TODO

Marcus Sellebjerg, 201808635
Kasper Ilsøe, 201808327

Master's Thesis (30 ECTS) in Computer Science
Advisor: Peter Scholl
Department of Computer Science, Aarhus University
June 15th, 2023

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Marcus Sellebjerg, Kasper Ilsøe
Aarhus, May 31th, 2023.

Contents

Chapter 1

Introduction

With the emergence of increasingly powerful quantum computers, the question of constructing public-key encryption (PKE) schemes that are resistant to quantum computers, becomes more and more relevant. The field concerning itself with PKE schemes that are resistant to both quantum and classical computers, is known as Post-Quantum Cryptography (PQC). A standardization process initiated by NIST for PQC has also concluded recently, which further highlights the importance of this field of research.

A tool for building such schemes is the learning with errors (LWE) assumption. This assumption allows achieving desirable cryptographic properties, such as plausible resistance to quantum computers, or support for homomorphic computations.

Another important cryptographic problem is that of threshold decryption, in which a secret key is split into shares, such that several shares are needed to be able to decrypt. In particular, for the *t-out-of-n* setting t shares are required to be able to decrypt.

For this thesis we want to look at a (fully homomorphic) encryption scheme based on a variant of the LWE assumption, allowing *t-out-of-n* threshold decryption. The particular encryption scheme that we will look at in this project, is one instantiated using the lattice-based Kyber encryption scheme, which is one of the algorithms chosen in the NIST PQC standardization process.

COPIED FROM THESIS CONTRACT, REPLACE LATER

Chapter 2

Preliminaries

2.1 Secret sharing schemes

When sharing a polynomial we have a few different ways of doing those.

2.1.1 Additive secret sharing

In additivce secret sharing the key will be split into shares, such that each player can have their own share of the key. Imagine an example where 3 players will share a polynomial p. Player 1 and 2 will then choose random polynomials p1 and p2 from the ring and end those to a dealer, such that the dealer can calculate the last persons share

$$p3 = p - p1 - p2$$

each player will now have a share of the polynomial for which it holds that

$$p = p1 + p2 + p3$$

and as such they all need to be present to construct the original polynomial. Note that we could further represent this as a matrix, in which the players are indexed on the rows and the shares are in the coloumns.

$$\begin{pmatrix} p1 \\ p2 \\ p3 \end{pmatrix}$$

and running a recombine function would take every entry in the matrix and add all those shares together.

2.1.2 Replicated secret sharing

In Replicated secret sharing, each player will hold an amount of keys of the secret polynomial, such that if t + 1 players meet, they are able to reconstruct the polynomial p. For simplicity we will only go over the case in which we are sharing 1 polynomial, but do mind that more polynomials can be shared by repeating this and having more matrices instead.

Given n amount of players in which there is t corruptions, we can start by generating all sets of length t of those players and giving each player a share of the key in the subset they are not present.

The case for n = 4 and t = 2, would equal us having 6 subset of players

$$\{1, 2\}\ \{1, 3\}\ \{1, 4\}\ \{2, 3\}\ \{2, 4\}\ \{3, 4\}$$

We will for each of those subsets have an associated key $p1 \ldots p6$ such that reassembling these keys gives back the secret polynomial p.

$$p = p_1 + p_2 + p_3 + p_4 + p_5 + p_6$$

Then each key is given to a player if they are not part of the subset corresponding to the key. For the case mentioned this could be represented by the following matrix

$$\begin{pmatrix} 0 & 0 & 0 & p_4 & p_5 & p_6 \\ 0 & p_2 & p_3 & 0 & 0 & p_6 \\ p_1 & 0 & p_3 & 0 & p_5 & 0 \\ p_1 & p_2 & 0 & p_4 & 0 & 0 \end{pmatrix}$$

Where each row is a player and coloumn the share being held.

To recombine $t+1$ players meet and take an apropriate liniar combination of the shares, such that they end up with $p = p_1 + \cdots + p_6$ and there by the secret polynomial p. For this we have only considered sharing one polynomial, but in the context of kyber we will have to share up to k polynomials. Luckily this can be done rather easily by applying this technique to each polynomial in the vectors, in which case the k polynomials will result in k matrices, as given above, where the rows are players and the columns the share they hold.

Note that $t+1$ players need to be in the scheme to have all the keys $p_1 \dots p_6$, If only t players meet, they would still miss a random share that has been chosen at random in the ring, thus being very hard to find.

2.1.3 Naive secret sharing

Naive secret sharing is the dual of Replicated, in which we give the shares to all players in the subset. The representing matrix for this scheme, using the same $n = 4, t = 2$ setting as before would result in the matrix

$$\begin{pmatrix} p_1 & p_2 & p_3 & 0 & 0 & 0 \\ p_1 & 0 & 0 & p_4 & p_5 & 0 \\ 0 & p_2 & 0 & p_4 & 0 & p_6 \\ 0 & 0 & p_3 & 0 & p_5 & p_6 \end{pmatrix}$$

where players are on the rows and columns are their shares.

Recombination also works exactly as in the case of replicated, where we take some linear combination such that we end up with all shares and

$$p = p_1 + \cdots + p_6$$

2.1.4 On the robustness of those schemes

hejsa

- We should add which ring we work in.
- We should have more information on the specific bounds.
- Maybe we can draw this a bit better?
- What are the expected runtimes in the different secret sharing schemes
- What are the expected amount of memory to be held in the different secret sharing schemes.
- There is a really nice figure of how those works in the article from Peter and Katharina
- There should be something about how partial decryption works in the case of these different secret sharing schemes, but I don't think this is the place to do it.

2.2 Notes on the different types of LWE

Here I'm introducing the LWE scheme, extending the idea into the RLWE scheme and lastly I'll use the MLWE scheme. MLWE is the one used in Kyber, but the other provides a very nice insight in how one will reduce the problems stated in MLWE to $n * K$ LWE problems, which is the most effective attack against Kyber.

2.2.1 LWE

The following is the LWE scheme

- Pick a vector $s \in \mathbb{Z}_q^n$ at random
- Pick a matrix $a \in \mathbb{Z}_q^{nxm}$ using the uniform distribution
- pick errors as the vector $e \in \mathbb{Z}_q^n$ using some distribution χ
- compute $b_i = \langle a_i, s_i \rangle / q + e_i$ for $i = 1 \ldots m$
- output (a, b_i)

Doing encryption of a bit $x \in \{0, 1\}$ is done by

- Choose subset S of vectors in a
- compute $(\sum_{i \in S} a_i, \frac{x}{2} + \sum_{i \in S})$

Doing decryption of (a, b) is then defined by

- compute $x' = b - \langle a, s \rangle / q$
- round x' to the the nearest of 0 and 1.

Search

The search variant of the LWE problem is defined as the problem of finding the secret $s \in \mathbb{Z}_q^n$ given access to a polynomial amount of public keys in the form (a_i, b_i)

Decision

The decision variant is checking iff a real key has been used in the inner product.

2.2.2 RLWE

The Ring learning with error problem utilises the same ideas the LWE scheme above. With the one exception that instead of working on numbers we are working in the polynomial ring $Z_q[X]/(f(x))$ for some polynomial f(x). The keygen, encryption and decryption are the same as above, but instead uses polynomials.

search

The search problem is also the same as before, but instead we are searching for the polynomial $s \in Z_q[X]/(f(x))$

decision

Also the same as before, but we are instead doing calculations in the ring instead of using the inner product.

2.2.3 MLWE

Module learning with error (MLWE) reuses the concept from RLWE but instead of having only a single polynomial, we instead work on vectors of polynomials. Think vectors just as in the LWE scheme mentioned as the very first thing. A great example (baby kyber) of how the MLWE version of the scheme can be found at

https://cryptopedia.dev/posts/kyber/

This problem is especially thought as being secure against quantum computers. Which can be seen as the recent choice of Kyber as the standard post quantum cryptography algorithm.

search

The search problem is defined as the problem of finding the error vector $s \in R_q^k$ given a polynomial amount of keys from the MLWE scheme.

decision

The decision problem is defined by distringuishing uniformly chosen

$$(a_i, b_i) \leftarrow_{uniform} (R_q^k, R_q)$$

from the parameters that are generated in the exact scheme.

2.2.4 Relation

The exists a very simple relation between those problems. Intuitivly you can see the MLWE problem as k amount of RLWE problems (reducing it down to just working on polynomials), where k is the amount of polynomials in the vector. Each RLWE problem can then be seen as n amount of LWE problems, where n is the degree of the polynomials in question.

All in all you will get $n \cdot k$ LWE problems from a MLWE problem and the most efficient attack known against MLWE will be to attack the underlying $n \cdot k$ LWE problems, by using either LLL, Primal, Dual or BKZ attacks.

2.2.5 Extra

Those notes could also have included

- The BKZ algorithm
- How to use Martin Albrecht tool for estimating LWE security of Params.
- LLL, Primal, Dual etc. (lots of lattice attacks).
- Remember that Regev has shown that search and decision is equal for the LWE scheme (Marcus Kyber report).
- we could have written vectors in **bold** but I was lazy.
- Probably need a source to the original crypto system by Regev
- more information can be found at wikipedia or Marcus brain

Which will depend more on what we decide regarding MPC protocols etc. the above mentioned extra things, could be used to prove / argue for the security of those things.

2.3 Rényi Divergence

When measuring the distance between two distributions we can use the Rényi Divergence (RD) as an alternative to the Statistical Distance (SD).

The RD of order $a \in (1, +\infty)$ for two discrete probability distributions P and Q with $\text{Supp}(P) \subseteq \text{Supp}(Q)$ is defined ([BS23]) as

$$R_a(P||Q) = \left(\sum_{x \in \text{Supp}(P)} \frac{P(x)^a}{Q(x)^{a-1}} \right)^{\frac{1}{a-1}}$$

Given distributions P, Q with $\text{Supp}(P) \subseteq \text{Supp}(Q)$ and $a \in (1, +\infty)$ the following three properties hold:

Data Processing Inequality: For any function f, let $f(P)$ (resp. $f(Q)$) denote the distribution of $f(y)$ induced by sampling $y \leftarrow P$ (resp. $y \leftarrow Q$), then

$$RD_a(f(P)||f(Q)) \leq RD_a(P||Q)$$

Probability Preservation: Given an event $E \subset \text{Supp}(Q)$ and $a \in (1, \infty)$ it holds that

$$Q(E) \cdot RD_a(P||Q) \geq P(E)^{\frac{a}{a-1}}$$

Multiplicativity: Given two random variables (Y_1, Y_2) sampled from the probability distributions P, Q, we let P_i denote the marginal distribution of Y_i under P (resp. Q). Let $P_{2|1}(\cdot|y_1)$ (resp $Q_{2|1}(\cdot|y_1)$) denote the conditional distribtion of Y_2 given $Y_1 = y_1$, then for $a \in (1, \infty)$

$$RD_a(P||Q) \leq RD_a(P_1||Q_1) \cdot \max_{y_1 \in Y_1} RD_a(P_{2|1}(\cdot|y_1)||Q_{2|1}(\cdot|y_1))$$

Applications: In general, using the Rényi Divergence as opposed to the SD seems to be especially useful for search problems ([BLL+15]), whereas in the case of distinguishing problems, the multiplicativity property of RD appears to make it less useful. In general the SD has properties similar to the RD, but instead of the multiplicativity property, it has an additive property.

Distributions such as continuous Gaussian distributions and Gaussian distributions with lattice supports often come up in security proofs of *lattice-based cryptography.* The RD appears to work well for quantifying the distance between Gaussian distributions, so by replacing the SD by the RD it is possible to improve some of these.

The following is a list of concrete applications, as proven in [BLL+15]:

1. Smaller storage requirements for the Fiat-Shamir BLISS signature scheme
2. Smaller parameters in the dual-Regev encryption scheme
3. Alternative and more general proof that LWE with noise chosen uniformly in an interval is no easier than the LWE problem with Gaussian noise.
4. Alternative proof that the LWR problem is no easier than LWE. This reduction preseves the dimension for a composite LWE modulus that is a multiple of the LWR rounding modulus, without using noise flooding.

2.4 Threshold Public Key Encryption

A Threshold Public Key Encryption (TPKE) scheme is formally defined ([BS23]) to be a tuple of PPT algorithms TPKE $=$ (Setup, Enc, PartDec, Combine), which are defined in the following manner: Only included TPKE and not TFHE here, is this fine?

Setup$(1^\lambda, n, t) \rightarrow (pp, pk, sk_1, ..., sk_n)$: Given security parameter λ, number of parties n, and a threshold value $t \in 1, ..., n-1$, Setup outputs the public parameters pp, a public key pk, and a set of secret key shares $sk_1, ..., sk_n$.

Enc$(pk, m) \rightarrow ct$: Given public key pk and message m this algorithm outputs a ciphertext ct.

PartDec$(sk_i, ct) \rightarrow d_i$: On input a key share sk_i for some $i \in [n]$ and a ciphertext ct, the algorithm outputs a decryption share d_i.

Combine$(\{d_i\}_{i\in S}, ct) \rightarrow m'$: Given a set of shares $\{d_i\}_{i\in S}$ and a ciphertext ct, s.t. $S \subset [n]$ is of size at least $t+1$, this algorithm outputs a message $m' \in \mathcal{M} \cup \{\perp\}$.

2.4.1 Notions of security for TPKE schemes

OW-CPA Security for TPKE

Broadly speaking, the OW-CPA security notion for PKE schemes encapsulates the fact that given a public key pk and an encryption $c = \text{Enc}_{pk}(m)$, it is hard to find m.

We now state the extension of OW-CPA security to TPKE schemes as defined by [BS23].

Definition (ℓ**-OW-CPA for TPKE). A TPKE scheme is ℓ-OW-CPA** secure for security parameter λ, the threshold parameters n, t and query bound ℓ, if for all PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

$$\text{Adv}^{\ell\textbf{-OW-CPA}}_{\text{TPKE}}(\mathcal{A}) := \Pr[\text{Expt}^{\ell\textbf{-OW-CPA}}_{\mathcal{A},\text{TPKE}}(1^\lambda, n, t) = 1] = \text{negl}(\lambda)$$

Where the experiment $\text{Expt}^{\ell\textbf{-OW-CPA}}_{\mathcal{A},\text{TPKE}}$ and OPartDec(m) are defined as in Figure **??**

```
Expt^{ℓ-OW-CPA}_{A,TPKE}
1: $(\text{pp}, \text{pk}, \text{sk}_1, ..., \text{sk}_n) \leftarrow \textsc{Setup}(1^\lambda, n, t)$
2: $S \leftarrow \mathcal{A}_1(\text{pp}, \text{pk}) : S \subset [n] \wedge |S| \leq t$
3: $m \leftarrow U(\mathcal{M})$
4: $ct \leftarrow \textsc{Enc}(\text{pk}, m)$
5: $m' \leftarrow \mathcal{A}_2^{\text{OPartDec}}(\text{pk}, \{\text{sk}_i\}_{i\in S}, ct)$
6: return $m = m'$
```

```
OPartDec(m)
1: $ctr = ctr + 1$
2: if $ctr > \ell$ then return $\perp$
3: if $m \notin \mathcal{M}$ then return $\perp$
4: $ct \leftarrow \textsc{Enc}(\text{pk}, m)$
5: $\rho =$ randomness used for Enc
6: $d_i \leftarrow \textsc{PartDec}(\text{sk}_i, ct) \quad i \in [n]$
7: return $\rho, (d_i)_{i\in[n]}$
```

Figure 2.1: Definition of the experiment $\text{Expt}^{\ell\textbf{-OW-CPA}}_{\mathcal{A},\text{TPKE}}$ and oblivious partial decryption, OPartDec, respectively

IND-CPA Security for TPKE

In IND-CPA security for PKE schemes the adversary is given a public key pk. Then the adversary inputs a message m. The adversary must then not be able to distinguish between an encryption of m, $Enc_{pk}(m)$, and an encryption of a random message r, $Enc_{pk}(r)$, except with negligible probability.

As with OW-CPA security we now state the extension to TPKE schemes as specified in [BS23].

Definition (**ℓ-IND-CPA for TPKE**). A TPKE scheme is **ℓ-IND-CPA** secure for security parameter λ, the threshold parameters n, t and query bound ℓ, if for all PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4)$

$$\text{Adv}_{\text{TPKE}}^{\boldsymbol{\ell}\textbf{-IND-CPA}}(\mathcal{A}) := |\Pr[\text{Expt}_{\mathcal{A},\text{TPKE}}^{\boldsymbol{\ell}\textbf{-IND-CPA}}(1^\lambda, n, t) = 1] - \frac{1}{2}| = \text{negl}(\lambda)$$

Where $\text{Expt}_{\mathcal{A},\text{TPKE}}^{\boldsymbol{\ell}\textbf{-IND-CPA}}$ is the following experiment

```
1: $(\text{pp}, \text{pk}, \text{sk}_1, ..., \text{sk}_n) \leftarrow \textsc{Setup}(1^\lambda, n, t)$
2: $S \leftarrow \mathcal{A}_1(\text{pp}, \text{pk}) : S \subset [n] \wedge |S| \leq t$
3: $state \leftarrow \mathcal{A}_2^{\text{OPartDec}}(\text{pk}, \{\text{sk}_i\}, ct)$
4: $b \leftarrow U(\{0, 1\})$
5: $(m_0, m_1) \leftarrow \mathcal{A}_3(\text{pp}, \text{pk}, \{\text{sk}_i\}_{i \in S})$
6: $ct \leftarrow \textsc{Enc}(\text{pk}, m_b)$
7: $b' \leftarrow \mathcal{A}_4^{\text{OPartDec}}(\text{pk}, \{\text{sk}_i\}_{i \in S}, ct_b)$
8: **return** $b = b'$
```

With $\text{OPartDec}(C, m_1, ..., m_k)$ defined as in section 2.4.1.

2.4.2 TPKE construction from [BS23]

In [BS23] the authors describe how to achieve an IND-CPA secure TPKE scheme. This is done by first constructing an OW-CPA secure TPKE, then using a transformation on that scheme to attain an IND-CPA secure TPKE scheme.

OW-CPA secure TPKE

The OW-CPA secure TPKE is parametrised by two noise distributions $\mathcal{D}_{\text{flood}}$, $\mathcal{D}_{\text{sim}}$, a LSS scheme, and an OW-CPA secure PKE scheme.

The noise distribution $\mathcal{D}_{\text{flood}}$ is over the ring $\mathbb{Z}_q$ with magnitude bounded by β_{flood}.

The other noise distribution, $\mathcal{D}_{\text{sim}}$, is also over $\mathbb{Z}_q$, where $\text{RD}_a(\mathcal{D}_{\text{sim}}||\mathcal{D}_{\text{flood}}) \leq \epsilon_{\text{RD}_a}$, for $a \in (1, \infty), \epsilon_{\text{RD}_a} > 1$, and for all B with $|B| \leq \beta_{\text{fhe}}$.

The LSS scheme is a t-out-of-n linear secret sharing scheme $\text{LSS} = (\text{Share}, (\text{Rec}_S)_{S \subset [n]})$ with strong $\{0, 1\}$-reconstruction and parameters L, $\tau_{\max}, \tau_{\min}$ and shares in $\mathbb{Z}_q^L$.

Finally, the OW-CPA secure PKE scheme is a tuple $(\textsc{Setup}', \textsc{Enc}, \textsc{Dec})$, with $\mathcal{M} \subseteq R_p$, $\mathcal{C} = R_q^r$, and $(\beta_{\text{fhe}}, \epsilon)$-linear decryption for some $\beta_{\text{fhe}} < q/(2p) - \tau_{\min}\beta_{\text{flood}}$ and negligible ϵ.

Er overstående for tæt på beskrivelsen i artiklen?

Constructing the scheme is then done by defining the $\textsc{Enc}$ algorithm as in the PKE scheme, while $\textsc{Setup}$, $\textsc{PartDec}$, and $\textsc{Combine}$ are as described in figure **??**.

```
**Setup**$(1^\lambda, n, t)$
  1. $(pp, pk, sk) \leftarrow \text{Setup}'(1^\lambda)$
  2. $(sk_1, ..., sk_n) \leftarrow LSS.Share(sk)$
  3. Return $(pp, pk, sk_1, ..., sk_n)$
```

```
**PartDec**$(sk_i, ct)$
  1. $e_{i,j} \leftarrow \mathcal{D}_{\text{flood},R_q}$ for $j \in [L]$
  2. $d_{i,j} \leftarrow \langle ct, sk_{i,j} \rangle + e_{i,j}$
  3. Return $d_i \leftarrow (d_{i,1}, ..., d_{i,L})$
```

```
**Combine**$(\{d_i\}_{i \in S}, ct)$
  1. $y \leftarrow \text{Rec}_S((d_i)_{i \in S})$
  2. Return $\lfloor (p/q) \cdot y \rceil$
```

Figure 2.2: Setup, PartDec, and Combine algorithms for the OW-CPA secure TPKE.

Transformation from OW to IND

The transformation from One-Wayness to Indistinguishability is parametrized by $\delta \in \mathbb{N}$, controlling a tradeoff between ciphertext compactness and security loss of the reduction. In addition, the transformation also uses the two random oracles $F : \mathcal{M}^\delta \to \mathcal{M}'$ and $G : \mathcal{M}^\delta \to \{0,1\}^{2\lambda}$.

Given an OW-CPA secure TPKE (Setup, Enc, PartDec, Combine) the IND-CPA secure TPKE (Setup', Enc', PartDec', Combine') is constructed as follows:

Setup$'(1^\lambda, n, t)$: Run Setup$(1^\lambda, n, t)$.

Enc$'(pk, m)$: Sample $\mathbf{x} := (x_1, ..., x_\delta) \leftarrow U(\mathcal{M}^\delta)$, set $c_0 = m + F(\mathbf{x})$, $c_{\delta+1} = G(\mathbf{x})$, and $c_j = \text{Enc}(\text{pk}, x_j)$ for $j \in [\delta]$. Output $ct := (c_0, ..., c_{\delta+1})$.

PartDec$'(sk_i, ct)$: On input (sk_i, ct) for $i \in [n]$, compute $d_{ij} \leftarrow \text{PartDec}(sk_i, c_j)$ for $j \in [\delta]$ and output $\mathbf{d}_i := (d_{ij})_{j \in [\delta]}$.

Combine$'(\{d_i\}_{i \in S}, ct)$: Given input $((\mathbf{d}_i)_{i \in S}, ct)$ where $ct = (c_j)_{0 \le j \le \delta+1}$ and $\mathbf{d}_i = (d_{ij})_{j \in [\delta]}$, compute $x'_j \leftarrow \text{Combine}(\{d_{ij}\}_{i \in S}, c_j)$ for $j \in [\delta]$. Then, set $\mathbf{x}' = (x'_1, ..., x'_\delta)$ and compute $m' := c_0 - \text{F}(\mathbf{x}')$. If $c_{\delta+1} = \text{G}(\mathbf{x}')$ output m', else output $\perp$.

Parameters

If the construction described is instantiated using the PKE scheme Kyber ([Pet23]), then this yields the thresholdized Kyber TPKE scheme, TKyber.

Extra

- Explain strong $\{0,1\}$-reconstruction
- Explain LSS parameters, i.e. $\tau_{\min}, \tau_{\max}$
- Explain Linearity property of LSS schemes.

2.5 Gladius: Hybrid encryption scheme with distributed decryption

Gladius ([CCMS21]) is an encryption scheme constructed with the aim of allowing distributed decryption for a hybrid post-quantum encryption scheme. The scheme is built by constructing a deterministic encryption scheme Π_p, then using one of two different transformations presented by the authors to attain a hybrid encryption scheme.

The security of the deterministic encryption scheme Π_p is based on the hardness of either the Learning-With-Rounding (LWR) problem or the Module-LWR (MLWR) problem, depending on the variant of Gladius used. The LWR and MLWR problems are closely related to the LWE and MLWE problems respectively.

Gladius comes in 4 different variants: Gladius-Hispaniensis, Gladius-Pompeii, Gladius–Mainz, and Gladius-Fulham. In this section we will focus on Gladius-Hispaniensis, which is based on plain LWR, since the distributed decryption procedure is presented in [CCMS21] using Gladius-Hispaniensis which uses the Hybrid$_1$ transformation.

The main goal of studying Gladius is to apply the techniques for distributed decryption, to achieve threshold decryption for a hybrid encryption scheme based on Kyber.

2.5.1 LWR

The standard LWR problem is defined as follows: An LWR sample is defined as $(A, \lfloor A \cdot \mathbf{s} \rceil_p)$, where $A \in \mathbb{Z}_q^{m \times d}$ uniformly random, and $\mathbf{s} \in \mathbb{Z}_q^d$. In the decision variant of the LWR problem the goal is then to distinguish between LWR samples and $(A, \lfloor \mathbf{u} \rceil_p)$ for uniformly random $\mathbf{u} \in \mathbb{Z}_q^d$. For the search variant the goal is instead to extract $\mathbf{s}$ from a number of LWR samples. Giver nok mening at bruge samme notation som vi gør for LWE. Mangler MLWR, er denne nødvendig?

2.5.2 Hybrid$_1$ and Hybrid$_2$ constructions

Now we describe the two transformations that the different variants of Gladius use to achieve a hybrid encryption scheme.

Hybrid$_1$

Hybrid$_2$

2.5.3 Gladius-Hispaniensis

The deterministic encryption scheme used by the Gladius-Hispaniensis variant of Gladius is based on regular LWR. This particular variant is parametrised by $t, p, q, n, \ell, \sigma, \epsilon$. In addition μ and $\psi \in (-1/2, 1/2]$ are defined as

$$\frac{p \cdot \ell}{q} = \left\lfloor \frac{p \cdot \ell}{q} \right\rceil + \psi = \mu + \psi$$

We also need an error distribution $\mathcal{D}_\sigma$, which the authors of gladius define to be a distribution similar to a discrete gaussian distribution with standard deviation σ. In practice done as described in NewHope. Now we are ready to describe the deterministic encryption scheme Π_p itself.

Keygen: Sample $n \times n$ matrices (R_1, R_2) with entries from $\mathcal{D}_\sigma$ Sample a uniformly random matrix $A_1 \in \mathbb{Z}_q^{n \times n}$. Set $A_2 = A_1 \cdot R_1 + R_2 + G$, where G is the gadget matrix $\ell \cdot I_n$. Output $\mathrm{pk} = (A_1, A_2)$, $\mathrm{sk} = (\mathrm{pk}, R_1)$.

Enc(pk, $\mathbf{m}$): Compute and output $(\mathbf{c}_1, \mathbf{c}_2) = (\lfloor \mathbf{m}^T \cdot A_1 \rceil_p, \lfloor \mathbf{m}^T \cdot A_2 \rceil_p)$

Dec(sk, $(\mathbf{c}_1, \mathbf{c}_2)$): Compute $\mathbf{w}^T = \mathbf{c}_2 - \mathbf{c}_1 \cdot R_1 \pmod{q}$. Set $\mathbf{e}^T = \mathbf{w}^T \pmod{p}$ and $\mathbf{v}^T = \mathbf{w}^T \pmod{\mu}$. Then compute $\mathbf{m}^T = (\mathbf{e}^T - \mathbf{v}^T)/\mu$. Now, re-encrypt $(\mathbf{c}_1', \mathbf{c}_2') = \text{Enc}(\text{pk}, \mathbf{m})$. If $\mathbf{c}_1 \neq \mathbf{c}_1'$ or $\mathbf{c}_2 \neq \mathbf{c}_2'$ output $\perp$. Finally, output $\mathbf{m}^T$

2.5.4 Distributed Key Generation

In addition to providing a protocol for distributed decryption, the article also presents a protocol for generating a key pair in a distributed manner. The protocol for this can be seen in Figure 2.3.

Protocol for Distributed Key Generation for Kyber $\Pi_{\text{DKeyGenKyber}}$

1. For $i, j \in [1, ..., n]$
   - $\langle b \rangle, \langle b' \rangle, \langle c \rangle, \langle c' \rangle \leftarrow \text{Bits}()$
   - $\langle R_1^{(i,j)} \rangle \leftarrow \langle b \rangle - \langle b' \rangle$
   - $\langle R_2^{(i,j)} \rangle \leftarrow \langle c \rangle - \langle c' \rangle$
   - $A^{(i,j)} \leftarrow \mathbb{F}_q$
2. $\langle A_2 \rangle \leftarrow A_1 \cdot \langle R_1 \rangle + \langle R_2 \rangle + G$
3. $A_2 \leftarrow \textbf{Output}(\langle A_2 \rangle)$
4. $\text{pk} \leftarrow (A_1, A_2)$
5. $\text{sk} \leftarrow (A_1, A_2, \langle R_1 \rangle)$

Figure 2.3: A protocol for distributed key generation for Gladius-Hispaniensis

2.5.5 Distributed Decryption

For the distributed decryption (DDec) protocol we need sub-protocols for bit decomposition, bit addition, bit negation, and bit less than. These protocols will now be explained in depth.

BitDecomp: The BitDecomp procedure is used for decomposing a shared value from $\mathbb{F}_q$, into a number of shared bits representing the bit decomposition of the shared value. So given $\langle a \rangle$, where $a \in \mathbb{F}_q$, the protocol computes a vector of bits $\langle \mathbf{a} \rangle = (\langle a_0 \rangle, ..., \langle a_{\lfloor \log_2 q \rfloor} \rangle)$ s.t. $a = \sum_i 2^i a_i$.

Bit addition: Given as input the shared bits $\langle a \rangle$, $\langle b \rangle$, BitAdd outputs the vector of shared bits $\langle c \rangle$, s.t. $\sum_i c_i 2^i = \sum_i (a_i + b_i) 2^i$.

Bit negation: This operation flips the bits of $\langle a \rangle$ to get $\langle \overline{\mathbf{a}} \rangle$, then computes BitAdd$(\langle \overline{\mathbf{a}} \rangle, \mathbf{1})$, where $\mathbf{1} = \text{BitDecomp}(1, |\overline{\mathbf{a}}|)$ is a bit vector of the correct length representing 1.

Bit less than: Computes a shared output bit $\langle c \rangle$ of the comparison $\sum_i a_i \cdot 2^i < \sum_i b_i \cdot 2^i$.

Centre

When using BitDecomp we get the decomposition of a in the interval $[0, q)$, but we want a to be in the interval $(-q/2, ..., q/2)$ for DDec. To achieve a centered interval we use use BitAdd, BitNeg, and BitLT. The protocol for doing this looks as follows:

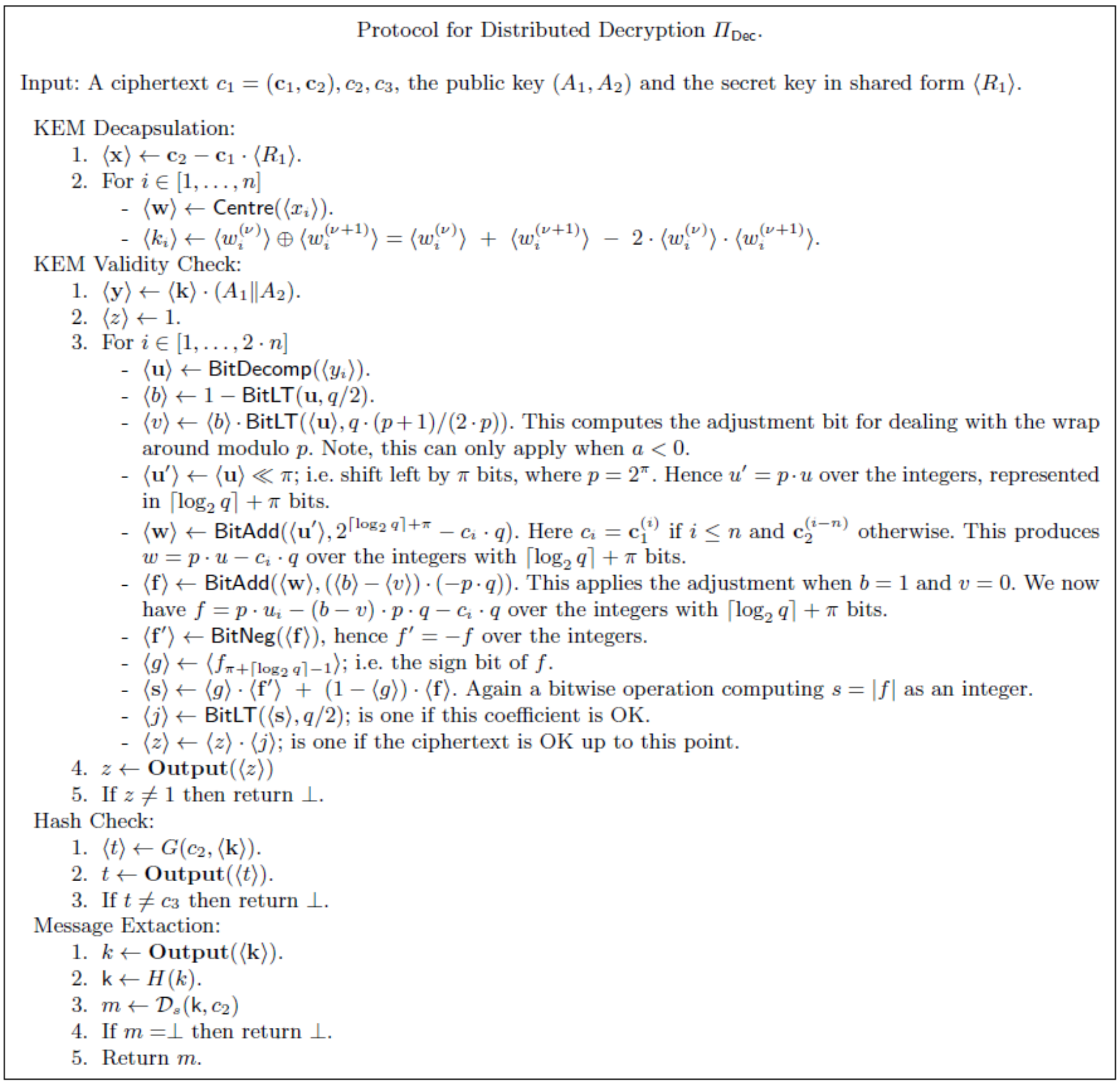
```
Subroutine Centre($\langle x \rangle$)

1. $\langle \mathbf{b} \rangle \leftarrow$ BitDecomp($\langle x \rangle$)
2. $\langle \mathbf{b}' \rangle \leftarrow$ BitAdd($\langle \overline{\mathbf{b}} \rangle, q+1$) // Computes $b' = q - u_i$ over integers.
3. $\langle \mathbf{b}'' \rangle \leftarrow$ BitNeg($\langle \mathbf{b}' \rangle, q+1$)
4. $\langle \mathbf{f} \rangle \leftarrow$ BitLT($\langle \mathbf{b} \rangle, q/2$)
5. $\langle \mathbf{a} \rangle \leftarrow \langle f \rangle \cdot \langle \mathbf{b} \rangle + (1 - \langle f \rangle) \cdot \langle \mathbf{b}'' \rangle$
6. Return $\langle \mathbf{a} \rangle$
```

Figure 2.4: The Centre subroutine used for DDec

Main protocol

The full distributed decryption protocol, which uses the operations explained above, can then be seen in Figure 2.5



```
Protocol for Distributed Decryption $\Pi_{\mathsf{Dec}}$.

Input: A ciphertext $c_1 = (\mathbf{c}_1, \mathbf{c}_2), c_2, c_3$, the public key $(A_1, A_2)$ and the secret key in shared form $\langle R_1 \rangle$.

KEM Decapsulation:
    1. $\langle \mathbf{x} \rangle \leftarrow \mathbf{c}_2 - \mathbf{c}_1 \cdot \langle R_1 \rangle$.
    2. For $i \in [1, \ldots, n]$
        - $\langle \mathbf{w} \rangle \leftarrow \mathsf{Centre}(\langle x_i \rangle)$.
        - $\langle k_i \rangle \leftarrow \langle w_i^{(\nu)} \rangle \oplus \langle w_i^{(\nu+1)} \rangle = \langle w_i^{(\nu)} \rangle + \langle w_i^{(\nu+1)} \rangle - 2 \cdot \langle w_i^{(\nu)} \rangle \cdot \langle w_i^{(\nu+1)} \rangle$.
KEM Validity Check:
    1. $\langle \mathbf{y} \rangle \leftarrow \langle \mathbf{k} \rangle \cdot (A_1 \| A_2)$.
    2. $\langle z \rangle \leftarrow 1$.
    3. For $i \in [1, \ldots, 2 \cdot n]$
        - $\langle \mathbf{u} \rangle \leftarrow \mathsf{BitDecomp}(\langle y_i \rangle)$.
        - $\langle b \rangle \leftarrow 1 - \mathsf{BitLT}(\mathbf{u}, q/2)$.
        - $\langle v \rangle \leftarrow \langle b \rangle \cdot \mathsf{BitLT}(\langle \mathbf{u} \rangle, q \cdot (p+1)/(2 \cdot p))$. This computes the adjustment bit for dealing with the wrap
          around modulo $p$. Note, this can only apply when $a < 0$.
        - $\langle \mathbf{u}' \rangle \leftarrow \langle \mathbf{u} \rangle \ll \pi$; i.e. shift left by $\pi$ bits, where $p = 2^\pi$. Hence $u' = p \cdot u$ over the integers, represented
          in $\lceil \log_2 q \rceil + \pi$ bits.
        - $\langle \mathbf{w} \rangle \leftarrow \mathsf{BitAdd}(\langle \mathbf{u}' \rangle, 2^{\lceil \log_2 q \rceil + \pi} - c_i \cdot q)$. Here $c_i = \mathbf{c}_1^{(i)}$ if $i \leq n$ and $\mathbf{c}_2^{(i-n)}$ otherwise. This produces
          $w = p \cdot u - c_i \cdot q$ over the integers with $\lceil \log_2 q \rceil + \pi$ bits.
        - $\langle \mathbf{f} \rangle \leftarrow \mathsf{BitAdd}(\langle \mathbf{w} \rangle, (\langle b \rangle - \langle v \rangle) \cdot (-p \cdot q))$. This applies the adjustment when $b = 1$ and $v = 0$. We now
          have $f = p \cdot u_i - (b - v) \cdot p \cdot q - c_i \cdot q$ over the integers with $\lceil \log_2 q \rceil + \pi$ bits.
        - $\langle \mathbf{f}' \rangle \leftarrow \mathsf{BitNeg}(\langle \mathbf{f} \rangle)$, hence $f' = -f$ over the integers.
        - $\langle g \rangle \leftarrow \langle f_{\pi + \lceil \log_2 q \rceil - 1} \rangle$; i.e. the sign bit of $f$.
        - $\langle \mathbf{s} \rangle \leftarrow \langle g \rangle \cdot \langle \mathbf{f}' \rangle + (1 - \langle g \rangle) \cdot \langle \mathbf{f} \rangle$. Again a bitwise operation computing $s = |f|$ as an integer.
        - $\langle j \rangle \leftarrow \mathsf{BitLT}(\langle \mathbf{s} \rangle, q/2)$; is one if this coefficient is OK.
        - $\langle z \rangle \leftarrow \langle z \rangle \cdot \langle j \rangle$; is one if the ciphertext is OK up to this point.
    4. $z \leftarrow \mathbf{Output}(\langle z \rangle)$
    5. If $z \neq 1$ then return $\perp$.
Hash Check:
    1. $\langle t \rangle \leftarrow G(c_2, \langle \mathbf{k} \rangle)$.
    2. $t \leftarrow \mathbf{Output}(\langle t \rangle)$.
    3. If $t \neq c_3$ then return $\perp$.
Message Extaction:
    1. $k \leftarrow \mathbf{Output}(\langle \mathbf{k} \rangle)$.
    2. $\mathsf{k} \leftarrow H(k)$.
    3. $m \leftarrow \mathcal{D}_s(\mathsf{k}, c_2)$
    4. If $m = \perp$ then return $\perp$.
    5. Return $m$.
```

Figure 2.5: Distributed decryption protocol for Gladius-Hispaniensis using Hybrid$_1$

The first step of the KEM decapsulation corresponds to computing $\mathbf{w}^T$. The second step of KEM decapsulation however corresponds to lines computing $\mathbf{e}^T$, $\mathbf{v}^T$, and $\mathbf{m}^T$. This is because for Gladius-Hispaniensis, if we choose μ and p to be powers of two, i.e. $\mu = 2^\nu$ and $p = 2^\pi$, then this part can be simplified into $m_i = w_i^{(\nu)} \oplus w_i^{(\nu+1)}$.

Re-encrypting $\mathbf{m}$ corresponds to step 1 of the KEM validity check, while the check done in decryption corresponds to steps 2, 3, 4, and 5 of the KEM validity check.

The Hash Check and Message Extraction stems from the hybrid$_1$ transformation. In the paper the authors construct the G hash function by combining SHA-3 and Rescue, which is an MPC-friendly hash function. The motivation behind defining G in this manner, is that it improves the efficiency of the protocol.

Chapter 3

Adapting the Distributed Decryption of Gladius to Kyber

3.1 Distributed Decryption using Kyber

By utilising the same method as done for Distributed Decryption in Gladius we can can do threshold decryption by putting Kyber decryption composed with the Hybrid 1 scheme inside an MPC protocol.

The motivation for doing this is that when using the approach of [BS23], all players have to add a noise term in PartDec, so we end up with more noise than what we would have otherwise had. This in turn then means that we have to increase the value of q to get enough "space" for the additional noise, to ensure correctness of decryption.

When decrypting inside an MPC protocol we do not have to add noise for each player participating, and so we get significantly less noise.

The protocol for Distributed Decryption using Kyber composed with Hybrid 1 then looks as in Figure 3.1.

In section 4.2 we describe how we realised the Distributed Decryption protocol for Kyber. In the rest of this section we describe how this protocol differs from the one described in the Gladius article, and why this protocol works.

3.1.1 KEM Decapsulation

The first major change is that KEM decapsulation is different due to the differences in how the message is derived in Gladius-Hispaniensis and Kyber.

As mentioned in section x.x, decrypting in Kyber given a ciphertext (u, v), is done by computing $c' = v - u^T s$, then outputting $\lfloor c' \cdot 2/q \rceil$. Computing c' in the distributed decryption protocol corresponds to line 1 of KEM decapsulation.

The scaling and rounding is then done in the loop on line 2. The trick that is used for this is that once we use Centre to get the bit-decomposition of w in the centered interval, then scaling and rounding corresponds to picking the most significant bit (msb) of each of the entries of $\mathbf{w}$. This works because the message space of Kyber consists of binary polynomials, and so $p = 2$. Er det her forklaret godt nok???

3.1.2 No KEM validity check

The second major difference is that we no longer have a KEM validity check, which is caused by the fact that Kyber decryption does not do a validity check like Gladius does.

3.1.3 Hash Check and Message Extraction

Apart from these changes the Hash Check and Message Extraction phases stay the same, which is due to these stemming from the Hybrid 1 transformation, which we use for Kyber directly as described for Gladius. Alternatively, it would also be possible to use the Kyber CPA

```
Protocol for Distributed Decryption using Kyber, $\Pi_{\text{DDecKyber}}$

Input: A ciphertext $c_1 = (v, u), c_2, c_3$, and the secret key in shared form $\langle s \rangle$

KEM Decapsulation:

  1. $\langle x \rangle \leftarrow v - \mathbf{u}^T \langle s \rangle$
  2. For $i \in [1, ..., n]$
       - $\langle \mathbf{w} \rangle \leftarrow \text{Centre}(\langle x_i \rangle)$
       - $\langle k_i \rangle \leftarrow \langle w_i^{(\text{msb})} \rangle$

Hash Check:

  1. $\langle t \rangle \leftarrow G(c_2, \langle \mathbf{k} \rangle)$
  2. $t \leftarrow \textbf{Output}(\langle t \rangle)$
  3. If $t \neq c_3$, return $\perp$

Message extraction:

  1. $k \leftarrow \textbf{Output}(\langle \mathbf{k} \rangle)$
  2. $\mathbf{k} \leftarrow H(k)$
  3. $m \leftarrow D_S(\mathbf{k}, c_2)$
  4. If $m = \perp$, return $\perp$
  5. return $m$
```

Figure 3.1: Distributed Decryption protocol for Kyber

PKE to IND-CCA2-secure KEM transformation presented in the original Kyber specification, but this involves more hashing, which is difficult to do inside an MPC protocol, and so to avoid these difficulties we use Hybrid 1 from the Gladius paper [CCMS21].

3.2 Distributed Key Generation for Kyber

Chapter 4

Implementation

This section describes the implementation details relating to our implementation of the TPKE scheme TKyber. We start by detailing the implementation of the three parts that make up the TKyber implementation, then we describe how we tested the correctness of our implementation.

All of the code pertaining to this thesis can be found at `https://github.com/Sellebjergen/ThresholdKyber`.

4.1 Implementing TKyber

The implementation of Thresholdized Kyber, TKyber, was written in Go. This implementation was built on top of an existing Kyber implementation in Go, kyber-k2so[1], which is listed on the official Kyber website.

The TKyber implementation is split into three parts: The LSS scheme, the OW-CPA TKyber implementation, and the IND-CPA TKyber implementation.

4.1.1 LSS Scheme

For the LSS scheme we need to be able to secret share polynomials. For working with polynomials we use the polynomial type Poly native to the kyber-k2so Kyber library.

Additive LSS

Implementing Additive LSS was simple, as kyber-k2so includes a function Sub for subtracting variables of the Poly type. In particular, let p be the polynomial that we wish to share among n parties. Then to share p we sample $n-1$ uniformly random polynomials $p_2, ..., p_n$ of the Poly type with coefficients in $[0, q)$. Then we compute

$$p_1 = p - \sum_{i=1}^{n} p_i$$

by using Sub. Following this we return a Slice[2] of Poly variables corresponding to $p_1, ..., p_n$, such that we can distribute p_i to party i.

4.1.2 OW-CPA TKyber

To be able to vary the LSS scheme used as well as the noise distribution we utilise a strategy design pattern. We then use a struct OwcpaParams to specify the Kyber parameters, along with the LSS scheme and noise distribution to use.

[1]`https://github.com/SymbolicSoft/kyber-k2so`

[2]A slice in Go is a dynamic sized view of an array.

Setup

For the Setup function we use IndcpaKeypair from kyber-k2so to sample a keypair. Then we unpack the secret key into a vector of polynomials. Finally, we secret share each of the polynomials in the secret key using the LSS scheme. At the end we return the public key and the secret key shares $s_1, ..., s_n$.

Enc

The Enc function picks random coins, then uses these with IndcpaEncrypt from kyber-k2so to encrypt the message given as input.

PartDec

For PartDec we sample noise terms $e_{i,j}$ using the distribution specified in the OwcpaParams struct given as input. Then we do the computation corresponding to line 2 of PartDec in figure **??**. Since we instantiate the TPKE using Kyber this means that we compute

$$d_{i,j} = v \cdot 1_{i,j} - \mathbf{u}^T s_{i,j} + e_{i,j}$$

and set the i'th players share to be $d_i = (d_{i,j})_{j \in [L]}$, where $1_{i,j}$ is a share of 1, since we need to ensure that we don't end up with too many v terms once we combine.

Computing the above formula is done by using the operations on Poly values that kyber-k2so provides. In particular, the inner product is computed by using PolyvecNtt to get u on NTT form, then PolyvecPointWiseAccMontgomery is used to compute the inner product, and finally PolyInvNttToMont is used to get the result in the form of a Poly not on NTT form. The addition and subtraction is then performed using PolyAdd and PolySub. Uddyb måske?.

Combine

In combine we get t decryption shares d_i as input and recombine using our LSS implementation to get the result y, which is returned. Notice that we don't any scaling in the Combine function as done in Figure **??**. This is because scaling is already done in the methods PolyFromMsg and PolyToMsg from kyber-k2so. These methods are however not used when using Combine as a subroutine in the Combine algorithm of IND-CPA TKyber, so scaling needed to be implemented separately. The functions for scaling are called Upscale and Downscale, and they return $\lfloor (q/p) \rceil \cdot y$ and $\lfloor (p/q) \cdot y \rceil$ respectively when given y as input.

4.1.3 IND-CPA TKyber

Now we turn to describing our implementation of the transformation from OW-CPA to IND-CPA.

Realising the Random Oracles F and G

For the transformation we need two random oracles $\mathrm{F} : \mathcal{M}^\delta \to \mathcal{M}'$ and $\mathrm{G} : \mathcal{M}^\delta \to \{0,1\}^{2\delta}$.

Implementing F was done by converting all of the polynomials given as input to bytes using PolyToBytes from kyberk2so, hashing the concatenation of these using Shake256 to get a byte slice of size $13 \cdot (3329/8) + 12$, then using PolyFromBytes from kyberk2so to convert the resulting bytes into a polynomial of the Poly type. The reasoning behind using Shake256 is that it is an XOF from sha3, so we can specify the output length.

The implementation of G is very similar. Here we once again convert the polynomials given as input to bytes, then hash using Shake256 to get a byte slice of 2λ bits, which we return directly.

Setup

For the IND-CPA Setup function we just run Setup from the owcpa_TKyber package and return the public key and secret key shares returned by it.

Enc

In Enc we first sample δ uniformly random binary polynomials of degree $d = 255$, which we store in a slice x. Then we compute $c_0 = m + \mathrm{F}(x)$ using PolyAdd from kyber-k2so. Then for each polynomial $x[i]$ in the slice x, we upscale $x[i]$, convert the upscaled polynomial to a message of bytes, then encrypt using Enc from owcpa_TKyber and store the resulting ciphertext. Finally, we compute $c_{\delta+1} = \mathrm{G}(x)$, then return a struct containing it along with c_0 and the encryptions of the upscaled $x[i]$'s.

PartDec

When the PartDec function is called with some sk_i as input we loop over all of the encryptions in the ciphertext ct. For each of these we then compute the partial decryption using PartDec from owcpa_TKyber, then we return a slice containing these.

Combine

For Combine we iterate over the Slices of partial decryptions $d_j = (d_{i,j})_{i \in [n]}$ given as input. For each we use Combine from owcpa_TKyber to attempt to recompute the plaintext x_j, which is then subsequently downscaled using Downscale from owcpa_TKyber, then stored in a Slice x_prime.

Afterwards we subtract F(x_prime) from the value c_0 stored in the ciphertext, and we store this value in variable mp.

We then check whether the value $c_{\delta+1}$ stored in the ciphertext is equal to G(x_prime). If this is the case we return mp, otherwise we call panic, such that the code fails.

4.1.4 Testing of correctness

Testing the correctness of the TKyber implementation was done using automated tests using Go's built in testing framework.

4.1.5 Parameter Selection

In this section we describe some of the considerations associated with picking the parameters of TKyber. We then use these parameters in the benchmarking section to evaluate the performance of our implementation.

4.2 Implementing Gladius style DDec for Kyber

Currently planning to use [Kel20]. SCALE-MAMBA was also considered, but is no longer actively developed.

Chapter 5

Benchmarking

Chapter 6

Conclusion

Acknowledgments

...

Bibliography

[BLL$^+$15] Shi Bai, Adeline Langlois, Tancrëde Lepoint, Amin Sakzad, Damien Stehle, and Ron Steinfeld. Improved security proofs in lattice-based cryptography: using the rényi divergence rather than the statistical distance. Cryptology ePrint Archive, Paper 2015/483, 2015. https://eprint.iacr.org/2015/483.

[BS23] Katharina Boudgoust and Peter Scholl. Simple threshold (fully homomorphic) encryption from lwe with polynomial modulus. Cryptology ePrint Archive, Paper 2023/016, 2023. https://eprint.iacr.org/2023/016.

[CCMS21] Kelong Cong, Daniele Cozzo, Varun Maram, and Nigel P. Smart. Gladius: Lwr based efficient hybrid public key encryption with distributed decryption. Cryptology ePrint Archive, Paper 2021/096, 2021. https://eprint.iacr.org/2021/096.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[Pet23] Peter Schwabe et al. CRYSTALS-KYBER. National Institute of Standards and Technology, 2023. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.