{P}(P)$ is a monotone collection if for any sets B, C where $B \in \mathbb{A}$ and $B \subseteq C \subseteq P$ we have $C \in \mathbb{A}$. A monotone access structure on P is a non-empty collection $\mathbb{A} \subseteq \mathcal{P}(P)$. The sets in $\mathbb{A}$ are the valid sets and sets from $\mathcal{P}(P)$ that are not in $\mathbb{A}$ are the invalid sets.*

Example 2.6 (Monotone Access Structure).

Let $P = \{1,2,3,4\}$

Let $\mathbb{A} = \{\{1,2,3\},\{1,2,4\},\{1,3,4\},\{2,3,4\},\{1,2,3,4\}\}$

For any choice of B *all supersets of* $B \in \mathbb{A}$

Example 2.7 (Non-monotone Access Structure).

Let $P = \{1,2,3,4\}$

Let $\mathbb{A} = \{\{1,2\},\{3,4\}\}$

If $B = \{1,2\}$ *the possible values for* C *are* $\{1,2\}$, $\{1,2,3\}$, $\{1,2,4\}$ *and* $\{1,2,3,4\}$. *Only one possible value for* C *is in* $\mathbb{A}$.

A class of monotone access structures is the collection of monotone access structures on P. A threshold access structure can be used to create a t-of-N threshold scheme where only t of the N participants are required to successfully decrypt a ciphertext.

Definition 2.21 (Threshold Access Structure). *Let* $P = \{P_1, \cdots, P_N\}$ *be a set of participants. An access structure* $\mathbb{A}_t$ *is called a threshold access structure if for every set of participants* $S \subseteq P$, $S \in \mathbb{A}_t$ *if and only if* $|S| \geq t$.

Definition 2.22 (Monotone Boolean Formula). *A monotone Boolean formula*

$$C : \{0,1\}^N \to \{0,1\}$$

is a Boolean circuit with the following properties

- *There is a single output gate.*
- *Every gate is one of AND or OR gate with a fan-in 2 and fan-out 1.*
- *The input wires can have multiple fan-out.*

Definition 2.23 (Monotone Boolean Formula Access Structure). *Let* $P = \{P_1, \cdots, P_N\}$ *be the set of participants and* $C : \{0,1\}^N \to \{0,1\}$. *An access structure* $\mathbb{A}_C$ *is a monotone Boolean formula access structure if for every set of participants* $S \subseteq P$, $S \in \mathbb{Q}$ *if and only if* $C(x) = 1$.

A threshold access structure is a subset of the monotone Boolean formula access structure making it possible to write a 2-of-3 threshold as the Boolean formula $(P1 \wedge P2) \vee (P1 \wedge P3) \vee (P2 \wedge P3)$. The "Folklore" algorithm 2.2 can be used to convert a monotone Boolean formula into a share matrix.

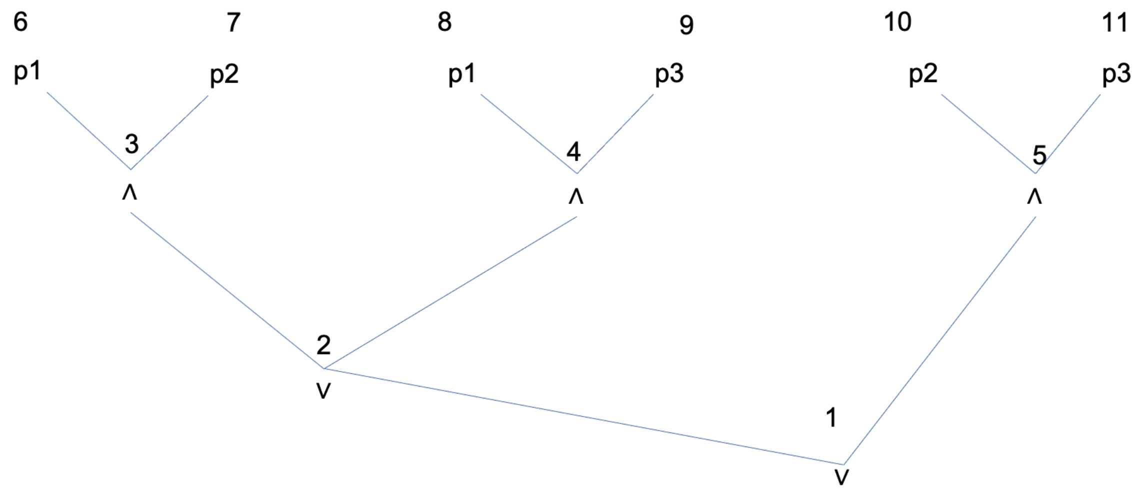
Algorithm 2.2 "Folklore" Algorithm

Input: A special monotone Boolean formula $C : \{0,1\}^N \to \{0,1\}$

Output: An LSSS share matrix M for the access structure C.

```
1. Label the root $r$ with the vector $m_1 = (1)$.
2. Set $counter = 1$
3. for each node $n$ in the tree formed by $C$:
    (a) If $n$ is an OR, assign its children the value of $m$.
    (b) if $n$ is an AND, pad $m_n$ with 0's to make it length $count$. Append a 1 to this value and assign it to one of the children. Assign the other child a vector of $(0, \cdot, -1)$ of length $count + 1$. Increase $count$ by 1.
4. Take the leaf values and pad them with 0's to make them of even size.
```

Example 2.8 (2-of-3 Access Matrix). *Let* $C = (P1 \wedge P2) \vee (P1 \wedge P3) \vee (P2 \wedge P3)$



count	m[count]	left child	right child
1	m[1] = (1)		
1	m[1] = (1)	m[2] = (1)	m[5] = 1
1	m[2] = (1)	m[3] = (1)	m[4] = (1)
1	m[3] = (1)	m[6] = (1,1)	m[7] = (0,-1)
2	m[4] = (1)	m[8] = (1,0,1)	m[9] = (0,0,-1)
3	m[5] = (1)	m[10] = (1,0,0,1)	m[11] = (0,0,0,-1)

The share matrix is $\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix}$

Linear secret sharing schemes (LSSS) [4] use linear algebra and a shared matrix defined by an access structure to take a private key and generate shared keys.

Definition 2.24 (Linear Secret Sharing Scheme). *Let $P = \{P_1, \cdots, P_N\}$ be the set of participants and $\mathbb{S}$ be a class of access structures on P. A secret sharing scheme SS with secret key space $\mathcal{K} = \mathbb{Z}_p$ for some prime p is called a linear secret sharing scheme if:*

- *SS.Share($k, \mathbb{A}$): There exists a share matrix $M \in \mathbb{Z}_p^{\ell x N}$ with each party associated with a partition $T_i \subseteq [\ell]$. To create secret shares of k, sample random values $r_2, \cdots, r_n \xleftarrow{R} \mathbb{Z}_p$ and define a vector $w = M \cdot (k, r_2, \cdots, r_n)^T$ The shares for P_i consists of the entries $\{w_j\}_{j \in T_i}$.*

- *SS.Combine(B): For any valid set $S \in \mathbb{A}$*

  $$(1, 0, \cdots, 0) \in span(\{M[j]\}_{j \in \cup_{i \in S} T_i})$$

  over $\mathbb{Z}_p$ where $M[j]$ is the jth row of M. Any valid set of parties $S \in \mathbb{A}$ can find the coefficients $\{c_j\}_{j \in \cup_{i \in S} T_i}$ satisfying

  $$\sum_{j \in \cup_{i \in S} T_1} c_j \cdot M[j] = (1, 0, \cdots, 0)$$

  and recover the secret by computing $k = \sum_{j \in \cup_{i \in S} T_1} c_j \cdot w_j$. The coefficients $\{c_j\}$ are called the recovery coefficients.

Example 2.9 ({0,1}-LSSS)**.** *Using the share matrix from Example 2.8 and $p = 47$.*

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 47 \\ 63 \\ 2 \\ 15 \end{bmatrix} = \begin{bmatrix} 110 \\ -63 \\ 49 \\ -2 \\ 62 \\ -15 \end{bmatrix}$$

- *P1 gets 110 to use with P2 and 49 to use with P3*
- *P2 gets -63 to use with P1 and 62 to use with P3*
- *P3 gets -2 to use with P1 and -15 to use with P2*

While LSSS works for secret sharing, it does not work well for threshold secret sharing. A partial decryption would leak information about the share used in the decryption. One way to resolve the issue would be to add noise to the decryption (LWE). However the noise adds up quickly and causes decryption failures. A special case of LSSS called {0,1}-LSSS, does work well by using a monotone Boolean formula access structure.

Definition 2.25 ({0,1}-LSSS)**.** *Let $P = \{P_1, \cdots, P_N\}$ be a set of participants. The class of access structure $\{0,1\} - LSSS_N$ is the collection of access structures $\mathbb{A} \in LSSS_N$ for which there exists a linear secret sharing scheme SS = (SS.Share, SS.Combine) over the secret space $K = \mathbb{Z}_p$ satisfying*

- *Let k be a shared secret and $\{w_j\}_{j \in T_I}$ be the share of participant P_i for $i \in [N]$. For every set $S \in \mathbb{A}$ there exists a subset $T \subseteq \cup_{i \in S} T_i$ such that $k = \sum\limits_{j \in T} w_j$.*

Chapter 3

Threshold Lattice-Based Cryptography

Overview

Our scheme relies on a fully trusted dealer. The dealer will generate the key pair based on the chosen NTRU variant and will be the only party to know the private key. There will be N participants who fully trust the dealer, but not necessarily each other. We want t of the N participants to be able to work together to decrypt a ciphertext. By requiring only t of N, not all participants have to be online to perform a decryption and if one is compromised it does not compromise the private key.

The dealer will be responsible for generating the key shares. The N participants will be partitioned into a subset of t participants. This will generate $\frac{N!}{t!(N-t)!}$ partitions. For each partition, a participant will have a key share to use with the other participants in the partition. Each participant will have $\frac{(N-1)!}{(t-1)!(N-t)!}$ shares. The dealer will use a variation of {0,1}-LSSS based on polynomials to generate the key shares.

The participants will need to treat their share as if it were a private key. The loss of a single key would not compromise the private key, but a loss of all the keys for a valid subset would reveal the private key. The participants will each create a partial decryption using their share. The partial decryptions could be used to derive the participant's share, so we will need to mask the partial decryption. The mask will need to hide the share, but at the same time it should not interfere in the final decryption. Once the participants have generated the partial decryptions, they can be combined into the final decryption revealing the plaintext message.

Working code for all three written using Sage can be found in the appendix. This is proof of concept code only and should not be considered production worthy.

Key Sharing

Key sharing will use {0,1}-LSSS, however instead of multiplying the access matrix by a column vector, a matrix is used. The first row of the matrix will be the private key, the remaining rows will be vectors generated using the same rules used to generate the private key. This works because we can change $\mathbb{Z}_p$ to be $\mathbb{Z}_p^m$ in Definition 2.24 and the definition will still hold.

Definition 3.1 (Polynomial Linear Secret Sharing Scheme)**.** *Let $P = \{P_1, \cdots, P_N\}$ be the set of participants and $\mathbb{S}$ be a class of access structures on P. A secret sharing scheme SS with secret key space $\mathcal{K} = \mathbb{Z}_p^m$ for some prime p of degree m is called a polynomial linear secret sharing scheme if:*

- *SS.Share($k, \mathbb{A}$): There exists a share matrix $M \in \mathbb{Z}_p^{\ell x N}$ with each party associated with a partition $T_i \subseteq [\ell]$. To create secret shares of k, sample random polynomials $r_2, \cdot, r_n \overset{R}{\leftarrow} \mathbb{Z}_p^m$ and define a matrix $w = M \cdot (k, r_2, \cdots, r_n)^T$ The shares for P_i consists of the entries $\{w_j\}_{j \in T_i}$.*

- *SS.Combine(B): For any valid set $S \in \mathbb{A}$*

$$(1, 0, \cdots, 0) \in span(\{M[j]\}_{j \in \cup_{i \in S} T_i})$$

  over $\mathbb{Z}_p^m$ where $M[j]$ is the jth row of M. Any valid set of parties $S \in \mathbb{A}$ can find the coefficients $\{c_j\}_{j \in \cup_{i \in S} T_i}$ satisfying

$$\sum_{j \in \cup_{i \in S} T_1} c_j \cdot M[j] = (1, 0, \cdots, 0)$$

  and recover the secret by computing $k = \sum_{j \in \cup_{i \in S} T_1} c_j \cdot w_j$. The coefficients $\{c_j\}$ are called the recovery coefficients.

Making the same change to Definition 2.25 gives

Definition 3.2 (Polynomial {0,1}-LSSS)**.** *Let $P = \{P_1, \cdots, P_N\}$ be a set of participants. The class of access structure $\{0, 1\} - LSSS_N$ is the collection of access structures $\mathbb{A} \in LSSS_N$ for which there exists a linear secret sharing scheme SS = (SS.Share, SS.Combine) over the secret space $K = \mathbb{Z}_p^n$ satisfying*

- *Let k be a shared secret and $\{w_j\}_{j \in T_I}$ be the share of participant P_i for $i \in [N]$. For every set $S \in \mathbb{A}$ there exists a subset $T \subseteq \cup_{i \in S} T_i$ such that $k = \sum_{j \in T} w_j$.*

For Stinson's and NTRU with R-LWE, there is only one key to share. For standard NTRU both f and f_p must be shared, thus the sharing algorithm must be executed twice.

Example 3.1 (Polynomial {0,1}-LSSS). *Using the share matrix form Example 2.8 and the private key from Example 2.3*

$$M = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

$$Key = \begin{bmatrix} 1 & 0 & -3 & 0 & -3 & 0 & 0 & 0 & 3 & -3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 1 & -3 & 0 & 3 & 0 & 0 & 0 & 0 & 3 & -3 & 0 & -3 & 0 & 0 & 0 & 3 & 3 & 0 & 0 & 0 & -3 & 0 & 0 \\ -2 & 0 & 0 & 0 & -3 & 3 & 0 & 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 & 3 & -3 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & -3 & 0 & 3 & 0 & 0 & 0 & 3 & 0 & 3 & -3 & 3 & -3 \end{bmatrix}$$

$$w = \begin{bmatrix} 2 & -3 & -3 & 3 & -3 & 0 & 0 & 0 & 6 & -6 & 0 & -3 & 0 & 0 & 0 & 3 & 3 & 0 & 3 & 0 & -3 & 0 & 0 \\ -1 & 3 & 0 & -3 & 0 & 0 & 0 & 0 & -3 & 3 & 0 & 3 & 0 & 0 & 0 & -3 & -3 & 0 & 0 & 0 & 3 & 0 & 0 \\ -1 & 0 & -3 & 0 & -6 & 3 & 0 & 0 & 3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 3 & 0 & 0 & 3 & -3 \\ 2 & 0 & 0 & 0 & 3 & -3 & 0 & 0 & 0 & -3 & -3 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & -3 & 3 \\ 2 & 0 & -3 & 0 & -3 & 0 & 0 & 0 & 3 & -3 & -3 & -3 & 0 & 3 & 0 & 0 & 0 & 3 & 3 & 3 & -3 & 3 & -3 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 3 & 0 & -3 & 0 & 0 & 0 & -3 & 0 & -3 & 3 & -3 & 3 \end{bmatrix}$$

- *P1 gets row 0 to use with P2 and row 2 to use with P3.*
- *P2 gets row 1 to use with P1 and row 4 to use with P3.*
- *P3 gets row 3 to use with P1 and row 5 to use with P2*

Partial Decryption

Partial decryption for Stinson's and NTRU with R-LWE works the same. Calculate the first step in the decryption algorithm using the share instead of the private key. Let S be a valid set of participants. Let $s_i \in S$ for P_i. The partial decryption of ciphertext y is

$$a_i = s_i * y \pmod{s\, q}$$

For standard NTRU the previous step uses the shares of f. The partial decryption is combined, and the process is repeated with the shares of f_p.

As defined, the partial decryption can leak the key share. Let t be the number of participants in S. Let $A = (a_1, \cdots, a_t)$ be the set of partial decryptions. Given A and y it would be possible for an attacker to compute the shares and thus determine the private key. To avoid this we use R-LWE from Definition 2.14 and add in a small error to each a_i.

Definition 3.3 (Partial Decryption)**.**

- *SNTRU.PartialDecrypt and pNE.PartialDecryption: Let S be a set of the shares of valid participants. Let $s_i \in S$. Let e_i be an independently generated polynomial using a discrete Gaussian distribution with a small standard deviation for the coefficients. For each $s_i \in S$*

$$a_i = s_i * y + p * e_i \ (\mathrm{mods}\, q)$$

- *NTRU.PartialDecryption: Let S_f be the set of shares of f for t valid participants. Let $s_{fi} \in S_f$ where $i \in \{1, \cdots, t\}$ Let e_i be an independently generated polynomial using a discrete Gaussian distribution with a small standard deviation for the coefficients. For each $s_{fi} \in S$*

$$c_i = s_{fi} * y + p * e_i \ (\mathrm{mods}\, q)$$

  Then sum the c_i to get the first partial decryption.

$$b = \sum_{i=1}^{t} c_i \ (\mathrm{mods}\, q)$$

  Let S_{f_p} be the set of shares of f_p for t valid participants. Let $s_{f_p i} \in S_{f_p}$ where $i \in \{1, \cdots, t\}$. Let e_{pi} be an independently generated polynomial using a discrete Gaussian distribution with a small standard deviation for the coefficients. For each $s_{f_p i} in S_p$

$$a_i = s_{f_p i} * b + p * e_{pi} \ (\mathrm{mods}\, q)$$

Example 3.2 (Partial Decryption for SNTRU and pNE)**.** *Using the keys and ciphertext from Example 2.3 and the shares generated in Example 3.1. P_1 and P_2 would like to decrypt the ciphertext y. P_1 calculates*

$$\begin{aligned}
s_1 &= -3x^{20} + 3x^{18} + 3x^{17} + 3x^{16} - 3x^{12} - 3x^{10} - 3x^9 + 6x^8 - 3x^4 + 3x^3 - 3x^2 - 3x + 2 \\
e_1 &= x^{22} + x^{20} - 2x^{19} - x^{18} + x^{17} - x^{15} + x^{14} - x^{13} + 2x^{11} - x^{10} + x^8 - 2x^6 - x^5 - x^4 \\
&\quad + x - 1 \\
a_1 &= s_1 * y + p * e_1 \pmod{s\, q} \\
a_1 &= -2x^{22} - x^{21} + 5x^{20} - 6x^{19} + 8x^{18} + 15x^{17} + 12x^{16} + 7x^{15} - 8x^{14} + 15x^{13} - 11x^{12} \\
&\quad - 5x^{11} - 13x^{10} + 3x^9 - 2x^8 + 4x^7 + 10x^6 - 10x^5 - 15x^4 - 13x^3 - 9x^2 + x + 15
\end{aligned}$$

P_2 calculates

$$\begin{aligned}
s_2 &= 3x^{20} - 3x^{17} - 3x^{16} + 3x^{12} + 3x^{10} - 3x^8 - 3x^3 + 3x - 1 \\
e_2 &= -x^{21} - 2x^{20} + x^{18} + 2x^{17} - x^{16} - x^{15} - 2x^{12} - x^9 + 2x^8 + 2x^4 + x^3 - x + 1 \\
a_2 &= s_2 * y + p * e_2 \pmod{s\, q} \\
a_2 &= 11x^{22} + x^{21} - 14x^{20} - 3x^{19} - 8x^{18} - 9x^{17} - 15x^{16} - 6x^{15} + 11x^{14} - 12x^{13} + 4x^{12} \\
&\quad + 2x^{11} + 13x^{10} - 3x^9 + 11x^8 - 9x^7 + 15x^6 + 7x^5 - 7x^4 - 12x^3 + 15x^2 - 4x - 10
\end{aligned}$$

Example 3.3 (Partial Decryption for Standard NTRU)**.** *Using the keys and ciphertext from Example 2.2 P_1 and P_2 would like to decrypt the ciphertext y. P_1 calculates*

$$\begin{aligned}
s_{f1} &= -x^{10} + 2 * x^9 - 2 * x^4 + x^2 + 2 * x - 1 \\
e_1 &= x^{10} - x^8 + x^7 - x \\
c_1 &= s_{f1} * y + p * e_1 \pmod{s\, q} \\
c_1 &= x^{10} - 4x^9 - 12x^8 + 11x^7 + 7x^6 - 2x^5 + 12x^4 - 13x^3 + 2x^2 + 5x - 7
\end{aligned}$$

P_2 calculates

$$\begin{aligned}
s_{f2} &= -x^9 + x^6 + x^4 - x \\
e_2 &= -x^8 + x^7 + x^4 - 1 \\
c_2 &= s_{f2} * y + p * e_2 \pmod{s\, q} \\
c_2 &= -5x^{10} + x^9 + 11x^8 + 2x^7 - x^6 + 9x^5 + x^4 + 2x^3 - 12x^2 - 15x + 7
\end{aligned}$$

P_1 and P_2 calculate

$$b = c_1 + c_2 = -4x^{10} - 3x^9 - x^8 + 13x^7 + 6x^6 + 7x^5 + 13x^4 - 11x^3 - 10x^2 - 10x$$

Next P_1 calculates

$$s_{f_p1} = -x^{10} + 2x^9 + x^8 + 2x^7 + x^5 + 2x^4 + 3x^3 - x^2 + 2x + 2$$
$$e_{p1} = -x^{10} + x^9 - x^7 + x$$
$$a_1 = -x^{10} + x^9 - x^8 - x^7 - x^6 - x^5 - x^4 - x^3 + x - 1$$

P_2 calcuates

$$s_{f_p2} = x^{10} - x^3 + x^2 - 1$$
$$e_{p2} = -x^{10} + x^8 - x^7 + 1$$
$$a_2 = -x^{10} + x^7 + x^6 + x^5 - x^3 - x$$

Final Decryption

Final decryption for all three is the sum of the partial decryptions mods q and mods p.

Definition 3.4 (Final Decryption)**.**

- *FinalDecrypt: Let S be a set of valid participants. Let t be the number of participants in S. Let $A = (a_1, \cdots, a_t)$ be the set of partial decryptions. The final decryption of the ciphertext y is*

$$(\sum_{i=1}^{t} a_i \ (\operatorname{mods} q)) \ (\operatorname{mods} p)$$

Example 3.4 (Final Decryption)**.** *Using the partial decryptions from Example 3.2:*

$$\begin{aligned} a_1 + a_2 = 9x^{22} &+ 9x^{21} - 15x^{20} + 3x^{19} + 25x^{17} - 3x^{16} + 7x^{15} \\ &- 25x^{14} + 25x^{13} - 4x^{12} + 16x^{11} - 25x^{10} + 3x^9 + x^7 + 25x^6 \\ &+ 3x^5 + 3x^4 - 28x^3 + 3x^2 - 6x - 1 \end{aligned}$$

$$(a_1 + a_2 \ (\operatorname{mods} q)) \ (\operatorname{mods} p) = x^{15} - x^{12} + x^7 - 1$$

Proof. Proof of Correctness for Stinson's and NTRU With R-LWE. Let S be a set of valid participants. Let $s_i \in S$. Let e_i be small independent polynomials. Final decryption is

$$(\sum_{i}^{t} a_i \ (\operatorname{mods} q)) \ (\operatorname{mods} p)$$

a_i is defined as

$$a_i = s_i * y + p * e_i \ (\operatorname{mods} q)$$

Substituting a_i into the previous equation gives

$$(\sum_{i}^{t} s_i * y + p * e_i \pmod{s\,q} \pmod{s\,q}) \pmod{s\,p}$$

Rearranging the terms gives

$$(\sum_{i}^{t} s_i * y \ (\mathrm{mods}\, q)\ (\mathrm{mods}\, q))\ (\mathrm{mods}\, p) + (\sum_{i}^{t} p * e_i \ (\mathrm{mods}\, q)\ (\mathrm{mods}\, q))\ (\mathrm{mods}\, p)$$

Which reduces to

$$(\sum_{i}^{t} s_i * y \ (\mathrm{mods}\, q))\ (\mathrm{mods}\, p)$$

Which can be rewritten as

$$(\sum_{i}^{t} s_i) * y \ (\mathrm{mods}\, q)\ (\mathrm{mods}\, p)$$

Recall from Definition 3.2 the sum of the shares is the private key giving

$$f * y \ (\mathrm{mods}\, q)\ (\mathrm{mods}\, p)$$

Which is the original decryption formula. □

The proof for standard NTRU is similar. In the previous proof, replace S with S_f, the result is the formula for the first step of standard NTRU decryption. Repeat the proof using S_{f_p} for S and a for y. The result of this proof is the second formula from standard NTRU.

Chapter 4

Security and Related Work

Related Work

A 2018 paper by Andrew Xia [35] covers a similar topic. That paper uses multiparty computation to create a dealer free threshold multi-key fully homomorphic lattice-based encryption scheme. The multiparty computation allows the participants to negotiate the shares without a dealer and without revealing the secret keys. Fully homomorphic means addition and multiplication can be performed without having to decrypt the message. In this paper we focus on single key threshold lattice-based encryption with a dealer. We also provide examples and working prototypes to validate the functionality of the scheme.

Key Sharing Security

One avenue of attack with secret sharing is to recover the key from one or more shares from a partition. Could a possessor of a share use that share to discover the private key? The way the shares are generated, they are random. The possessor of a share will not be able to reconstruct the private key.

Under certain conditions, the primary share for standard and Stinson's NTRU can leak information about the private key in certain cases. For each exponent, the primary share coefficient is the sum of the matching coefficients from the private key and the random polynomials. Let N be the degree of the polynomials. Let $i \in [0, N-1]$. Let t be the number of polynomials in the partition. Let t_0 be the primary key and the remaining the random polynomials. The coefficient i of the primary share is calculated by

$$\sum_{j=0}^{t} c_{ji}$$

If the coefficients are all the same value, the sum divided by t will reveal the coefficient of the primary key. For example, if the coefficients are all 3 and there are 5 polynomials in the partition, the sum will be 15 and $15/5 = 3$. To avoid the leakage, reject any combination of private key and random polynomials where all coefficients are the same value for the respective exponent.

The R-LWE variant does not have this problem. The coefficient range of the private key for the R-LWE version is much larger. There are no values that would stand out as unusual. Although 99.7% of the coefficients will be within 3 standard deviations fo the mean, there is a 0.3% chance a coefficient will be abnormally large.

With this condition in place, the primary share does not leak any information about the private key. The random polynomials added to the private key obfuscate enough to make extracting the private key from the share computationally expensive. The lowest recommended value for N is 439, the number of combinations to test would be 3^{439}. It would be higher for NTRU with R-LWE since the coefficients are not limited to three values.

Partial Decryption Security

Given a partial decryption, could an attacker recover the key share? If partial decryption was unmodified and an attacker had both the ciphertext and partial decryption, it would be possible, but difficult, to derive the share used. The security of the partial decryption can be enhanced by adding a small error.

The security of a partial decryption for NTRU with R-LWE can be derived directly from R-LWE 2.14 with slight changes. Let Φx be $x^N + 1$ an irreducible polynomial. The finite quotient is $\mathbb{Z}_q/(x^N + 1)$. Instead of s being a secret it is the ciphertext and $a_i(x)$ are the shared keys. We have redefined what needs to be kept secret. The values for $e_i(x)$ are small independent random unknown polynomials with the coefficients sampled from the R-LWE error distribution. Having met the requirements for R-LWE security we can say the partial decryptions are provably secure.

The same security argument cannot be applied to standard nor Stinson's NTRU. The value of $\Phi(x)$ is $x^N - 1$ which is reducible. The rest of the R-LWE requirements hold, the values for e_i are still small random independent unknown polynomials with the coefficients sampled form the R-LWE error distribution. This alters the result just enough to make deriving the key share from the partial decryption sufficiently complex but does not interfere with final decryption. Although we cannot claim R-LWE level security, we can still claim it is secure.

Final Decryption Security

With the key shares properly masked, there are no secrets to protect in the final decryption. Combining the partial decryptions to get the plaintext is a safe operation. The only thing an attacker can get is the plaintext, which would be the output of standard decryption as well.

Chapter 5

Summary

Threshold cryptography provides a secure way to keep a secret online long term. A user, Bob, can generate a public/private key. After choosing how many servers, N, and a threshold, t, Bob will create shares of the private key to distribute to the N servers. The servers can work together to decrypt ciphertext for Bob. The t servers will generate partial decryptions that can be used to recover the plaintext. The partial decryptions are generated in a way to avoid leaking the key share. Should a key share be compromised it will not reveal Bob's private key. If less than t shares are compromised, Bob can regenerate and distribute new shares to the servers. Bob's private key is still secure and the compromised share is no longer a threat.

In this paper, we explained how to create a threshold cryptography scheme for three variants of the lattice-based encryption scheme NTRU. We defined a version of {0,1}-LSSS that works with polynomials to share a private key. The decryption algorithms of standard, Stinson's, and R-LWE NTRU were modified to include partial and final decryption steps. Throughout chapter 3 we included concrete examples of the different steps. Finally, we included a proof of correctness for the equations involved in the partial and final decryptions.

In chapter 4, we discussed the security of key sharing and partial decryption. When correctly constructed, the shares do not leak information about the private key. The only way to reconstruct the private key is to acquire t of N shares from the same partition, excluding a brute force attack. The shares are used to securely partially decrypt the ciphertext. We do not claim provable security in all cases, but we can state there is no loss of security by partial decryption.

Appendix A

Threshold Standard NTRU

```
from sage.stats.distributions.discrete_gaussian_polynomial \
    import DiscreteGaussianDistributionPolynomialSampler

#start https://github.com/kpatsakis/NTRU_Sage/blob/master/ntru.sage
# with bug fixes
R.<x> = ZZ['x']

def mods(f, m):
    coeffs = f.list()
    m2 = m/2
    for i in range(len(coeffs)):
        coeffs[i] = coeffs[i] % m
        if coeffs[i] > m2:
            coeffs[i] -= m
    return R(coeffs)

def mod(f, m):
    return R([ i % m for i in f.list()])

def poly_mod_2(poly):
    k = 0; b = 1; c = 0*x
    f = poly; g = x^N - 1
    res = False

    f = mods(f, 2)
    while True:
        while f(0) == 0 and not f.is_zero():
            f = f.shift(-1)
            c = c.shift(1)
            c = mods(c, 2)
            k += 1
        if f.is_one():
            e = (-k) % N
            retval = x^e * b
            res = True
            break
        elif f.degree() == -1 or f.is_zero():
            break
        if f.degree() < g.degree():
            f,g = g,f
            b,c = c,b
        f += g
        b += c
        f = mods(f, 2)
        c = mods(c, 2)
    if res:
        retval = retval % (x^N - 1)
        retval = mods(retval, 2)
        return True, retval
    else:
        return False, 0

def poly_mod_prime_pow(poly):
    success, b = poly_mod_2(poly)
    if success:
        qr = 2
        while qr < q:
            qr = qr^2
            b = b * (2 - poly * b)
            b = b % (x^N - 1)
            b = mod(b,q)
        return True, b
    else:
        return False, 0

def poly_mod_3(poly):
    k = 0; b = 1; c = 0*x
    f = poly; g = x^N - 1
    res = false

    while True:
        while f(0) == 0 and not f.is_zero():
            f = f.shift(-1)
            c = c.shift(1)
            k+=1
        if f.is_one():
            e = (-k) % N
            retval = x^e * b
            res = True
            break
        elif (-f).is_one():
            e = (-k) % N
            retval = -x^e * b
            res = True
            break
        elif f.degree() == 1 or f.is_zero():
            break
        if f.degree() < g.degree():
            f,g = g,f
            b,c = c,b
        if f(0) == g(0):
            f -= g
            b -= c
        else:
            f += g
            b += c
        f = mods(f, 3)
        c = mods(c, 3)
    if res:
        retval = retval % (x^N - 1)
        retval = mod(retval, 3)
        return True, retval
    else:
        return False, 0

#end https://github.com/kpatsakis/NTRU_Sage/blob/master/ntru.sage


def sample():
    s=[1]*n1+[-1]*n1+[0]*n0
    shuffle(s)
    return R(s)


# parameters
N = 439
p = 3
q = 2048
n1 = (N/6).round()
n0 = N - 2*n1
#generating an invertable f is not as easy as the other two
#using a constant key known to work.
f = -1 + x + x^2 - x^4 + x^6 + x^9 - x^10
g = -1 + x^2 + x^3 + x^5 - x^8 - x^10
```

```
# Compute h(x)
success, fp = poly_mod_3(f)
if not success:
    print("Something went wrong")
success, fq = poly_mod_prime_pow(f)
if not success:
    print("Something went wrong")
h = mods((p * fq * g) % (x^N - 1), q)

# encrypt
m = -1 + x^3 - x^4 - x^8 + x^9 + x^10
r = sample() #-1 + x^2 + x^3 + x^4 - x^5 - x^7
e = mod((r*h+m) % (x^N - 1), q)

fKeys = [f, sample(), sample(), sample()]

v = 4*[[]]
for i in range(len(fKeys)):
    v[i] = fKeys[i].list()
    v[i] += (N - len(v[i]))*[0]

alpha = sqrt((2*N)/pi)
Da = DiscreteGaussianDistributionPolynomialSampler(R, N, alpha)
fk = matrix(v)
am = matrix([[1,1,0,0],[0,-1,0,0],[1,0,1,0],[0,0,-1,0],[1,0,0,1],[0,0,0,-1]])
fsm = am*fk
print("f shares")
pretty_print(fsm)
fShares = [R(s.list()) for s in fsm]
partial = [mods((s*e + p*Da())%(x^N - 1), q) for s in fShares]

print("b with shares")
print("p12 + p21:", mods(partial[0]+partial[1],q))
print("p13 + p31:", mods(partial[2]+partial[3],q))
print("p23 + p32:", mods(partial[4]+partial[5],q))

partial[0] = partial[1] = mods(partial[0]+partial[1],q)
partial[2] = partial[3] = mods(partial[2]+partial[3],q)
partial[4] = partial[5] = mods(partial[4]+partial[5],q)

fpKeys = [fp, sample(), sample(), sample()]

v = 4*[[]]
for i in range(len(fpKeys)):
    v[i] = fpKeys[i].list()
    v[i] += (N - len(v[i]))*[0]

fpk = matrix(v)
fpsm = am*fpk
print("fp shares")
pretty_print(fpsm)
fpShares = [R(s.list()) for s in fpsm]
for i in range(len(partial)):
    partial[i] = mod(partial[i],p)
    partial[i] = mods((fpShares[i] * partial[i] + p*Da()) % (x^N - 1),p)

print("fp = ", fp)
print("fq = ", fq)
print("h = ", h)
print("m = ", m)
print("r = ", r)
print("e = ", e)

print("m‘ with shares:")
print("p12 + p21:", mods(mods(partial[0]+partial[1],q), p))
print("p13 + p31:", mods(mods(partial[2]+partial[3],q), p))
print("p23 + p32:", mods(mods(partial[4]+partial[5],q), p))
```

Appendix B

Threshold Stinson NTRU

```
from sage.stats.distributions.discrete_gaussian_polynomial \
    import DiscreteGaussianDistributionPolynomialSampler

#start https://github.com/kpatsakis/NTRU_Sage/blob/master/ntru.sage
# with bug fixes
R.<x> = ZZ['x']

def mods(f, m):
    coeffs = f.list()
    m2 = m/2
    for i in range(len(coeffs)):
        coeffs[i] = coeffs[i] % m
        if coeffs[i] > m2:
            coeffs[i] -= m
    return R(coeffs)

def mod(f, m):
    return R([ i % m for i in f.list()])

def poly_mod_2(poly):
    k = 0; b = 1; c = 0*x
    f = poly; g = x^N - 1
    res = False

    f = mods(f, 2)
    while True:
        while f(0) == 0 and not f.is_zero():
            f = f.shift(-1)
            c = c.shift(1)
            c = mods(c, 2)
            k += 1
        if f.is_one():
            e = (-k) % N
            retval = x^e * b
            res = True
            break
        elif f.degree() == -1 or f.is_zero():
            break
        if f.degree() < g.degree():
            f,g = g,f
            b,c = c,b
        f += g
        b += c
        f = mods(f, 2)
        c = mods(c, 2)
    if res:
        retval = retval % (x^N - 1)
        retval = mods(retval, 2)
        return True, retval
    else:
        return False, 0

def poly_mod_prime_pow(poly):
    success, b = poly_mod_2(poly)
    if success:
        qr = 2
        while qr < q:
            qr = qr^2
            b = b * (2 - poly * b)
            b = b % (x^N - 1)
            b = mod(b,q)
        return True, b
    else:
        return False, 0

def poly_mod_3(poly):
    k = 0; b = 1; c = 0*x
    f = poly; g = x^N - 1
    res = false

    while True:
        while f(0) == 0 and not f.is_zero():
            f = f.shift(-1)
            c = c.shift(1)
            k+=1
        if f.is_one():
            e = (-k) % N
            retval = x^e * b
            res = True
            break
        elif (-f).is_one():
            e = (-k) % N
            retval = -x^e * b
            res = True
            break
        elif f.degree() == 1 or f.is_zero():
            break
        if f.degree() < g.degree():
            f,g = g,f
            b,c = c,b
        if f(0) == g(0):
            f -= g
            b -= c
        else:
            f += g
            b += c
        f = mods(f, 3)
        c = mods(c, 3)
    if res:
        retval = retval % (x^N - 1)
        retval = mod(retval, 3)
        return True, retval
    else:
        return False, 0

#end https://github.com/kpatsakis/NTRU_Sage/blob/master/ntru.sage

def sample():
    s=[1]*n1+[-1]*n1+[0]*n0
    shuffle(s)
    return R(s)

# parameters
N = 439
p = 3
q = 2048
n1 = (N/6).round()
n0 = N - 2*n1
alpha = sqrt((2*N)/pi)
Da = DiscreteGaussianDistributionPolynomialSampler(R, N, alpha)

F = sample()
G = sample()
```

```
f = p*F + 1
g = p*G

# Compute h(x)
success, fq = poly_mod_prime_pow(f)
if not success:
    print("Something went wrong")
h = mods((fq * g) % (x^N - 1), q)

# encrypt
m = -1 + x^3 - x^4 - x^8 + x^9 + x^10
r = sample()
e = mod((r*h+m) % (x^N - 1), q)

fKeys = [f, p*sample() + 1,p*sample() + 1,p*sample() + 1]

v = 4*[[]]
for i in range(len(fKeys)):
    v[i] = fKeys[i].list()
    v[i] += (N - len(v[i]))*[0]

fk = matrix(v)
print("fKeys")
pretty_print(fk)
am = matrix([[1,1,0,0],[0,-1,0,0],[1,0,1,0],[0,0,-1,0],[1,0,0,1],[0,0,0,-1]])
print("access matrix")
pretty_print(am)
fsm = am*fk
print("f shares")
pretty_print(fsm)
fShares = [R(s.list()) for s in fsm]
partial = [mods((s*e + p*Da())%(x^N - 1), q) for s in fShares]
```

```
print("fq = ", fq)
print("h = ", h)
print("m = ", m)
print("r = ", r)
print("e = ", e)

print("m‘ with shares:")
print("p12 + p21:", mods(mods(partial[0]+partial[1],q), p))
print("p13 + p31:", mods(mods(partial[2]+partial[3],q), p))
print("p23 + p32:", mods(mods(partial[4]+partial[5],q), p))
```

Appendix C

Threshold NTRU With R-LWE

```
from sage.stats.distributions.discrete_gaussian_polynomial \
    import DiscreteGaussianDistributionPolynomialSampler

R.<x> = ZZ['x']

N = 256
p = 2
q = Primes().next(ceil(512*(N^6)*ln(N)))
sigma = 2 * N * sqrt(ln(8*N*q)*q)
alpha = sqrt((2*N)/pi)

def mods(f, m):
    coeffs = f.list()
    m2 = int(m/2)
    for i in range(len(coeffs)):
        coeffs[i] = coeffs[i] % m
        if coeffs[i] > m2:
            coeffs[i] -= m
    return R(coeffs)

def inverse_mods(f1, q):
    Z = GF(q)
    R2 = PolynomialRing(Z,'a'); a = R2.gen()
    S = R2.quotient(a^N + 1, 'x'); x = S.gen()
    newF = S(f1.list())
    resultZ = R((newF^-1).list())
    return mods(resultZ, q)
```

```
Ds = DiscreteGaussianDistributionPolynomialSampler(R, N, sigma)
f = p * Ds() + 1
fq = inverse_mods(f, q)
g = Ds()
gq = inverse_mods(f, q)
h = mods((fq * p * g) % (x^N + 1), q)
Da = DiscreteGaussianDistributionPolynomialSampler(R, N, alpha)
r = Da()
e = Da()
m = x^15 + x^12 + x^7 + 1
y = mods((r*h+p*e+m) % (x^N + 1), q)

#compute shares
keys = [f, p*Ds()+1, p*Ds()+1, p*Ds()+1]

v = 4*[[]]
for i in range(len(keys)):
    v[i] = keys[i].list()
    v[i] += (N - len(v[i]))*[0]

k = matrix(v)
am = matrix([[1,1,0,0],[0,-1,0,0],[1,0,1,0],[0,0,-1,0],[1,0,0,1],[0,0,0,-1]])
sm = am*k
shares = [R(s.list()) for s in sm]

partial = []
for i in range(len(shares)):
    partial.append(mods((shares[i]*y + p*Da())%(x^N + 1), q))

print("N:", N)
print("q:", q)
print("p:", p)
print("sigma: ", sigma.n().str(no_sci=2))
print("alpha:", alpha.n())
print("f: ", f)
print("fq: ", fq)
print("g: ", g)
print("h: ", h)
print("")
print("r: ", r)
print("e: ", e)
print("m: ", m)
print("y: ", y)
print("")

print("shares")
pretty_print(sm)

print("m‘ with shares:")
print("p12 + p21:", mods(mods(partial[0]+partial[1],q), p))
print("p13 + p31:", mods(mods(partial[2]+partial[3],q), p))
print("p23 + p32:", mods(mods(partial[4]+partial[5],q), p))
```

Bibliography

[1] IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices. *IEEE Std 1363.1-2008*, 2008.

[2] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 99–108, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917855. doi: 10.1145/237814.237838.

[3] G. R. Blakley. Safeguarding cryptographic keys. In *Proc. Nat. Computer Conf. AFIPS Conf. Proc*, pages 313–317, 1979.

[4] D. Boneh, R. Gennaro, S. Goldfeder, A. Jain, S. Kim, P. Rasmussen, and A. Sahai. *Threshold Cryptosystems from Threshold Fully Homomorphic Encryption: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I*, pages 565–596. 01 2018. ISBN 978-3-319-96883-4. doi: 10.1007/978-3-319-96884-1_19.

[5] D. Cabarcas, P. Weiden, and J. Buchmann. On the efficiency of provably secure ntru. In M. Mosca, editor, *Post-Quantum Cryptography*, pages 22–39, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11659-4.

[6] D. Coppersmith and A. Shamir. Lattice attacks on ntru. In *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'97, page 52–61, Berlin, Heidelberg, 1997. Springer-Verlag. ISBN 3540629750.

[7] R. A. Croft and S. P. Harris. Public-key cryptography and re-usable shared secrets. In H. Beker and F. Piper, editors, *Cryptography and coding*, pages 189–201. Clarendon Press, Royal Agricultural College, Cirencester, December 15–17 1989.

[8] Y. Desmedt. Society and group oriented cryptography: a new concept. In *Advances in Cryptology, Proc. of Crypto '87*, pages 120–127, August 16–20 1988.

[9] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology — Crypto '89*, pages 307–315, Berlin, August 20–24 1990.

[10] L. Ducas and P. Nguyen. Learning a zonotope and more: Cryptanalysis of ntrusign countermeasures. pages 433–450, 12 2012. ISBN 978-3-642-34960-7. doi: 10.1007/978-3-642-34961-4_27.

[11] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory*, 31:469–472, 1985.

[12] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and efficient sharing of RSA functions. In *Advances in Cryptology — Crypto '96*, pages 157–172, August 18–22 1996.

[13] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In *Advances in Cryptology — Eurocrypt '96*, pages 354–371, May 12–16 1996.

[14] R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk. Robust and efficient sharing of RSA functions. *Journal of Cryptology*, 13(2):273–300, 2000.

[15] J. Hoffstein, J. Pipher, and J. Silverman. Ntru: a new high speed public key cryptosystem. Crypto 1996.

[16] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. Silverman, and W. Whyte. Ntrusign: Digital signatures using the ntru lattice. 11 2003.

[17] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. Silverman, and W. Whyte. Ntrusign: Digital signatures using the ntru lattice [superseded]. 11 2003.

[18] J. Hoffstein, N. Howgrave-Graham, J. Pipher, and W. Whyte. *Practical Lattice-Based Cryptography: NTRUEncrypt and NTRUSign*, chapter 11, pages 349–390. Springer, 2010.

[19] J. Hoffstein, J. Pipher, J. Schanck, J. Silverman, W. Whyte, and Z. Zhang. Choosing parameters for ntruencrypt. pages 3–18, 02 2017. ISBN 978-3-319-52152-7. doi: 10.1007/978-3-319-52153-4_1.

[20] N. Howgrave-Graham, P. Q. Nguyen, D. Pointcheval, J. Proos, J. H. Silverman, A. Singer, and W. Whyte. The impact of decryption failures on the security of ntru

encryption. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 226–246, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45146-4.

[21] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[22] A. Lenstra, H. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261, 12 1982. doi: 10.1007/BF01457454.

[23] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. volume 60, pages 1–23, 05 2010. ISBN 978-3-642-13189-9. doi: 10.1007/978-3-642-13190-5_1.

[24] V. S. Miller. Use of elliptic curves in cryptography. In *Advances in Crypto — CRYPTO'85*, volume 218, pages 417–426, 1986.

[25] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *QIC*, 3(4):317–344, 2003. http://arxiv.org/abs/quantph/0301141.

[26] O. Regev. Lattice-based cryptography. In *Proceedings of CRYPTO 2006*, pages 131–141, 2006.

[27] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. volume 56, pages Article 34, 34:1–34:40, September 2009.

[28] O. Regev. The learning with errors problem. In *2010 25th Annual IEEE Conference on Computational Complexity*, 2010.

[29] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signature and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[30] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[31] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. URL http://arxiv.org/abs/quant-ph/9508027.

[32] V. Shoup. Practical threshold signatures. In *Advance in Cryptology – EUROCRYPT 2000*, pages 207–220, May 2000.

[33] D. Stehlé and R. Steinfeld. Making ntru as secure as worst-case problems over ideal lattices. In K. Paterson, editor, *Eurocrypt 2011*, volume 6632, pages 27–47, 2011.

[34] D. Stinson and M. Paterson. *Cryptography: Theory and Practice,Fourth Edition*. CRC/C&H, 4th edition, 2019. ISBN 9781138197015.

[35] A. Xia. *Thresholdizing lattice based encryption schemes*. PhD thesis, 01 2018.