Threshold Decryption Based on Kyber

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

Threshold decryption is the act of secret sharing a key, such that $t + 1$ shares are needed to be able to decrypt, where t is called the threshold parameter. The aim of this is to mitigate issues relating to having a single point of failure, which would be the case if a single party held the secret key.

The main goal of this study was to find out whether it was possible to construct and implement a distributed decryption protocol for doing threshold decryption using Kyber, based on Multiparty Computation (MPC), such as what has been done for the hybrid encryption scheme Gladius by Cong et al. Another goal was to compare such an implementation to another approach for threshold decryption from LWE by Boudgoust et al. In this approach, parties compute partial decryptions, then add a noise term to avoid leakage of information. All of the partial decryptions can then be combined to get the originally encrypted message back. When this is instantiated using Kyber, we call the resulting scheme TKyber.

To be able to answer these questions, we used the techniques from the Gladius paper to first construct, then implement, benchmark, and analyze a distributed decryption procedure for Kyber. We then also implemented TKyber and distributed decryption for the most basic variant of Gladius, Gladius-Hispaniensis, to be able to compare them with distributed decryption for Kyber.

It was found that the Kyber KEM distributed decryption took 10.33, 10.67, and 12.97 seconds for the parameter sets Kyber512, Kyber768, and Kyber1024, respectively. For TKyber, we found that Setup used between 0.106 and 0.358 ms, Enc used between 0.0848 and 0.0881 ms, PartDec used between 0.0171 and 0.131 ms, and lastly, Combine used between 0.00594 and 0.00853 ms to execute for the parameter sets chosen.

From our results, we found that it was indeed possible to implement a Gladius-inspired distributed decryption algorithm for Kyber. We also found that our implementation of TKyber was significantly faster than our implementation of distributed decryption for Kyber and that the primary limiting factor of the Kyber DDec implementation was the use of distributed hashing.

Marcus Sellebjerg, Kasper Ilsøe
Aarhus, June 15th, 2023.

Contents

Section 1

Introduction

Usually, any public-key encryption scheme would have only one party generate the secret key sk and the public key pk. However, as one party exclusively holds the secret key, we end up with a single point of failure should this party be compromised.

As such, an emerging cryptographic problem is that of threshold decryption, in which a secret key is split into n shares $sk_1, sk_2, \ldots, sk_n$, such that several shares are needed to be able to decrypt. In particular, for the *t-out-of-n* setting, $t+1$ shares are required to be able to decrypt. Thus, any adversary needs to corrupt $t+1$ of the parties compared to earlier, thereby avoiding any issues relating to having a single point of failure. Additionally, the U.S.-based National Institute of Standards and Technology (NIST) has recently started a standardization process for Threshold Cryptography, which further highlights the importance of threshold decryption.

Another critical area of research is that of Post-Quantum Cryptography (PQC). Here, increasingly powerful quantum computers can break large instances of problems thought to be hard on conventional computers, such as factorization or computing the discrete logarithm. This motivates the need for new public-key encryption (PKE) schemes that are resistant to quantum computers.

Tools for building such PKE schemes, resistant to quantum computers, rely on the use of the learning with errors (LWE) problem, learning with rounding (LWR) problem, or their ring and module variants.

A recent development in PQC is the standardization of the cryptosystem Kyber, a system based on LWE, by NIST, which makes it interesting to look at threshold decryption for Kyber specifically. As Kyber uses Module-LWE, it has some very nice mathematical properties, one of which is that Kybers decryption algorithm can almost be seen as a linear function of the secret key. This property is especially useful, as each party can operate on the ciphertext using their own share of the secret key.

We look at two different constructions for Threshold Cryptography. The first is an encryption scheme based on a variant of the LWE assumption, allowing *t-out-of-n* threshold decryption by Boudgoust et al. [BS23]. This scheme is called TKyber when instantiated using Kyber. The second construction is a hybrid encryption scheme named Gladius [CCMS21], which allows a more general distributed decryption protocol through the use of Multiparty-Computation (MPC).

We aim to answer the following research question:

> *"Is it possible to adapt and implement a distributed decryption procedure for threshold decryption using Kyber, similar to the one for Gladius by Cong et al.* [CCMS21]. *If it is possible, how does it compare to a specialized approach for threshold decryption from LWE by Boudgoust et al.* [BS23]*."*

It is hypothesized that TKyber is faster than a Gladius-inspired protocol for distributed decryption based on Kyber. This is because the TKyber construction is a specialized approach that can potentially exploit specific properties of the LWE problem, such as nearly linear decryption. On the other hand, the approach used by Gladius is more generic and can be applied to a broader range of schemes. The Gladius approach also makes heavy use of MPC, which is associated with a large overhead in computation.

The thesis starts with a review of some preliminaries in section 2. In section 3, we present two different constructions for threshold decryption, TKyber and Gladius. In section 4, we adapt the techniques used by Gladius to the Kyber KEM. Throughout section 5, we present our implementation of TKyber, Gladius, and the Gladius-inspired distributed decryption protocol for the Kyber KEM. We analyze and benchmark the implementations in section 6. Finally, the conclusions of the work are presented in section 7.

Section 2

Preliminaries

In the following section, we will introduce some of the notation and preliminaries that are used for the rest of this thesis.

2.1 Linear Secret sharing schemes

An important cryptographic primitive for threshold cryptography is secret sharing. Secret sharing is the technique of sharing a secret by distributing shares to n parties, such that at least $t + 1$ of these parties have to collaborate by combining their shares in order to reconstruct the secret, t being the number of tolerated adversaries in the system. For Linear Secret Sharing (LSS) schemes, we also require linearity, which means that the reconstruction algorithm must be a linear map.

2.1.1 Formal definition and properties

To formally define LSS schemes, we first go through some basic notation based on the notation used in [BS23].

An access structure $\mathbb{A}$ is defined by a collection of sets of parties who have to collaborate in order to access some resource. In the context of secret sharing schemes, the resource is a secret value $x \in \mathbb{Z}_q$. A monotone access structure is an access structure where for any set of parties $S \in \mathbb{A}$ if $T \subset S$ then $T \in \mathbb{A}$.

We use the notation $\boldsymbol{v}|_S$ to denote a vector of shares restricted to parties in the set S. We call a set of shares valid if it allows reconstruction of the secret, and we call a set of shares invalid if it reveals nothing about the underlying secret.

Definition 1 (Linear Secret Sharing)**.** Let q, L, and n be positive integers. A Linear secret sharing (LSS) scheme for a monotone access structure $\mathbb{A}$ is then defined by algorithms $\texttt{Share}$: $\mathbb{Z} \to (\mathbb{Z}_q^L)^n$ and $\texttt{Rec}$: $(\mathbb{Z}_q^L)^{|S|} \to \mathbb{Z}_q$. The $\texttt{Share}$ algorithm shares the secret by distributing L shares to each party, while $\texttt{Rec}$ is used to recombine the shares into the secret, assuming enough shares are available. The algorithms need to satisfy Privacy, Reconstruction, and Linearity. These properties are defined as follows:

Privacy: For privacy, we want any set of shares that doesn't satisfy the access structure to reveal no information about the underlying secret. In other words, given $S \notin \mathbb{A}$, $\mathbf{v} \in \mathbb{Z}_q^{L|S|}$ and any values $x, x' \in Z_q$, then x and x' are equally likely to be the shared secret. Thus it holds that $\Pr[\texttt{Share}(x)|_S = \boldsymbol{v}] = \Pr[\texttt{Share}(x')|_S = \boldsymbol{v}]$.

Reconstruction: The reconstruction algorithm $\texttt{Rec}$: $(\mathbb{Z}_q^L)^{|S|} \to \mathbb{Z}_q$ should be able to correctly reconstruct the secret given a valid set of shares. That is, for any $S \in \mathbb{A}$, any $x \in \mathbb{Z}_q$ and $\boldsymbol{v} = \texttt{Share}(x)$, the reconstruction algorithm should output $\texttt{Rec}(\boldsymbol{v}|_S) = x$.

Linearity: As mentioned earlier, the linearity property specifies that the $\texttt{Rec}$ algorithm should be a linear map. This means that for any $\alpha, \beta \in \mathbb{Z}_q$, any set S with $|S| > t$ and any vectors of shares $\boldsymbol{u}, \boldsymbol{v}$ it holds that $\texttt{Rec}(\alpha\boldsymbol{u}|_S + \beta\boldsymbol{v}|_S) = \alpha\texttt{Rec}(\boldsymbol{u}|_S) + \beta\ \texttt{Rec}\ (\boldsymbol{v}|_S)$.

2.1.2 Examples of LSS schemes with strong {0, 1}-reconstruction

In the following, we assume that we want to share some integer value $x \in \mathbb{Z}_q$. These schemes are also easily extendable to polynomials and vectors of polynomials by sharing each coefficient individually. Due to the linearity property, this would then allow us to perform R_q-linear operations on shared polynomials from R_q, which will become important later. The following schemes all satisfy strong $\{0, 1\}$-reconstruction, which is defined in Definition 2. Further details and parameters of the schemes are presented in Section 2.1.2.

Definition 2 (Strong $\{0, 1\}$-reconstruction)**.** Given any LSS scheme and any secret x with $\texttt{Share}(x) = (\boldsymbol{v_1}, \ldots, \boldsymbol{v_n})$, then for any set of shares which yields reconstruction $T \subseteq \{1, ..., n\} \times \{1, ..., L\}$, there exists a subset $T' \subseteq T$ such that $\sum_{(i,j)\in T} v_{i,j} = x$, where each $v_i = (v_{i,1}, \ldots, v_{i,L})$.

Additive secret sharing

One of the simplest LSS schemes is additive secret sharing. To share a value $x \in \mathbb{Z}_q$ among n players using additive secret sharing, sample uniformly random shares $x_1, ..., x_{n-1} \in \mathbb{Z}_q$, then set

$$x_n = x - \sum_{i=1}^{n-1} x_i$$

Share x_i is then distributed to player i. This allows for reconstruction of the original secret by adding the shares

$$x = \sum_{i}^{n} x_i$$

Here we can clearly reconstruct if we are given a valid set of shares, i.e., all of the n shares are used for recombination. Evidently, since we just add up the shares to reconstruct, the recombination algorithm is a linear map. For privacy the shares $x_1, ..., x_{n-1}$ are uniformly randomly sampled. The last share x_n is computed by subtracting something uniformly random from x, so the resulting share x_n will look uniformly random. Therefore a set of less than $t + 1$ shares reveal nothing about the underlying secret, and so any two values $x, x' \in \mathbb{Z}_q$ will look equally likely given access to t or fewer shares.

Replicated secret sharing

In Replicated Secret Sharing, each party holds $L = \binom{t}{n-1}$ shares. This allows the parties to reconstruct the secret even in the absence of some parties through replication.

To share a value x, we compute an additive sharing over each of the size-t subsets of the parties $\{1, ..., n\}$. This would mean that we get k subsets and thus the shares $x_1, ..., x_k$. For each subset, we distribute the corresponding share to the parties not in that particular subset.

To recombine, we compute $x = \sum_i^k x_i$, which would mean that one party absent from each subset would have to participate. This, of course, requires $t + 1$ parties in order to recombine.

As an illustration, consider the case of $n = 4$ and $t = 2$. This results in the 6 subsets

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

Each of these will have an associated share $x_1 \dots x_6$ such that adding these shares results in recomputing the secret x. Here party 3 and 4 gets x_1, as they are not in the first set $\{1, 2\}$, while player 2 and 4 gets x_2 as the second set is $\{1, 3\}$ and so on.

Again, since we use an additive sharing over the subsets, we have linearity and reconstruction. Additionally, the privacy property holds for the same arguments as for additive sharing, namely, any set of less than $t + 1$ shares looks uniformly random, so they reveal nothing about the underlying secret.

Naive Threshold Secret Sharing

In Naive Threshold Secret Sharing, we distribute a new sharing of the secret to each subset of size $t + 1$. In that way, this type of secret sharing can be seen as the dual of Replicated Secret Sharing since we distribute a share if a party is in a particular subset, as opposed to Replicated Secret Sharing, where a party gets a share if they are not in a subset.

Once again, linearity, reconstruction, and privacy hold for the same reason as for replicated secret sharing.

Parameters for LSS schemes

We here present some additional information about the LSS schemes. In the following, $\tau_{\max}$ is used to denote the size of the smallest maximal invalid set of shares, and $\tau_{\min}$ to denote the size of the largest minimal valid set of share elements. The parameters of the LSS schemes are then shown in Table 2.1.

Scheme	L	$\tau_{\max}$	$\tau_{\min}$
Additive	1	$n-1$	n
Replicated	$\binom{n-1}{t}$	$(n-t)(\binom{n}{t}-1)$	$\binom{n}{t}$
Naive	$\binom{n-1}{t}$	$t \cdot \binom{n}{t+1}$	$t+1$

Table 2.1: Parameters for previously presented LSS schemes.

2.2 The Learning With Errors problem and variants

In 1994 Peter Shor developed an algorithm for factorization and computing the discrete logarithm efficiently on a quantum computer [Sho97]. This algorithm is known as Shor's Algorithm, and it motivates the search for new hard math problems to obtain quantum-resistant cryptography. One particular solution is that of Learning With Errors (LWE) and its variants Ring Learning With Errors (R-LWE), Module Learning With Errors (M-LWE), and a slightly different version called Learning With Rounding (LWR).

2.2.1 Learning With Errors

We recall the definitions first stated by Oded Regev [Reg05], in which he introduced the search and decision variants of the Learning With Errors (LWE) problem. The versions presented below are based on a recent Ph.D. course by Katharina Boudgoust at Aarhus University.

To define this problem, we first state the definition of the LWE distribution, which is defined as follows.

Definition 3 (LWE distribution)**.** Let q and n be positive integers, and let χ be a distribution over all integers $\mathbb{Z}$. Assume a fixed secret $\boldsymbol{s} \in \mathbb{Z}_q^n$; then the LWE distribution is defined as samples of the form $(\boldsymbol{a_i}, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ where $a_i \leftarrow U(\mathbb{Z}_q)$ and for some small error $e_i \leftarrow \chi$ $b_i = \langle a_i, s \rangle + e_i \bmod q$.

Based on the above definition, we can then define the search variant of the LWE problem.

Definition 4 (Search LWE)**.** For any positive number of independent LWE samples $(\boldsymbol{a_i}, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ for a uniformly sampled secret $\boldsymbol{s} \in \mathbb{Z}_q^n$, the search problem asks to find the secret $\boldsymbol{s}$.

The decision variant of the problem is then stated as follows.

Definition 5 (Decision LWE)**.** Given any number of positive independent samples of LWE samples either drawn from $(\boldsymbol{a_i}, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ or from the uniform distribution $(\boldsymbol{a_i}, b_i) \leftarrow U(\mathbb{Z}_q^n \times \mathbb{Z}_q)$, the decision variant asks to distinguish these cases with a non-negligible probability.

Using the above definitions, it is possible to construct an LWE PKE scheme from this problem, as shown in Figure 2.1. This is a slightly tweaked version of the original LWE cryptosystem introduced by Oded Regev in 2005 [Reg05].

While this scheme is simple and has a nice construction, the message space becomes only a single bit, which in turn means that the scheme becomes somewhat unusable, as every single bit would have to be independently encrypted. To get a smaller and more efficient construction, we can instead base a PKE scheme on the Ring Learning With Errors (R-LWE) problem, which we introduce now.

2.2.2 Ring-LWE

The R-LWE problem is in many ways similar to the LWE problem but does operations over the ring $R_q = Z_q[X]/(f(x))$ for some polynomial $f(x)$ instead of over the integers.

Now we are ready to define the extension of the LWE search problem to the Ring-LWE setting. This variant was first described by Lyubashevsky et al. [LPR12].

Definition 6 (Search R-LWE)**.** Given any number of positive samples $(a_i(x), b_i(x)) \in R_q \times R_q$ where $b_i(x) = a_i(x) \cdot s(x) + e_i(x)$, with $s(x), e_i(x) \in R_q$, the R-LWE search problem asks to find the secret polynomial $s(x)$.

LWE Encryption scheme

Key Generation

1: Sample a random vector $\mathbf{s} \in Z_q^n$

2: Sample $\mathbf{a}_i$ uniformly at random

3: Sample e_i according to some error distribution D_{error}

4: Compute $\{c_i\}_{i=1}^m$, where $c_i = (\mathbf{a}_i, \mathbf{a}_i \cdot \mathbf{s} + e_i)$

5: Set $sk = s$ and $pk = \{c_i\}_{i=1}^m$

Encrypt

1: Message is a bit w

2: Choose random bits $b_1, ..., b_m$

3: Return the ciphertext $(\mathbf{u}, v) = \sum_i^m b_i c_i + (0, \lceil q/2 \rceil w)$

Decrypt

1: Compute $c' = v - \mathbf{s} \cdot \mathbf{u}$

2: Output 0 if c' is closer to 0 than it is to $\lceil q/2 \rceil$.

3: Output 1 otherwise.

Figure 2.1: The Learning With Errors (LWE) encryption scheme

The corresponding decision variant is then defined in the following manner.

Definition 7 (Decision R-LWE)**.** Given any number of positive samples $(a_i(x), b_i(x)) \in R_q \times R_q$ where $b_i(x) = a_i(x) \cdot s(x) + e_i(x)$, with $s(x), e_i(x) \in R_q$, the R-LWE decision variant asks to distinguish the shares $(a_i(x), b_i(x)) \leftarrow R_q \times R_q$ from ones sampled uniformly at random as follows $(a_i(x), b_i(x)) \leftarrow U(R_q \times R_q)$.

One of the fundamental differences between a construction based on R-LWE compared with one based on plain LWE is that working with a polynomial ring provides additional algebraic structure. This structure is exactly what is exploited to get a more efficient construction. The issue here is that the additional structure is something that a potential attacker can exploit. This highlights two possibly conflicting goals where on one hand, we want a small and efficient construction, but on the other hand, we also want to minimize structure. A trade-off between these which allows more flexibility would be to instead base an encryption scheme on the Module-LWE problem, which is a natural extension of the RLWE problem to vectors of polynomials, which we now explain in detail.

2.2.3 Module-LWE

Based on the R-LWE problem, we can further extend the construction into the M-LWE problem, first defined by Langlois et al. [LS15], which is the foundation of the presumably quantum-resistant cryptosystem Kyber, which is detailed in Section 2.3.

Definition 8 (Module-LWE distribution)**.** Let d and q be positive integers, and let χ be any distribution over R_q. Then for a fixed secret $\boldsymbol{s} \in R_q^d$, we define the M-LWE distribution over $R_q^d \times R_q$ by choosing $\boldsymbol{a} \leftarrow U(R_q^d)$, $e \leftarrow \chi$ and outputting $(\boldsymbol{a}, b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e \bmod q)$.

Definition 9 (Search Module-LWE)**.** Consider any number of positive samples $(\boldsymbol{a_i}, b_u) \in R_q^k \times R_q$ that are drawn from the M-LWE sample with a uniformly random secret $\boldsymbol{s} \in R_q^k$. The M-LWE search problem is then defined as finding the secret $\boldsymbol{s}$.

Definition 10 (Decision Module-LWE)**.** Consider any positive number of samples $(\boldsymbol{a_i}, b_i) \in R_q^k \times R_q$ that are either drawn from an M-LWE sample with a random secret $\boldsymbol{s}$ with probability distribution χ or from the uniform distribution $(\boldsymbol{a_i}, b_i) \leftarrow U(R_q^k \times R_q)$. Then the decision variant of the problem is defined as distinguishing the 2 cases.

By now, it becomes apparent that the encryption scheme from Oded Regev can be extended to the Module setting, such that we can encrypt polynomials instead of bits. Doing so allows us to encode messages as 256 degree polynomials, where each coefficient encodes a single bit. This allows encrypting messages of 32 bytes, which then enables us to encrypt 32-byte symmetric keys. This will be particularly relevant later on.

Size of the modulus q For a cryptosystem based on any of the LWE variants, we want the modulus to be small for efficiency and a smaller ciphertext size, but we still want the modulus to be large enough for the scheme to not become insecure. Therefore, we would ideally want the modulus to be of a polynomial size so as not to make the scheme impractical.

2.2.4 Learning With Rounding

Another variant of the LWE problem we will be using throughout the report is the LWR problem [CCMS21], which contrasts LWE by being deterministic due to relying on rounding instead of adding randomly sampled noise. The rounding can be done by simply dropping bits, which allows for a simpler construction than when having to sample and add noise. Additionally, dropping bits is more efficient than sampling from the distributions typically used for LWE. The downside here, as mentioned by the Kyber specification [Pet23], is that this adds algebraic structure, something that could potentially be exploited in an attack.

The search and decision problem can then be defined similarly to the problems for the LWE variants.

Definition 11 (Search LWR)**.** For any positive number of independent LWR samples $(\boldsymbol{a_i}, b_i = \lfloor \boldsymbol{A} \cdot \boldsymbol{s} \rceil_p) \in \mathbb{Z}_q^n \times \mathbb{Z}_p$ for a uniformly sampled secret $\boldsymbol{s} \in \mathbb{Z}_q^n$, the search problem asks to find the secret $\boldsymbol{s}$.

Definition 12 (Decision LWR)**.** Given any number of positive independent LWR samples either drawn from $(\boldsymbol{a_i}, b_i = \lfloor \boldsymbol{A} \cdot \boldsymbol{s} \rceil_p) \in \mathbb{Z}_q^n \times \mathbb{Z}_p$ or from the uniform distribution $(\boldsymbol{a_i}, b_i) \leftarrow U(\mathbb{Z}_q^n \times \mathbb{Z}_p)$, the decision variant asks to distinguish these cases with a non-negligible probability.

These problems also have natural extensions to polynomials and vectors of polynomials, just like for LWE.

2.3 Kyber

In the following section, we will briefly discuss the overall structure of the post-quantum Key-Encapsulation Mechanism (KEM) Kyber, which has recently been chosen by NIST as a part of the next Post-Quantum Cryptography standard. For a better understanding, one should take a look at the algorithms in the original Kyber specification [Pet23].

2.3.1 Instantiating Kyber

Kyber is instantiated using the following parameters

- **n**: Maximal degree of the polynomials.
- **k**: Number of polynomials in the vectors of the underlying MLWE problem.
- **q**: Modulus for all coefficients and numbers.
- $\boldsymbol{\eta_1, \eta_2}$: Says something about how large the "small coefficients," mostly used by the error vector, can be.
- $\mathbf{d_u}, \mathbf{d_v}$: Controls the amount of compression on the ciphertext (**u**, v).
- $\boldsymbol{\delta}$: Denotes the probability that the decryption fails.

The following three parameter sets are used in the standard Kyber specification. Note that while they provide a different amount of security, they are easily extendable with only minor changes between the parameter sets, as can further be seen in the TPKE parameter sets 3.1.4 shown later.

Kyber parameters							
name	n	k	q	η_1	η_2	(d_u, d_v)	δ
Kyber512	256	2	3329	3	2	(10,4)	2^{-139}
Kyber768	256	3	3329	2	2	(10,4)	2^{-164}
Kyber1024	256	4	3329	2	2	(11,5)	2^{-174}

Figure 2.2: Parameters in the original Kyber scheme

2.3.2 Symmetric primitives

For the sake of completeness, Kyber introduces two different instantiations for the symmetric primitives, as described below. One uses a more modern approach, for which new hardware is needed, and another where functions have been specifically handpicked to currently available hardware.

Modern

A modern instantiation of the functions is as follows. To instantiate the functions `PRF`, `XOF`, `H`, `G` and `KDF`, we use hash functions defined in the FIPS-202 standard [Div14]. All these functions can be obtained by a call to the Keccak function with appropriate arguments.

- `XOF` is instantiated with SHAKE-128
- `H` is instantiated with SHA3-256
- `G` is instantiated with SHA3-512
- `PRF(s, b)` is instantiated with SHAKE-256(s || b)
- `KDF` is instantiated with SHAKE-256

"90s" variant

The idea behind the choices in this "90s" variant is to choose already standardized functions by NIST, which have existing hardware implementations. These usually boil down to SHA-256 and AES.

- $\texttt{XOF}(p, i, j)$ is instantiated with AES-256 in CTR mode where p is used as the key and $i||j$ padded with zeros to become a 12-byte nonce. The counter should start at 0.
- `H` is instantiated with SHA-256
- `G` is instantiated with SHA-512
- $\texttt{PRF}(s, b)$ is instantiated with AES-256 in CTR mode where s is used as the key and b is padded to a 12-byte nonce. The counter should start at 0.
- `KDF` is instantiated with SHA-256

Note also that all the symmetric primitives for the modern variant could have been instantiated using only a single hash function, but that the authors of Kyber made an active choice to choose different functions from the FIPS-202 standard [Div14] to avoid having to prefix a character with the input to get domain separation.

2.3.3 Compression and decompression

Kyber defines two functions `Compress` and `Decompress` for compressing and decompressing values, effectively removing some of the low-order bits, which does not have much effect on the failure probability of the decryption, making ciphertexts smaller. `Compress` and `Decompress` are defined as follows

$$\texttt{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d$$
$$\texttt{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor \bmod 2^d$$

This compression is lossy, and if we define an element x' by

$$x' = \texttt{Decompress}_q(\texttt{Compress}_q(x, d), d)$$

then we say that x and x' are close iff

$$|x' - x \bmod q| \leq B_q = \lceil q/2^{d+1} \rfloor$$

While the compression does compress the ciphertexts, making them smaller, it also handles the LWE error correction when encrypting and decrypting. In the encryption function, described in algorithm 5, the decompress function creates error tolerance gaps by sending a message bit 0 to the number 0 and 1 to the number $q/2$, as we know is necessary from usual LWE. Compress will reverse this in the decryption function based on whether $v - \boldsymbol{s}^T \cdot \boldsymbol{u}$, taken from algorithm 6, is closer to $\lceil q/2 \rceil$ or 0, correctly decoding to the corresponding bit.

2.3.4 Number Theoretic Transformation

The Number Theoretic Transformation (NTT) can be applied to polynomials in R_q to make polynomial multiplication faster. Initially, polynomial multiplication in the ring can be done in $\mathcal{O}(n^2)$, as each coefficient of either polynomial have to be multiplied with each coefficient in the other polynomial.

$$c = a \cdot b = (a_0 + a_1 x + \dots) \cdot (b_0 + b_1 x + \dots) = a_0 \cdot b_0 + a_0 \cdot b_1 x + \dots$$

By utilizing the NTT transformation, we can instead use some cleverly constructed algorithms to perform polynomial multiplication with complexity $\mathcal{O}(n \cdot log(n))$.

The overall idea is to split polynomials into many polynomials of smaller degrees, which by the Chinese Remainder Theorem, can then be assembled back together, to get the original polynomial. By reducing polynomials into smaller degrees, they will have fewer coefficients and can thus be multiplied faster.

In Kyber, we can represent the modulus polynomial as

$$x^{256} + 1 = \prod_{i=0}^{127} x^2 + \zeta^{2i+1} = \prod_{i=0}^{127} x^2 + \zeta^{2 \cdot br_7(i)+1}$$

where $br_7(i)$ is unsigned 7-bit reverse for any integer $i \in \mathbb{N}$. The NTT of any polynomial $f \in R_q$ can then be represented by 128 polynomials as

$$(f \bmod x^2 + \zeta^{2 \cdot br_7(0)+1}, f \bmod x^2 + \zeta^{2 \cdot br_7(1)+1}, \ldots, f \bmod x^2 + \zeta^{2 \cdot br_7(127)+1})$$

Any polynomial $f \in R_q$ can then be defined by the following

$$\texttt{NTT}(f) = f_0 + f_1 x + f_2 x^2 + f_{255} x^{255}$$

where it holds that

$$f_{2i} = \sum_{j=0}^{127} f_{2j} \zeta^{(2 \cdot br_7(i)+1) \cdot j}$$
$$f_{2i+1} = \sum_{j=0}^{127} f_{2j+1} \zeta^{(2 \cdot br_7(i)+1) \cdot j}$$

Note here that we have defined a new polynomial with the same degree as the original polynomial but that this NTT version of the polynomial should not be seen as a polynomial in R_q. It's simply a construction to reuse the already present polynomial data structure, which is most relevant when implementing Kyber.

There exist no 512'th primitive roots for the specific choices of parameters in the Kyber scheme, and hence we cannot split the input polynomial into 256 polynomials. For the specific choices of Kyber, we have the root of unity $\zeta = 17$ as $x^{256} \equiv 1 \bmod 3329$ for $x = 17$.

By using this, we can now do multiplications on polynomials $f, g \in R_q$ by computing

$$\texttt{NTT}^{-1}(\texttt{NTT}(f) \cdot \texttt{NTT}(g))$$

which means that each coefficient can be calculated as

$$h_{2i} + h_{2i+1} x = (f_{2i} + f_{2i+1} x)(g_{2i} + g_{2i+1}) \bmod x^2 + \zeta^{2 \cdot br_7(i)+1}$$

This technique can easily be generalized to polynomial vectors as well by converting each polynomial in the vector to its corresponding NTT form, then applying the above transformation to each polynomial.

2.3.5 Montgomery domain

Let n be a positive integer and R, T be integers for which it holds that $R > n$, $gcd(n, R) = 1$ and $0 \leq T < n \cdot R$. Then the montgomery reduction of T modulo n with respect to R is defined by

$$T \cdot R^{-1} \bmod n$$

Note that in many implementations, it makes sense to choose the factor R as a multiple of 2 to make use of efficient bit shifting.

If we wish to calculate

$$c \equiv a \cdot b \bmod n$$

then instead of working on a and b directly, we can then draw them into the Montgomery domain by applying

$$\overline{a} = a \cdot R^{-1} \bmod n$$
$$\overline{b} = b \cdot R^{-1} \bmod n$$

Which uses a small overhead to convert a and b into $\overline{a}$ and $\overline{b}$, hoping that the speedup will make up for this at a later stage. To do addition and subtraction, we can do as usual

$$\begin{aligned}\overline{c} &= \overline{a} + \overline{b} \bmod n \\ &= (a \cdot R^{-1}) + (b \cdot R^{-1}) \bmod n \\ &= (a + b) \cdot R^{-1} \bmod n\end{aligned}$$

and similarly, we can do subtraction the trivial way.

Montgomery reduction

Whenever we have two numbers on the Montgomery form, we will use the following Montgomery reduction algorithm.

montgomery_reduce(a)	
1 :	$u \leftarrow a \cdot q^{-1}$
2 :	$t \leftarrow u \cdot q$
3 :	$t \leftarrow a - t$
4 :	Return $t >> 16 \bmod q$

Figure 2.3: Montgomery reduction

Barrett reduction

When doing multiple reductions, the Barret reduction algorithm seen in Figure 2.4 is advantageous to use in Kyber. The algorithm works best when $a < n^2$ and is in the sense of Kyber used only when recreating a polynomial from its NTT form.

2.3.6 Algorithms

In the following section, we will present the different algorithms included in the Kyber specification. For the full algorithms, see appendix A.

1. Parse: Kyber samples polynomials in R_q deterministically by using the `Parse` algorithm. `Parse` takes as input a byte stream $\mathcal{B} = b_1, b_2, \ldots, b_8$ and outputs an element $\hat{a} = \hat{a}_0 + \hat{a}_1 x + \hat{a}_2 x^2 + \cdots + \hat{a}_{255} x^{255}$ in R_q in its NTT form.

The intuition behind `Parse` is that it should output something statistically close to uniformly random in the ring R_q, given a byte stream that is statistically uniformly random.

barrett_reduction(a)

1 : $v \leftarrow ((1 >> 26) + q/2)/q$

2 : $t \leftarrow (v \cdot a) + (1 << 25) >> 26$

3 : $t \leftarrow t \cdot q$

4 : Return $a - t$

Figure 2.4: Barrett reduction

Note that by making `Parse` deterministic, it becomes possible to recreate the exact sequence of polynomials by giving the same byte input. This is a property that will be used when generating the matrix of polynomials $\boldsymbol{A}$.

2. CBD: Noise in Kyber is sampled from a centered binomial distribution (`CBD`) parametrized by η, which determines the size of the coefficients in the "small" polynomials sampled. When sampling a vector $f \in R_q$ from `CBD`, we mean that each individual coefficient will get sampled from `CBD`. Note that this function is very carefully chosen, such that we do not end up with too much noise in the underlying MLWE scheme and, as a consequence, is very specific to the chosen modulo q.

3. Decode / Encode: As the input and output types of Kyber are in bytes and byte streams, we need some kind of mechanism for converting byte streams to polynomials and vice versa. Note that in the original Kyber specification, there is no definition of how to encode polynomials, only how to decode, but we will describe both.

The `Encode` function of Kyber establishes a way of converting any polynomial into a series of $256 \cdot 12 = 3072$ bits or equivalently 384 bytes. It does so by encoding each coefficient of the polynomial in 12 bits and appending all 3072 bits together to form one large byte stream with each coefficient appended. A visual representation of this can be seen in Figure 2.5.

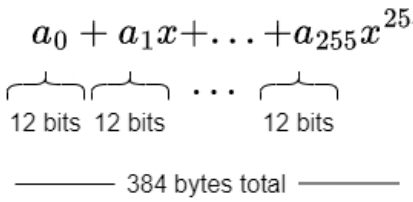

Figure 2.5: Visualisation of the Encode algorithm in Kyber

Trivially the `Decode` function takes this byte stream and converts it back into the individual coefficients by converting the chunks of 12 bits into each coefficient.

It's important to note that 12 bits are very consistently chosen with the standard Kyber modulus 3329 as $2^{11} < 3329 < 2^{12}$. To not lose any information, the number of bits can be no lower than 12 bits. For a larger modulus, one could just increase the number of bits to use for encoding/decoding in a trivial manner.

4. IND-CPA.KeyGen: To generate the keys, Kyber starts by using the `Parse` function to generate a matrix $A \in R_q^{k \times k}$ in which there are polynomials on the NTT form. The matrix A will be a square matrix of length k.

Afterward, the secret $\boldsymbol{s} \in R_q^k$ and error vector $\boldsymbol{e} \in R_q^k$ are chosen at random using the `CBD` function with parameter η_1. This results in the two vectors of polynomials with coefficients in the interval $[-\eta_1, \eta_1]$. So, again, the coefficients are small to ensure that there is not too much noise to be able to decrypt.

Using these, the public key and secret key are defined by the following

$$pk = (\mathbf{A}, \mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e})$$
$$sk = \mathbf{s}$$

The public key and the secret key can be converted into byte streams by using the `Encode` functionality.

Note that Kyber utilizes different symmetric primitives to regenerate seemingly random values from the randomly chosen coins. Inserting the same coins will yield the same output. This is especially used when calculating the Matrix A as A does not need to be sent, only the randomness used to generate A, which is much smaller.

5. IND-CPA.Encrypt: The Encrypt algorithm encrypts a binary stream $m \in \mathcal{B}^{32}$ by using the public key and some random coins. These coins are used such that IND-CPA encryption can be used deterministically.

To encrypt a byte stream $m \in \mathcal{B}^{32}$ the matrix $A \in R_q^{k \times k}$ is sampled using the public key, from which the randomness initially used to generate the matrix A lies.

Randomness $\boldsymbol{r} \in R_q^k$ is sampled from `CBD` with parameter η_1. While the error vectors $\boldsymbol{e}_1 \in R_q^k$ and $e_2 \in R_q$ are sampled using `CBD` with parameter η_2. Using all these values, one can now encrypt by calculating

$$\boldsymbol{u} = \mathbf{A}^T \mathbf{r} + \mathbf{e}_1$$
$$v = \mathbf{r}^T \mathbf{t} + e_2 + \lfloor q/2 \rceil \cdot m$$

then compressing and using the `Encode` to convert $(\boldsymbol{u}, v)$ into a byte stream, appending v on $\boldsymbol{u}$.

6. IND-CPA.Decrypt: Decrypting any ciphertext $(\boldsymbol{u}, v)$ into the message $m \in \mathcal{B}^{32}$, encrypted by the `Encrypt` functionality is done by using the secret keys $\boldsymbol{s}$ and calculating

$$m = v - \mathbf{u}^T \mathbf{s}$$

then compressing and encoding the result to get the correct byte stream.

7. KeyGen: Using a slightly tweaked Fujisaki-Okamoto transformation, Kyber transforms its IND-CPA scheme, as defined above, into one resistant against attacks in which the ciphertexts can be chosen, a so-called chosen ciphertext attack (CCA) secure scheme.

The algorithm samples some random bytes $z \in \mathcal{B}^{32}$, then uses the `IND-CPA.KeyGen` to receive public key `pk` and secret key `sk`. From this, the secret key is defined as

$$sk' = (sk||pk||H(pk)||z)$$

Where `H` is a hash function defined by the symmetric primitives.

8. Encapsulate: `Encapsulate` is used by a party to generate a key K and a ciphertext c, to send to the other party.

First, 32 random bytes $m \in \mathcal{B}^{32}$ are sampled, then m is updated to be $\mathtt{H}(m)$. Then the key $\overline{K}$ and randomness r are derived using `G` by computing $(\overline{K}, r) := G(m||H(pk))$. Using the randomness r, we utilize `IND-CPA.Encrypt` to encrypt m under the public key pk, which yields the ciphertext c. The key is then computed as $K = \mathtt{KDF}(\overline{K}||H(c))$. Finally, the output is the ciphertext and key (c, K), where the cipher can be sent to the recipient.

9. Decapsulate: When receiving the ciphertext c from `Encapsulate`, the recipient wishes to calculate the same key K as the sender has gotten from `Encapsulate`.

To do this, apply the `IND-CPA.Decrypt` function on c using the secret key `sk` to obtain m', which concatenated with `H` applied on the public key, can be used as input to function `G`, returning $(K', r') := G(m'||H(pk))$. Afterward, a check for the correct ciphertext c' is done by re-encrypting the found message m' using the public key pk and the randomness r' found earlier. If $c = c'$ then `Decapsulate` will return $K := \texttt{KDF}(K'||H(c))$ otherwise $K := \texttt{KDF}(z||H(c))$.

2.4 Rényi Divergence

When measuring the distance between two distributions, we can use the Rényi Divergence (RD) as an alternative to the Statistical Distance (SD). Here we present a definition by Boudgoust et al. [BS23].

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 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 distribution 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: Notice how the RD has multiplicative equivalents of the properties that the SD has. In the case of the probability preservation property, this means that for search problems, we may end up with a small RD where the statistical distance might have been large, making the RD especially applicable to these types of problems [BLL+15]. In the case of decision problems, however, we end up with the ideal winning probability increasing by a multiplicative factor due to the multiplicativity of the probability preservation property, which makes the RD less useful.

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

The following is a list of concrete applications, as proven by Bai et al. [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 preserves the dimension for a composite LWE modulus that is a multiple of the LWR rounding modulus without using noise flooding.

2.5 Threshold Decryption

In regular Public Key Encryption (PKE) schemes, a key pair consisting of a secret and a public key is generated, and the secret key is then used to decrypt ciphertexts encrypted under the public key. In Threshold Decryption, the secret key is instead split into multiple shares and distributed, such that a subset of $t+1$ shares is needed to be able to decrypt, depending on the specific secret sharing scheme. This setting is known as t-out-of-n threshold decryption. The setting where n shares are needed to be able to decrypt is called full-threshold decryption. The benefit of using threshold decryption is that we mitigate the issues relating to having a single point of failure, as in regular PKE schemes where one party holds the key.

A public key encryption scheme with threshold decryption is called a Threshold Public Key Encryption (TPKE) scheme. A way of achieving such a scheme is to generate a key pair, then use one of the techniques presented in Section 2.1 to secretly share the secret key. We now formally define TPKE schemes, then present notions of security for TPKE schemes.

2.5.1 Threshold Public Key Encryption

In this thesis, we will present a definition of TPKE schemes by Boudgoust et al. ([BS23]). Here, a TPKE scheme is defined to be a tuple of PPT algorithms TPKE = (Setup, Enc, PartDec, Combine), which are defined in the following manner:

Setup$(1^\lambda, n, t) \to (pp, pk, sk_1, ..., sk_n)$**:** 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$(pk, m) \to ct$**:** Given public key pk and message m this algorithm outputs a ciphertext ct.

PartDec$(sk_i, ct) \to d_i$**:** On input a key share sk_i for some $i \in \{1, ..., n\}$ and a ciphertext ct, the algorithm outputs a decryption share d_i.

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

In addition, the above algorithms must satisfy correctness, which for TPKE schemes is defined as follows:

Definition 13 (Decryption Correctness)**.** Let λ, n, t, S be the TPKE parameters, let $S \subset \{1, ..., n\}$, $|S| \geq t+1$, and $\text{negl}(\lambda)$ a negligible function. A TPKE scheme satisfies decryption correctness if for $(\text{pp}, \mathsf{pk}, \mathsf{sk}_1, ..., \mathsf{sk}_n) \leftarrow \text{Setup}(1^\lambda, n, t)$, $\text{ct} \leftarrow \text{Enc}(\mathsf{pk}, m)$, and decryption shares $d_i \leftarrow \text{PartDec}(\mathsf{sk}_i, ct)$ for $i \in S$, it holds that

$$\Pr[\text{Combine}(\{d_i\}_{i\in S}, \text{ct}) = m] = 1 - \text{negl}(\lambda)$$

2.5.2 Notions of security for TPKE schemes

Now we state some security notions 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 $c = \text{Enc}_{pk}(m)$ of a random message m, it is hard to find m.

For TPKE schemes, the adversary is also given access to the secret key shares of the corrupted parties, of which there are at most t. In addition to this, the adversary is also allowed to make ℓ adaptive queries to get partial decryption shares for all parties for message-ciphertext pairs. Thus the security notion becomes parametrized by ℓ and is thus called ℓ-OW-CPA security for TPKE schemes. We now state the formal definition as defined by Boudgoust et al. [BS23].

Definition 14 (ℓ**-OW-CPA for TPKE**)**.** A TPKE scheme is ℓ**-OW-CPA** secure for security parameter λ, n parties, threshold parameter t, and query bound ℓ, if for all PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

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

Where the experiment $\mathrm{Expt}^{\ell\textbf{-OW-CPA}}_{\mathcal{A},\mathrm{TPKE}}$ and OPartDec(m) are defined as in Figure 2.6.

$\mathrm{Expt}^{\ell\textbf{-OW-CPA}}_{\mathcal{A},\mathrm{TPKE}}$

1 : $(\mathrm{pp}, \mathsf{pk}, \mathsf{sk}_1, ..., \mathsf{sk}_n) \leftarrow \textsc{Setup}(1^\lambda, n, t)$
2 : $S \leftarrow \mathcal{A}_1(\mathrm{pp}, \mathrm{pk}) : S \subset [1, ..., n] \wedge |S| \leq t$
3 : $m \leftarrow U(\mathcal{M})$
4 : $ct \leftarrow \textsc{Enc}(\mathrm{pk}, m)$
5 : $m' \leftarrow \mathcal{A}_2^{\mathrm{OPartDec}}(\mathrm{pk}, \{\mathrm{sk}_i\}_{i \in S}, ct)$
6 : **Return** $m = m'$

Figure 2.6: Definition of the experiment $\mathrm{Expt}^{\ell\textbf{-OW-CPA}}_{\mathcal{A},\mathrm{TPKE}}$

OPartDec(m)

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

Figure 2.7: Definition of oblivious partial decryption, OPartDec

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.

Extending the IND-CPA security definition to TPKE schemes is done similarly to the extension of OW-CPA security. Namely, the adversary is given access to secret key shares of corrupted parties and gets access to ℓ partial decryption queries.

As with OW-CPA security, we now state the formal definition of ℓ-IND-CPA security for TPKE schemes as specified by Boudgoust et al. [BS23].

Definition 15 (ℓ**-IND-CPA for TPKE**)**.** A TPKE scheme is ℓ**-IND-CPA** secure for security parameter λ, n parties, the threshold t, and query bound ℓ, if for all PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4)$

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

Where $\mathrm{Expt}^{\ell\textbf{-IND-CPA}}_{\mathcal{A},\mathrm{TPKE}}$ is the experiment in Figure 2.8.
With OPartDec(m) defined as in section 2.5.2.

$\text{Expt}^{\ell\text{-IND-CPA}}_{\mathcal{A},\text{TPKE}}$

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 : $state \leftarrow \mathcal{A}_2^{\text{OPartDec}}(\text{pk}, \{\text{sk}_i\}, ct)$

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 $b = b'$

Figure 2.8: Definition of the experiment $\text{Expt}^{\ell\text{-IND-CPA}}_{\mathcal{A},\text{TPKE}}$

2.5.3 Robustness

The article by Boudgoust et al. [BS23] also defines two different variants of an additional property that TPKE schemes can have called robustness. The first variant is called weak chosen-ciphertext robustness, while the second variant is named strong chosen-plaintext robustness.

Weak Chosen-Ciphertext Robustness In this version of the robustness property, we imagine an experiment in which the adversary is given all secret key shares. For the TPKE scheme to have this variant of robustness, it must then be hard for the adversary to find a ciphertext and two different sets of partial decryption shares that combine two different messages. What this means is that if the TPKE has this property, then at most the corrupted players can make decryption fail. What they cannot do, however, is to make the decryption output a different message from the one initially encrypted, except for a message indicating that the decryption failed.

Strong Chosen-Plaintext Robustness For this variant, the adversary is given an honestly formed ciphertext and the secret key shares of the corrupted parties. It must then be hard for the adversary to forge partial decryption shares, such that when combined with the honest partial decryption shares, they combine to a different message than the message that was initially encrypted. The interesting aspect of this is that if a TPKE has this property, then even when some parties are corrupt, it is still possible to decrypt.

From Weak Chosen-Ciphertext Robustness to Strong Chosen-Plaintext Robustness If we have less than $n/2$ corruptions, then we can transform any scheme with Weak Chosen-Ciphertext Robustness into on that is Strongly Chosen-Plaintext Robust. This is done simply by letting $\textsc{Combine}$ try all $t + 1$ size subsets of partial decryption shares. Since we have $t < n/2$ corruptions, there will be a subset of $t + 1$ honest partial decryption shares, allowing decryption even in the presence of corrupted players.

Section 3

Threshold Decryption Constructions

3.1 TKyber: Threshold Encryption From LWE With Polynomial Modulus

Er det en konvention at det er super polynomielt andre steder? In a paper by Boudgoust et al. [BS23], the authors describe how to construct an IND-CPA secure TPKE scheme based on LWE, where the modulus q does not have to be superpolynomial in size. As mentioned previously in section 2.4, using the Rényi Divergence (RD) has a lot of useful applications in lattice cryptography, and in this case, the RD is what allows the polynomial modulus.

As also mentioned earlier, the RD seems to work especially well for search problems due to the multiplicativity of its properties, whereas it seems to be less useful for distinguishability problems. To take advantage of this, the scheme itself is achieved by first constructing an OW-CPA secure TPKE scheme instantiated by a PKE scheme, then using a transformation on that scheme to attain an IND-CPA secure TPKE scheme.

3.1.1 Nearly linear decryption for PKE schemes

We say that any PKE system has nearly linear decryption if the decryption can be seen as a linear function of the secret key that outputs a noisy version of the encrypted message. This leads to the following definition of (β, ϵ)-linear decryption:

Definition 16 ((β, ϵ)-Linear Decryption for PKE schemes)**.** We are given a PKE scheme with algorithms (KeyGen, Enc, Dec), message space $\mathcal{M} \subseteq R_p$, and ciphertext space $\mathcal{C} = R_q^r$. We say that such a PKE scheme has (β, ϵ)-linear decryption iff for $\beta = \beta(\lambda) \in \mathbb{N}, \epsilon = \epsilon(\lambda) \in [0, 1]$, key pair $(\mathsf{pk}, \mathsf{sk})$, and ciphertext ct it holds that

$$\langle \mathsf{sk}, \mathrm{ct} \rangle = \lfloor q/p \cdot m \rceil + e \mod q$$

for some error term $e \in R_q$, such that $\Pr[||e||_\infty \leq \beta] \geq 1 - \epsilon$ and probability is taken over randomness used for the algorithms of the encryption scheme.

This property works well with LSS schemes, described in section 2.1, as each party can then effectively operate on the ciphertext using their own share of the secret key. This can be used to decrypt locally, which results in a partial decryption. Upon choosing to decrypt, the parties can then combine their partial decryptions, and due to the (β, ϵ)-linear decryption property, this will achieve a decrypted but noisy version of the encrypted message.

Kyber has Nearly Linear Decryption We now show that Kyber satisfies this particular requirement, which we will use later. To show this, we can view the secret key as $\mathsf{sk} = (-\mathbf{s}, 1)^T$ and the ciphertext as $\mathrm{ct} = (v, \mathbf{u})$, then we can describe decryption as computing $\langle \mathsf{sk}, \mathrm{ct} \rangle = v \cdot 1 - \mathbf{s}^T \mathbf{u}$, then scaling and rounding the result. Now, by the definition of Kyber

encryption, we can rewrite the inner product to get

$$\begin{aligned}
\langle \mathsf{sk}, \mathrm{ct} \rangle &= v \cdot 1 - \mathbf{s}^T \mathbf{u} \\
&= \mathbf{r}^T \mathbf{e} - \mathbf{e}_1^T \mathbf{s} + e_2 + \lfloor q/p \rceil \cdot m \\
&= \mathbf{r}^T \mathbf{e} - \mathbf{e}_1^T \mathbf{s} + e_2 + \lfloor q/p \cdot m \rceil \qquad \text{(Since m is a binary polynomial)} \\
&= \lfloor q/p \cdot m \rceil + e_{\text{noise}} \mod q \qquad \text{(For } e_{\text{noise}} = \mathbf{r}^T \mathbf{e} - \mathbf{e}_1^T \mathbf{s} + e_2\text{)}
\end{aligned}$$

Where $||e_{\text{noise}}||_\infty$ depends on the η_1, η_2 parameters used for sampling from CBD in Kyber.

3.1.2 OW-CPA secure TPKE

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

- 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 the Rényi Distance is $\text{RD}_a(\mathcal{D}_{\text{sim}} || \mathcal{D}_{\text{flood}} + B) \leq \epsilon_{\text{RD}_a}$, for $a \in (1, \infty), \epsilon_{\text{RD}_a} > 1$, and for all B with $|B| \leq \beta_{\text{pke}}$.
- 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 $(\textsc{KeyGen}, \textsc{Enc}, \textsc{Dec})$, with $\mathcal{M} \subseteq R_p$, $\mathcal{C} = R_q^r$, and $(\beta_{\text{pke}}, \epsilon)$-linear decryption for some $\beta_{\text{pke}} < q/(2p) - \tau_{\min}\ \beta_{\text{flood}}$ and negligible ϵ.

Constructing the scheme is then done by defining the Enc algorithm as in the PKE scheme, while Setup, PartDec, and Combine are as described in figure 3.1.

Setup(1^λ, n, t)	
1 :	$(\mathsf{pk}, \mathsf{sk}) \leftarrow \textsc{KeyGen}(1^\lambda)$
2 :	$(\mathsf{sk}_1, ..., \mathsf{sk}_n) \leftarrow \text{LSS.Share}(\mathsf{sk})$
3 :	Return $(\mathsf{pk}, \mathsf{sk}_1, ..., \mathsf{sk}_n)$

PartDec(m)	
1 :	$e_{i,j} \leftarrow \mathcal{D}_{\text{flood},R_q}$ for $j \in [0, ..., L]$
2 :	$d_{i,j} \leftarrow \langle ct, \mathsf{sk}_{i,j} \rangle + e_{i,j}$
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 3.1: Setup, PartDec, and Combine algorithms for the OW-CPA secure TPKE.

Security of the construction

For the security of the above construction, we present the following theorem, and accompanying proof adapted from Boudgoust et al. [BS23]. The adaptations made to the proof consist of simplifying it to only concern TPKE and PKE schemes, as opposed to Threshold Fully Homomorphic Encryption (TFHE) and Fully Homomorphic Encryption (FHE) schemes.

Theorem 1 (OW-CPA security of TPKE)**.** Let ℓ be the partial decryption query bound. If the underlying PKE is OW-CPA secure, then the TPKE is ℓ-OW-CPA secure.

Proof. The theorem is proven by showing that for any adversary $\mathcal{A}$ against the ℓ-OW-CPA TPKE game, we have an adversary $\mathcal{B}$ against the OW-CPA security of the PKE used with advantage

$$\mathrm{Adv}^{\ell-\text{OW-CPA}}_{\text{TPKE}}(\mathcal{A}) \leq \left(\mathrm{Adv}^{G_3}_{\text{TPKE}}(\mathcal{A}) \cdot \epsilon_{\text{RD}_a}^{\ell d(nL-\tau_{\max})}\right)^{(a-1)/a}) + \ell\epsilon$$

To prove this, the authors define five games.

G_0: The first game G_0 is defined exactly as the original TPKE ℓ-OW-CPA game.

G_1: In the game G_1 we consider the set $S_L = \{(i,j)\}_{i\in S, j\in\{1,\ldots,L\}}$ where S is the size-t set of corrupted parties chosen by the adversary. We fix $T \subseteq S_L$ to be the smallest set such that the corresponding shares reveal no information about the underlying secret. For all $(i,j) \in T$ we compute the corresponding partial decryptions as in G_0:

$$\widetilde{\mathbf{d}}_{i,j} = \langle ct, \mathrm{sk}_{i,j}\rangle$$

Fix $T_{i,j} \subseteq T \cup \{(i,j)\}$ to be the smallest set that allows reconstruction of the secret. Now compute

$$\widetilde{\mathbf{d}}_{i,j} = \langle ct, \mathrm{sk}\rangle - \sum_{(k,l)\in T_{i,j}\setminus\{(i,j)\}} \widetilde{\mathbf{d}}_{k,l}$$

Lastly, sample noise $\mathbf{e}_{i,j} \leftarrow \mathcal{D}_{\text{flood},R_q}$ and set $\mathbf{d}_{i,j} = \widetilde{\mathbf{d}}_{i,j} + \mathbf{e}_{i,j}$ for $i \in \{0,\ldots,n\}$.

G_2: In G_2, a check is made to ensure that a ciphertext is a valid encryption of m with bounded noise before outputting a partial decryption for that particular ciphertext. To do this, we check

$$\langle ct, sk\rangle = \lfloor q/p\rceil \cdot m + e$$

for noise e satisfying that the infinity norm is $||e||_\infty \leq \beta_{\text{pke}}$. If this check fails, the game aborts.

G_3: In G_3 simulate partial decryptions outside T by computing

$$\widetilde{\mathbf{d}}_{i,j} = \lfloor q/p \cdot m\rceil - \sum_{(k,l)\in T_{i,j}\setminus\{(i,j)\}} \widetilde{\mathbf{d}}_{k,l}$$

In addition to this we also instead sample $\mathbf{e}_{i,j} \leftarrow \mathcal{D}_{\text{sim}}, R_q$ iff $(i,j) \notin T$.

G_4: In game G_4, we simulate the secret key shares seen by the adversary in G_3. This is done by sampling n secret key shares using LSS.share(0), then distributing the shares corresponding to the corrupted parties to the adversary.

Now we can analyze the games. The advantage of the adversary in G_0 is $\mathrm{Adv}^{\ell-\text{OW-CPA}}_{\text{TPKE}}(\mathcal{A})$, since this is just the original ℓ-OW-CPA game.

By the strong $\{0,1\}$-reconstruction property of the LSS used, the views of the adversary in the games G_0 and G_1 are indistinguishable. In other words, the adversary has the same advantage in G_1 as in G_0.

The probability that the game aborts in the check introduced in G_2 is ϵ, by the $(\beta_{\text{pke}}, \epsilon)$-Linear Decryption property for PKE schemes stated in Definition 16. By a union bound over all ℓ queries, the probability of aborting at this step is $\ell\epsilon$. So the advantage of the adversary in this game satisfies $\mathrm{Adv}^{G_1}_{\text{TPKE}}(\mathcal{A}) \leq \mathrm{Adv}^{G_2}_{\text{TPKE}}(\mathcal{A}) + \ell\epsilon$.

By Lemma 5.3 from the original paper, we know that for G_3, it holds that $\mathrm{Adv}^{G_2}_{\mathrm{TPKE}}(\mathcal{A}) \leq (\mathrm{Adv}^{G_3}_{\mathrm{TPKE}}(\mathcal{A}) \cdot \epsilon^{\ell d(nL-\tau_{\max})}_{\mathrm{RD}_a})^{(a-1)/a}$. The lemma focuses on TFHE, but TPKE can be seen as a special case of TFHE, so the lemma still holds.

By the privacy preservation property of LSS schemes presented in Section 2.1, the views of the adversary in games G_3 and G_4 are perfectly indistinguishable.

To finalize the proof, we note that the success probability of the adversary against G_4 is the same as the success probability of the adversary against the standard OW-CPA PKE game. This is because the only difference between OW-CPA security for PKE and TPKE schemes is that the adversary is given access to secret key shares of corrupted parties and ℓ partial decryption queries. In G_4, the secret key shares are simulated, and the partial decryptions no longer depend on sk, so the adversary would be able to simulate these by sampling random values as well. Thus, if any adversary has a non-negligible advantage against the OW-CPA security of the underlying PKE scheme, then the advantage of any adversary against the ℓ-OW-CPA TPKE game would also be non-negligible. □

3.1.3 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 $G : \mathcal{M}^\delta \to \{0,1\}^{2\lambda}$.

To generate an Indistinguishably secure scheme from the One-Wayness secure scheme, the same Setup function is used - returning the public key pk and each of the n secret key shares. When encrypting a value, we will sample δ values $\mathbf{x} = (x_1, \ldots, x_\delta)$ uniformly from the message space. The first value in the ciphertext is set to be the message summed with hash function F applied on $\mathbf{x}$, and $c_{\delta+1}$ set to be hash function G applied on $\mathbf{x}$, used for verification of the $\mathbf{x}$ values. By doing verification of the $\mathbf{x}$ values in this manner, we also obtain Weak Chosen-Ciphertext Robustness. This allows any party to compute locally on each of the δ ciphertexts using their own share of the secret key using the PartDec subroutine as defined in Figure 3.1. Lastly, all of the parties can recombine the message by using the Combine subroutine, where they will first compute x'_j for $j = 1, \ldots, \delta$, then verify with hash function G that the values are actually equal to $\mathbf{x}$ before applying hash function F and subtracting this from c_0, receiving the encrypted message m.

Formally speaking, given any OW-CPA secure TPKE = (Setup, Enc, PartDec, Combine) the IND-CPA secure TPKE$'$ = (Setup$'$, Enc$'$, PartDec$'$, Combine$'$) can be 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 $c_j = \mathrm{Enc}(\mathrm{pk}, x_j)$ for $j \in \{1, ..., \delta\}$. Output $ct := (c_0, ..., c_{\delta+1})$.

PartDec$'(sk_i, ct)$: On input (sk_i, ct) for $i \in \{1, ..., n\}$, compute $d_{ij} \leftarrow \mathrm{PartDec}(sk_i, c_j)$ for $j \in \{1, ..., \delta\}$ and output $\mathbf{d}_i := (d_{ij})_{j \in \{1,...,\delta\}}$.

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

Security of the transformation

Again we present a theorem from Boudgoust et al. [BS23] for the security of the above transformation.

Theorem 2 (Security of TPKE′)**.** Let δ be the security loss parameter described above, and ℓ be the query bound. If the TPKE is $(\ell\delta)$-OW-CPA secure, then TPKE′ is ℓ-IND-CPA secure in the Random Oracle Model (ROM).

Proof. The theorem is proven by constructing an $(\ell\delta)$-OW-CPA adversary $\mathcal{B}$ with advantage

$$\mathrm{Adv}^{\ell-\text{IND-CPA}}_{\text{TPKE}'}(\mathcal{A}) \leq q_F^{1/\delta} \cdot \mathrm{Adv}^{(\ell\delta)-\text{IND-CPA}}_{\text{TPKE}}(\mathcal{B})$$

from any ℓ-IND-CPA adversary $\mathcal{A}$ with advantage $\mathrm{Adv}^{\ell-\text{IND-CPA}}_{\text{TPKE}'}(\mathcal{A})$, where q_F is a bound on the number of queries to the oracle F.

To construct the adversary, we define two games, G_0 and G_1. The first game, G_0, is defined exactly as the ℓ-IND-CPA game where the encryption operation Enc′ is used on line 6 to generate the challenge ciphertext. The only difference between G_0 and G_1 is that in the second game, G_1, we initialize a boolean variable "flag" to be false. If F is queried on the vector $\mathbf{x}^*$ sampled for the challenge ciphertext, then F sets "flag" to true and aborts.

Now we can analyze the two games. Since G_0 is just the ℓ-IND-CPA game we know that the advantage of $\mathcal{A}$ is

$$\mathrm{Adv}^{\ell-\text{IND-CPA}}_{\text{TPKE}'}(\mathcal{A}) = |\Pr[G_0(\mathcal{A}) = 1] - 1/2|$$

Because the only change between G_0 and G_1 is that F aborts if queried on $\mathbf{x}^*$, it must be the case that

$$|\Pr[G_0(\mathcal{A}) = 1] - \Pr[G_1(\mathcal{A}) = 1]| \leq \Pr[\text{flag}]$$

Now, because F aborts in this case, the view of $\mathcal{A}$ is also independent of the bit b from the game, so $\Pr[G_1(\mathcal{A}) = 1] = 1/2$. Summarising the above, we get

$$\begin{aligned}\mathrm{Adv}^{\ell-\text{IND-CPA}}_{\text{TPKE}'}(\mathcal{A}) &= |\Pr[G_0(\mathcal{A}) = 1] - 1/2| \\ &= |\Pr[G_0(\mathcal{A}) = 1] - \Pr[G_1(\mathcal{A}) = 1]| \\ &\leq \Pr[\text{flag}]\end{aligned}$$

Now we need to bound the probability Pr[flag]. This is done by letting $\mathcal{B}$ use $\mathcal{A}$ as a subroutine in its $(\ell\delta)$-OW-CPA game. Here $\mathcal{B}$ simulates the random oracles F and G as well as the decryption queries to OPartDec′. The queries to OPartDec′ are done by $\mathcal{B}$ querying its own oracle OPartDec δ times, leading to the security loss of the reduction. □

3.1.4 Parameters

To instantiate the construction, we need a PKE scheme, as mentioned previously. If we instantiate it using the PKE scheme Kyber [Pet23], then this yields TKyber. In relation to this, the authors of the construction introduce three new Kyber parameter sets by incrementing the number of polynomials k in the Kyber scheme. This results in the following new Kyber parameter sets.

New Kyber parameter sets				
name	n	k	q	η
Kyber1280	256	5	3329	2
Kyber1536	256	6	3329	2
Kyber1792	256	7	3329	2

For the complete parameter sets, the authors of [BS23] provide a number of parameter sets derived in a simplified way based on Theorem 1. These can be seen in the table in Table 3.1.

Set	$(\beta_{\text{pke}}, \varepsilon)$	n	t	ℓ	$\lceil \log_2 \sigma \rceil$	$\lceil \log_2 q \rceil$	λ_{PKE}	λ_{TPKE}	Δ_λ
TKyber1024	$(390, 2^{-60})$	2	1	1	17	23	120	117	3
TKyber1024	$(934, 2^{-300})$	2	1	1	18	24	111	108	3
TKyber1024	$(390, 2^{-60})$	10	9	1	17	25	105	102	3
TKyber1280	$(435, 2^{-60})$	10	5	1	21	29	120	117	3
TKyber1536	$(476, 2^{-60})$	20	10	10	27	36	112	109	3
TKyber1792	$(513, 2^{-60})$	2	1	2^{32}	33	39	123	120	3

Table 3.1: Parameter sets derived in a generic manner

The authors also provide a number of additional parameter sets based on a Python script, which is an extension of the security estimation script accompanying Kyber [LD21]. These can be seen in Table 3.2.

Set	q	n	t	ℓ	$\mathcal{D}_{\text{flood}}$	$\mathcal{D}_{\text{sim}}$	λ_{TPKE}	Δ_λ
TKyber1024	$5 \cdot 3329$	2	1	1	947	1087	100	111
TKyber1024	$10 \cdot 3329$	2	1	2	1994	2034	104	91
TKyber1024	$9 \cdot 3329$	3	2	1	1197	1297	106	92

Table 3.2: Parameter sets derived using the Python script

3.2 Gladius: Hybrid encryption scheme with distributed decryption

Gladius [CCMS21] is a framework constructed with the aim of allowing distributed decryption for a hybrid post-quantum encryption scheme. The scheme is built by constructing an 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 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 for Gladius using Gladius-Hispaniensis, which uses the Hybrid_1 transformation.

The distributed decryption algorithm is then run inside a Multi-Party Computation (MPC) framework, such that the security follows immediately from the security of the underlying MPC framework.

The main goal of studying Gladius is to apply some of the techniques used for distributed decryption to achieve threshold decryption for Kyber. In addition to this, we also want to compare the Gladius protocol to one based on the Kyber KEM. In Section 4.1, we describe how we constructed a distributed decryption protocol for Kyber, while in Section 5.2, we detail our implementation using a concrete MPC framework. Section 5.3 describes how we implemented distributed decryption for Gladius-Hispaniensis.

3.2.1 Hybrid_1 and Hybrid_2 constructions

The Hybrid_1 and Hybrid_2 constructions are based on the basic Cramer-Shoup construction. Both of the constructions are parametrized by an encryption scheme Π_p, which is rigid. The rigidity requirement and its implications are discussed in Section 4.1.3.

Hybrid_1

Let an OW-CPA secure, rigid, and deterministic PKE be defined by the algorithms $\Pi_p = (\mathcal{K}_p, \mathcal{E}_p, \mathcal{D}_p)$ using the message space $\mathcal{M}_p = \{0,1\}^*$ and ciphertext space $\mathcal{C}_p \in \{0,1\}^*$. Let $\Pi_s = (\mathcal{K}_s, \mathcal{E}_s, \mathcal{D}_s)$ be a one-time IND-CPA secure symmetric encryption scheme with message space $\mathcal{M}_s = \{0,1\}^*$, ciphertext space $\mathcal{C}_s \subset \{0,1\}^*$, and key space $\mathcal{K}_s$.

The Hybrid_1 construction uses two hash functions G and H defined by Figure 3.2, where $\mathcal{R}_s$ denotes the space of random coins.

$$
\begin{aligned}
&H : \mathcal{M}_p \to \mathcal{K}_s, \\
&G : \begin{cases} \{0,1\}^* \times \mathcal{M}_p \to \{0,1\}^{|G|} & \text{if } \Pi_p \text{ is deterministic} \\ \mathcal{C}_p \times \{0,1\}^* \times \mathcal{M}_p \to \{0,1\}^{|G|} & \text{if } \Pi_p \text{ is randomized} \end{cases}
\end{aligned}
$$

Figure 3.2: Hash functions for Hybrid_1

Using these, the transformation in Figure 3.3 is constructed. The security of the construction relies on Theorem 3.1 of the Gladius paper.

Transformation Hybrid_1

$\mathcal{K}_h(1^t)$:	$\mathcal{E}_h(\mathsf{pk}, m)$:	$\mathcal{D}_h(\mathsf{sk}, (d_1, d_2, d_3))$:
1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathcal{K}_p(1^t)$	1 : $k \leftarrow \mathcal{M}_p$	1 : $k \leftarrow \mathcal{D}_p(\mathsf{sk}, c_1)$
2 : Return $(\mathsf{pk}, \mathsf{sk})$	2 : $\mathbf{k} \leftarrow H(k)$	2 : If $k = \perp$ then return $(\perp, \perp)$
	3 : $r \leftarrow \mathcal{R}_s$	3 : $t \leftarrow G(c_2, k)$
	4 : $c_1 \leftarrow \mathcal{E}_p(\mathsf{pk}, k)$	(resp. $c_3 \leftarrow G(c_1, c_2, k)$)
	5 : $c_2 \leftarrow \mathcal{E}_s(\mathbf{k}, m; r)$	4 : If $t \neq c_3$, then return $(\perp, \perp)$
	6 : $c_3 \leftarrow G(c_2, k)$	5 : $\mathbf{k} \leftarrow H(k)$
	(resp. $c_3 \leftarrow G(c_1, c_2, k)$)	6 : $m \leftarrow \mathcal{D}_s(\mathbf{k}, c_2)$
	7 : Return (c_1, c_2, c_3)	7 : Return (k, m)

Figure 3.3: Hybrid_1 Transformation

Hybrid_2

Hybrid_2 is very similar to that of Hybrid_1, but instead utilizes the hash functions H, H', H'', and G as described in figure 3.4. This hybrid transformation is parametrized by a rigid,

$$\begin{aligned} H &: \mathcal{M}_p \to \mathcal{K}_s, \\ H', H'' &: \mathcal{M}_p \to \mathcal{M}_p, \\ G &: \{0,1\}^* \times \mathcal{M}_p \to \{0,1\}^{|G|} \end{aligned}$$

Figure 3.4: Hash functions for hybrid 2

deterministic, OW-CPA secure encryption scheme Π_p. This transformation then results in a scheme that is secure in the Quantum Random Oracle Model (QROM), as shown in Theorem 3.2 of the Gladius paper. The full construction can be seen in Figure 3.5.

3.2.2 Gladius-Hispaniensis

The encryption scheme Π_p used by the Gladius-Hispaniensis variant of Gladius is based on regular LWR. This particular variant is parametrized 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. Now we are ready to describe the 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 $\text{pk} = (A_1, A_2)$, $\text{sk} = (\text{pk}, R_1)$.

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

Transformation Hybrid_2

$\mathcal{K}_h(1^t)$:	$\mathcal{E}_h(\mathsf{pk}, m)$:	$\mathcal{D}_h(\mathsf{sk}, (d_1, d_2, d_3))$:
1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathcal{K}_p(1^t)$	1 : $k \leftarrow \mathcal{M}_p$	1 : $k \leftarrow \mathcal{D}_p(\mathsf{sk}, c_1)$
2 : Return $(\mathsf{pk}, \mathsf{sk})$	2 : $\mathbf{k} \leftarrow H(k)$	2 : If $k = \perp$ then return $(\perp, \perp)$
	3 : $\mu \leftarrow H'(k)$	3 : $t \leftarrow H''(k)$
	4 : $r \leftarrow \mathcal{R}_s$	4 : If $t \neq c_4$, then return $(\perp, \perp)$
	5 : $c_1 \leftarrow \mathcal{E}_p(\mathsf{pk}, k)$	5 : $\mu \leftarrow H'(k)$
	6 : $c_2 \leftarrow \mathcal{E}_s(\mathbf{k}, m; r)$	6 : $t' \leftarrow G(c_2, \mu)$
	7 : $c_3 \leftarrow G(c_2, \mu)$	7 : If $t' \neq c_3$, then return $(\perp, \perp)$
	8 : $c_4 \leftarrow H''(k)$	8 : $\mathbf{k} \leftarrow H(k)$
	9 : Return (c_1, c_2, c_3, c_4)	9 : $m \leftarrow \mathcal{D}_s(\mathbf{k}, c_2)$
		10 : Return (k, m)

Figure 3.5: Hybrid_2 Transformation

Dec$(\text{sk}, (\mathbf{c}_1, \mathbf{c}_2))$: 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 $\mathbf{m}^T$.

Parameter sets

For the purpose of benchmarking and analysis later, we here name some of the Gladius-Hispaniensis parameter sets presented in the original Gladius paper. These can be seen in Table 3.3, while their security estimates can be seen in Table 3.4.

Set	n	t	q	p	ℓ	σ	μ
Gladius-Hisp. 1	971	2	$2^{21} - 9$	2^9	2^{19}	$\sqrt{1/2}$	128
Gladius-Hisp. 2	1024	2	$2^{21} - 9$	2^9	2^{19}	$\sqrt{1/2}$	128

Table 3.3: Parameter sets for Gladius-Hispaniensis from the Gladius paper.

Set	LWE Security	LWR Security Theoretical	LWR Security Best-Attack
Gladius-Hisp. 1	$2^{128.3}$	$2^{61.25}$	$2^{465.7}$
Gladius-Hisp. 2	$2^{135.7}$	$2^{64.78}$	$2^{492.7}$

Table 3.4: Security of parameter sets for Gladius-Hispaniensis.

3.2.3 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 3.6.

Protocol for Distributed Key Generation for Gladius Π_{DKeyGen}

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

Figure 3.6: A protocol for distributed key generation for Gladius-Hispaniensis

3.2.4 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. Here we use the notation $\langle a \rangle$ to denote a secret shared value.

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 $\text{BitAdd}(\langle \overline{\mathbf{a}} \rangle, \mathbf{1})$, 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 BitAdd, BitNeg, and BitLT. The protocol for doing this looks as can be seen in Figure 3.7.

Distributed decryption protocol

The full distributed decryption protocol, which uses the operations explained above, can then be seen in Figure 3.8

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.

Subroutine Centre($\langle x \rangle$)

1 : $\langle \mathbf{b} \rangle \leftarrow \text{BitDecomp}(\langle x \rangle)$

2 : $\langle \mathbf{b}' \rangle \leftarrow \text{BitAdd}(\langle \overline{\mathbf{b}} \rangle, q+1)$ \\ Computes $b' = q - u_i$ over integers.

3 : $\langle \mathbf{b}'' \rangle \leftarrow \text{BitNeg}(\langle \mathbf{b}' \rangle, q+1)$

4 : $\langle \mathbf{f} \rangle \leftarrow \text{BitLT}(\langle \mathbf{b} \rangle, q/2)$

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 3.7: The Centre subroutine used for DDec

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.

Protocol for Distributed Decryption using Gladius, $\Pi_{\text{DDecGladius}}$

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

KEM Decapsulation

$\langle x \rangle \leftarrow \mathbf{c}_2 - \mathbf{c}_1 \cdot \langle R_1 \rangle$

for $i \in [1, ..., n]$ **do**

  $\langle \mathbf{w} \rangle \leftarrow \text{Centre}(\langle x_i \rangle)$

  $\langle k_i \rangle \leftarrow \langle w_i^{(\nu)} \rangle \oplus \langle w_i^{(\nu+1)} \rangle = \langle w_i^{(\nu)} \rangle + \langle w_i^{(\nu+1)} \rangle - 2 \cdot \langle w_i^{(\nu)} \rangle \cdot \langle w_i^{(\nu+1)} \rangle$

KEM Validity Check

$\langle \mathbf{y} \rangle \leftarrow \langle \mathbf{k} \rangle \cdot (A_1 || A_2)$

$\langle z \rangle \leftarrow 1$

for $i \in [1, ..., 2 \cdot n]$ **do**

$\langle u \rangle \leftarrow \text{BitDecomp}(\langle y_i \rangle)$

$\langle b \rangle \leftarrow 1 - \text{BitLT}(\mathbf{u}, q/2)$

$\langle v \rangle \leftarrow \langle b \rangle \cdot \text{BitLT}(\langle \mathbf{u} \rangle, q \cdot (p+1)/(2 \cdot p))$

$\langle \mathbf{u'} \rangle \leftarrow \langle \mathbf{u} \rangle \ll \pi$

$\langle \mathbf{w} \rangle \leftarrow \text{BitAdd}(\langle \mathbf{u'} \rangle, 2^{\lceil \log_2 q \rceil + \pi} - c_i \cdot q)$

$\langle \mathbf{f} \rangle \leftarrow \text{BitAdd}(\langle \mathbf{w} \rangle, (\langle b \rangle - \langle v \rangle \cdot (-p \cdot q))$

$\langle \mathbf{f'} \rangle \leftarrow \text{BitNeg}(\langle \mathbf{f} \rangle)$

$\langle g \rangle \leftarrow \langle f_{\pi + \lceil \log_2 q \rceil - 1} \rangle$

$\langle s \rangle \leftarrow \langle g \rangle \cdot \langle \mathbf{f'} \rangle + (1 - \langle g \rangle) \cdot \langle \mathbf{f} \rangle$

$\langle j \rangle \leftarrow \text{BitLT}(\langle s \rangle, q/2))$

$\langle z \rangle \leftarrow \langle z \rangle \cdot \langle j \rangle$

$z \leftarrow \textbf{Output}(\langle z \rangle)$

If $z \neq 1$ then return $\perp$

Hash Check

$\langle t \rangle \leftarrow G(c_2, \langle \mathbf{k} \rangle)$

$t \leftarrow \textbf{Output}(\langle t \rangle)$

If $t \neq c_3$, return $\perp$

Message Extraction

$k \leftarrow \textbf{Output}(\langle \mathbf{k} \rangle)$

$\mathbf{k} \leftarrow H(k)$

$m \leftarrow D_S(\mathbf{k}, c_2)$

If $m = \perp$, return $\perp$

Return m

Figure 3.8: Distributed decryption protocol for Gladius-Hispaniensis using Hybrid$_1$

Section 4

Constructing a Distributed Decryption protocol based on the Kyber KEM

4.1 Distributed Decryption using Kyber

By utilizing some of the techniques used for Distributed Decryption in Gladius (see 3.2.4), we can do threshold decryption by putting the Kyber KEM decapsulation inside an MPC protocol.

The motivation for doing this is that when using the approach of Boudgoust et al. [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, means that we have to increase the value of q to get enough "space" for the additional noise to ensure the correctness of the 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.

In section 5.2, we describe how we realized the Distributed Decryption protocol for Kyber.

4.1.1 Main Protocol

The protocol for Distributed Decryption using the Kyber KEM can be seen in Figure 4.1. We use the notation $\langle x \rangle$ to denote a secret-shared value.

KEM Decapsulation As mentioned in section 2.3, decrypting in IND-CPA 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 the KEM decapsulation seen in Figure 4.1.

The scaling and rounding are then done in the loop on line $2 - 5$ of Figure 4.1. To understand why this works, we first observe that scaling and rounding, as done in Kyber, is equivalent to setting each coefficient of the binary output polynomial to be 1 if the corresponding coefficient of c' is closer to $q/2$ than it is to $q \equiv 0$ modulo q and 0 otherwise. This is reminiscent of the method used for the original LWE scheme by Regev displayed in 2.1. Now recall the Centre sub-procedure (see Figure 3.7). What this operation does, in effect, is subtract q from a bit-decomposed value x_i if $x_i \geq q/2$, where the output will be in 2's complement form if it is negative. If the output is positive, then $x_i < q/2$, which means that x_i was closer to $q/4$ than to $3q/4$, and so the most significant bit (MSB) will be 0, whereas it will be 1 otherwise due to the 2's complement representation. By adding $q/4$ to x_i, we can shift the windows where we output 1 and 0, meaning that using Centre will result in the MSB being 1 if x_i is closer to $q/2$ than it is to $q \equiv 0$ modulo q, which is equivalent to the scaling and rounding done in Kyber decryption. The idea of shifting the windows is visualized in Figure 4.2.

Summarising the above considerations, we see that the scaling and rounding can be done simply by first adding $q/4$ to each coefficient, then using Centre to get the bit-decomposition of $\mathbf{w}$ in the centered interval, then picking the MSB of each of the entries of $\mathbf{w}$. This will then produce our binary output polynomial.

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

Input: A ciphertext $ct = (\mathbf{u}, v)$, the KEM secret key $sk = (\langle s \rangle, h, \langle z \rangle)$, and the public key $pk = (\mathbf{A}, \mathbf{t})$

KEM Decapsulation

$\langle x \rangle \leftarrow v - \mathbf{u}^T \langle s \rangle$
for $i \in [1, ..., n]$ **do**
  $\langle y_i \rangle \leftarrow \langle x_i \rangle + q/4$
  $\langle \mathbf{w} \rangle \leftarrow \text{Centre}(\langle y_i \rangle)$
  $\langle m'_i \rangle \leftarrow \langle w_i^{(\text{msb})} \rangle$

Re-encryption

$(\langle \overline{K}' \rangle, \langle \text{coins} \rangle) \leftarrow G(\langle \mathbf{m}' \rangle, h)$
for $i \in [0, ..., k-1]$ **do**
  $\langle \mathbf{r}[i] \rangle \leftarrow \text{CBD}_{\eta_1}(\text{PRF}(\langle \text{coins} \rangle, N))$
  $N \leftarrow N + 1$
for $i \in [0, ..., k-1]$ **do**
  $\langle \mathbf{e}_1[i] \rangle \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(\langle \text{coins} \rangle, N))$
  $N \leftarrow N + 1$
$\langle e_2 \rangle \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(\langle \text{coins} \rangle, N))$
$\langle \mathbf{u}' \rangle \leftarrow \mathbf{A}^T \langle \mathbf{r} \rangle + \langle \mathbf{e}_1 \rangle$
$\langle v' \rangle \leftarrow \langle \mathbf{r} \rangle^T \mathbf{t} + \langle e_2 \rangle + \lfloor q/2 \rceil \cdot \langle \mathbf{m}' \rangle$

Key Derivation

if $(\mathbf{u}, v) = (\langle \mathbf{u}' \rangle, \langle v' \rangle)$
  $\langle K \rangle = \text{KDF}(\langle \overline{K}' \rangle, H(c))$
else
  $\langle K \rangle = \text{KDF}(\langle z \rangle, H(c))$
$K \leftarrow \textbf{Output}(\langle K \rangle)$
return K

Figure 4.1: Distributed Decryption protocol for Kyber

Re-encryption The Kyber KEM requires that we re-encrypt the message m' after decrypting using IND-CPA Kyber, which is what the Re-encryption step in the protocol does. The step follows the exact same structure as the regular Kyber encryption does. The unfortunate part is that the PRF used by Kyber uses a hash function, namely SHAKE256. It is necessary to execute these hash function calls in a distributed manner since if we reveal the randomness used for encryption, then the scheme becomes insecure.

The issue here is that hashing is hard to do inside MPC, so the need to do $2k + 1$ PRF calls, and thereby hash function calls, for IND-CPA encryption means that the performance is decreased drastically.

The primary reason why it is difficult to hash inside an MPC protocol is that we want hash functions to not have too much internal structure so that it is secure. This results in

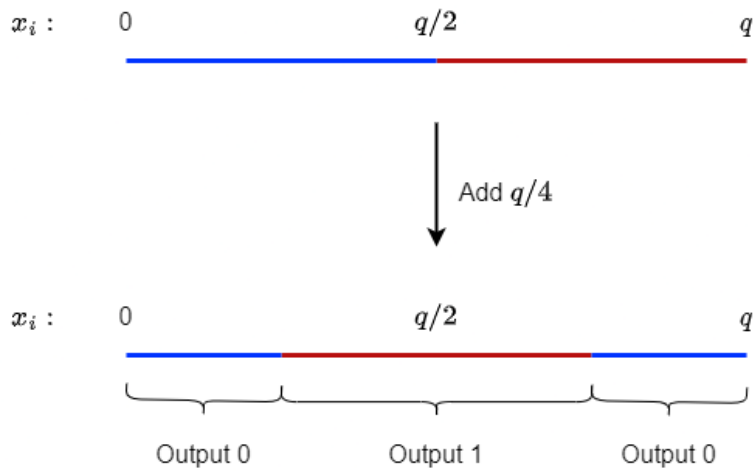

Figure 4.2: Illustration of the effect of using Centre then picking the MSB in the scaling and rounding of Distributed Decryption for Kyber. The blue indicates an output of 0, whereas the red indicates an output of 1.

a very non-linear function, which will require a lot of MPC operations to carry out. An additional reason is that many of the widely used hash functions, such as SHA2-256 or the SHA3 family, are made with efficient software and hardware implementations in mind, but the design constraints associated with these are different from the design constraints associated with MPC protocols [AABS+20]. Hash functions that are designed with MPC protocols in mind are often instead arithmetization-oriented, meaning that they operate in an arithmetic setting. An example of a hash function that has been designed with MPC in mind is the Rescue hash function mentioned earlier.

Key Derivation The Key Derivation step checks whether the result of re-encryption matches the input ciphertext. For this comparison, we cannot open the result of re-encryption, as mentioned by the Gladius paper. Then it uses the Key Derivation Function (KDF) from Kyber to get K in a shared form, which is then output. The KDF function is like the Kyber PRF implemented using SHAKE256, which means that we have to do an additional hash function call for our distributed decryption.

4.1.2 Preprocessing material

For the distributed decryption protocol, we need preprocessing material for arithmetic MPC and binary MPC, as well as edaBits, which allows for translation between arithmetic and binary MPC domains.

Arithmetic domain preprocessing material

One type of arithmetic MPC protocol is those based on LSS schemes. A lot of such MPC protocols, such as SPDZ [DKL+13] and BDOZ [BDOZ11], are split into an offline and online phase. In the offline phase, preprocessing material in the form of shared values is generated, and this material is then used for computations in the online phase.

In Section 6, we will do a theoretical analysis of the preprocessing material that will have to be generated in the offline phase in order to carry out the distributed decryption protocol. For this, we will assume a SPDZ [DKL+13] online phase. Here, given that we are doing MPC over a field $\mathbb{F}_q$, the secret-shared values are of the form

$$\langle a \rangle = ((a_1, ..., a_n), (\gamma(a)_1, ..., \gamma(a)_n))$$

for $a \in \mathbb{F}_q$ and n parties. Here, the a_i values are shares of a where $\sum_i^n a_i = a$, while the $\gamma(a_i)$ values are called MAC values. These should satisfy $\sum_i^n \gamma(a)_i = \alpha \cdot a$ for a secret-shared global key $\alpha \in \mathbb{F}_q$, generated at the start of the offline phase. The MAC values are then checked at the output step at the end of the online phase, and this is one of the measures used for SPDZ to achieve an actively secure MPC protocol. The MACCheck protocol presented in "Breaking the SPDZ Limits" [DKL$^+$13] for doing this check allows checking the MACs of opened values without opening the global key α, which means that unused preprocessing material can potentially be used for other computations at a later point. Assuming access to an ideal functionality for commitments, $\mathcal{F}_{\text{Commit}}$, the MACCheck protocol can be seen in Figure 4.3.

Protocol for MACCheck

Usage: Each Player i inputs their share of the global key α_i and MACs $\gamma(a_j)_i$ for $1, ..., m$.

Where all players have a set of publicly opened values $a_1, ..., a_m$.

The protocol fails if an inconsistent MAC value is found.

MACCheck$(a_1, ..., a_m)$

1: All players P_i sample a seed s_i and asks $\mathcal{F}_{\text{Commit}}$ to broadcast $\tau_i^s \leftarrow \text{Commit}(s_i)$

2: The players uses $\mathcal{F}_{\text{Commit}}$ with Open(τ_i^s), so that all players get s_j for $j = 1, ..., n$.

3: All players compute $s \leftarrow s_1 \oplus ... \oplus s_n$

4: All players sample $\mathbf{r} \leftarrow U(\mathbb{Z}_q^m)$ using seed s, such that they get the same vector.

5: All players compute public value $a = \sum_{j=1}^m r_j \cdot a_j$

6: Player i then computes $\gamma_i \leftarrow \sum_{j=1}^m r_j \cdot \gamma(a_j)_i$ and $\sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$

7: Player i uses $\mathcal{F}_{\text{Commit}}$ to broadcast $\tau_i^\sigma \leftarrow \text{Commit}(\sigma_i)$

8: Every player uses $\mathcal{F}_{\text{Commit}}$ with Open(τ_i^σ) so all players obtain σ_i for all i.

9: If $\sigma_1 + ... + \sigma_n \neq 0$ the players output $\varnothing$ and abort.

Figure 4.3: MACCheck protocol for the SPDZ online phase.

For additions, the linearity property of the LSS scheme ensures that we do not need any preprocessing material. For the SPDZ online phase, additions are performed by the players locally adding their shares of the input values.

For input sharing, we need to generate shared mask values $\langle r \rangle$ for uniformly random r. These are used to mask the private inputs from the participating parties. If a player P_i wants to give some input x_i, then they will broadcast $\epsilon = x_i - r_i$. All players then set $\langle x_i \rangle \leftarrow \langle r_i \rangle + \epsilon$. To add a constant c to a shared value $\langle x \rangle$, as done here, one of the players adds c to their share of x. Then all players update their corresponding MAC value to be $\gamma(x)_i + c \cdot \alpha_i$.

The second type of preprocessing material needed is multiplication triples, also called Beaver triples, of the form $\langle a \rangle, \langle b \rangle, \langle c \rangle$ for uniformly random a, b s.t. $ab = c$. These are used to be able to multiply values inside the MPC protocol. If the players hold shared values $\langle x \rangle, \langle y \rangle$ and want to multiply x and y to get $xy = z$, then they first take a multiplication triple $\langle a \rangle, \langle b \rangle, \langle c \rangle$, then open $\langle x \rangle - \langle a \rangle$ and $\langle y \rangle - \langle b \rangle$ to get ϵ and ρ respectively. Each player then locally computes $\langle z \rangle \leftarrow \langle c \rangle + \epsilon \cdot \langle b \rangle + \rho \cdot \langle a \rangle + \epsilon \cdot \rho$.

The last type of preprocessing material needed are integer bits, which are bits shared in $\mathbb{F}_q$. These are needed for integer comparisons and bit-decompositions.

Other offline phase protocols can be paired with the SPDZ online phase. These include MASCOT [KOS16] and Overdrive [KPR18].

daBits & edaBits

A Doubly-Authenticated Bit (daBit) is a type of preprocessing material used for conversions between an arithmetic and binary domain in MPC. A daBit is defined as a tuple $(\llbracket b \rrbracket_M, \llbracket b \rrbracket_2)$ for some bit b where $\llbracket \cdot \rrbracket_p$ is the secret sharing of the bit mod p. An Extended Doubly-Authenticated Bit (edaBit) is then an extension of this concept first presented by Escudero et al. [EGK+20], which improves the performance of daBits. An edaBit is defined to be a collection of bits $(r_{m-1}, \ldots, r_0)$, where each bit r_i and the integer $r = \sum_{i=0}^{m} r_i \cdot 2^i$ are secret shared.

To generate edaBits each party P_i will sample m random bits $r^i_{i,0}, \ldots, r^i_{i,m}$, and secret share these in $\mathbb{Z}_2$ along with the integer $r = \sum_{i=0}^{m} r_i \cdot 2^i$ over $\mathbb{Z}_M$. This will yield private edaBits, as one party must act as a trusted dealer. To generate global edaBits, the parties each generate a private edaBit and combine these to form global edaBits, in which no player knows the underlying values. The arithmetic shares can be simply added locally as $\llbracket r_i \rrbracket_M = \sum_{i=0}^{n} \llbracket r_i \rrbracket_M$ and the binary input can be added using any n-input binary adder.

edaBits can save a factor between 2 and 60 in communication costs, compared with a purely arithmetic approach, and a factor between 2 and 25 from any approach using daBits. These can be obtained by using edaBits in already existing conversion protocols.

To convert from a field to a binary share and vice versa using daBits, we can use the algorithms from "Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE" [AOR+19] defined in figure 4.5 and 4.6. These use the sub-protocol DaShare from figure 4.4.

DaShare

1: Execute Π_{daBits} to generate l daBits $(\llbracket b^{(k)} \rrbracket_p, \llbracket b^{(k)} \rrbracket_2)_{k=0}^{l-1}$.
2: Compute $\llbracket r \rrbracket_p \leftarrow \sum_{i=0}^{l-1} 2^k \cdot \llbracket b^{(k)} \rrbracket_p$ and set $\llbracket \boldsymbol{r} \rrbracket_p \leftarrow (\llbracket b^{(k)} \rrbracket_2)_{k=0}^{l-1}$.
3: Output $(\llbracket r \rrbracket_p, \llbracket \boldsymbol{r} \rrbracket_2)$.

Figure 4.4: DaShare method

ConvertToField

Input: A vector of sharings $\llbracket \mathbf{x} \rrbracket_2$

1: The parties call `DaShare` with $l = 64$, to obtain a doubly authenticated share $(\llbracket r \rrbracket_M, \llbracket \mathbf{r} \rrbracket_2)$.
2: The parties locally compute, for $i = 0 \ldots 63$, the shares $\llbracket v_i \rrbracket_2 \leftarrow \llbracket x_i \rrbracket_2 \oplus \llbracket r_i \rrbracket_2$
3: The value v_i is then opened to all parties.
4: The parties locally compute, for $i = 0 \ldots 63$, the shares $\llbracket x_i \rrbracket_p \leftarrow v_i + \llbracket r_i \rrbracket_p - 2 \cdot v_i \llbracket r_i \rrbracket_p$.
5: From these shared bits they can reconstruct x as an $\llbracket \cdot \rrbracket_p$-sharing via $\llbracket x \rrbracket_p \leftarrow -\llbracket x_{63} \rrbracket_p \cdot 2^{63} + \sum_{i=0}^{63} \llbracket x_i \rrbracket_p \cdot 2^i$.

Figure 4.5: ConvertToField method

Binary domain preprocessing material

For the protocol, we will also need to generate preprocessing material for binary domain MPC since we need to be able to compute various SHA3 hash functions for Kyber. If we here assume an LSS-based MPC protocol, like for the arithmetic MPC, we can view XOR operations as an analog of additions in that the linearity property ensures that we do not

ConvertToBinary

input: a sharing $[\![x]\!]_p$

1: Set $l = \lfloor log_2\ p \rceil$ in the case that $64 + sec \leq \lceil log_2\ p \rceil$, else set $l \leftarrow 64 + sec$.

2: The parties call `DaShare` with the given value of l to get a doubly authenticated share $([\![r]\!]_p, [\![\boldsymbol{r}]\!]_2)$.

3: All parties compute $[\![y]\!]_p \leftarrow [\![x]\!]_p - [\![r]\!]_p$

4: The value y is then opened to all parties such that they all obtain $y = x - r$ in clear text.

5: The parties compute $[\![x]\!]_2 \leftarrow y + [\![r]\!]_2$ by using an MPC protocol for binary circuits on a circuit representing addition modulo p, which converts integers in the set $[0, ..., 2^{63}) \cup (p - 2^{63}, p - 1)$ into 64-bit signed integers in two's complement representation.

Figure 4.6: ConvertToBinary method

need any preprocessing material. Additionally, AND gates can be viewed as an analog of multiplications in that they require triples in order to be carried out. In this case, the triples are instead secret shared bits.

4.1.3 Kyber KEM transformation compared to Gladius Hybrid transformations

The security of the Gladius distributed decryption procedure relies on theorem 3.1 of the Gladius paper. The accompanying proof relies on the public-key encryption scheme used being either a randomized, rigid, OW-PCA secure or a deterministic, rigid, OW-CPA secure PKE. The rigidity requirement is formalized as

$$\Pr[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathcal{K}(1^t), c \leftarrow \mathcal{C} \setminus \mathcal{C}^{\perp}, \exists r \in \mathcal{R}, : \mathcal{E}(\mathsf{pk}, \mathcal{D}(\mathsf{sk}, c); r) = c] = 1$$

What this says is that decryption $\mathcal{D}(\mathsf{sk}, c)$ always fails (outputs $\perp$) unless the ciphertext c is the output of an encryption operation $c = \mathcal{E}(\mathsf{pk}, m; r)$ for some message and randomness m, r.

It is not clear that the standard IND-CPA Kyber PKE is rigid. Specifically, Kyber uses the `CBD` algorithm for sampling polynomials used in computing the ciphertext. The `CBD` algorithm is a centered binomial distribution used for approximating a left and right truncated Gaussian distribution. In other words, the coefficients of a polynomial output by `CBD` are small and in a restricted interval. This means that possibly some ciphertexts of the form $(\mathbf{u}, v) \in (R_q^k, R_q)$ might not be the result of an encryption operation, and if a ciphertext is not of this form, then decryption does not fail. Therefore, the Hybrid_1 and Hybrid_2 security proofs would likely not hold for Kyber.

The PKE Π_p used for Gladius already relies on re-encryption by itself, which ensures rigidity. This, in effect, means that the Gladius PKE is already rigid. Thus, it can be used without further modifications.

The Gladius paper also mentions a way of transforming any randomized PKE into one that is rigid, called *Encrypt-With-Hash*. This, however, requires re-encryption, which in the case of Kyber, would introduce a lot of hashing. The benefit of the Gladius hybrid transformations is precisely that the amount of hash function calls is low, thus avoiding issues with distributed hashing, so having to do re-encryption defeats the purpose of using it in the first place. This motivates our use of the standard Kyber KEM rather than trying to combine IND-CPA Kyber with the Gladius Hybrid transformations.

4.1.4 Correctness and Security

The correctness of the distributed decryption protocol follows immediately from the considerations outlined above, alongside the correctness of the Kyber KEM.

As for the case of Distributed Decryption in Gladius, given that the Kyber KEM distributed decryption protocol has been implemented correctly, then the security follows directly from the security of the underlying MPC protocol. What this means is that if an actively secure MPC protocol is used for the implementation, then the $\Pi_{\text{DDecKyber}}$ protocol is actively secure.

4.2 Distributed Key Generation for Kyber

For generating a Kyber key pair in a distributed manner, we have two different methods.

4.2.1 Method 1: Adding each privately generated key

The first method is also the simplest. For this method, all players sample their own secret key $sk_i = (\mathbf{s}_i, \mathbf{e}_i)$ using the `CBD` Kyber algorithm. The shared secret key is then $sk = (\mathbf{s}, \mathbf{e})$ where

$$\mathbf{s} = \sum_i \mathbf{s}_i \quad \text{and} \quad \mathbf{e} = \sum_i \mathbf{e}_i$$

It then becomes possible to compute and open $\mathbf{t}$ inside an MPC protocol, along with just sampling $\mathbf{A}$ in the clear to get the public key. This protocol can be seen in Figure 4.7.

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

1: **Input**: Values $\langle e^{(i)} \rangle, \langle s^{(i)} \rangle$ sampled from **CBD** privately by each player i
2: $\langle s \rangle \leftarrow \sum_i \langle s^{(i)} \rangle$
3: $\langle e \rangle \leftarrow \sum_i \langle e^{(i)} \rangle$
4: $A \leftarrow R_q^{r \times r}$
5: $\langle t \rangle \leftarrow A\langle s \rangle + \langle e \rangle$
6: $t \leftarrow \textbf{Open}(\langle t \rangle)$
7: $\mathsf{sk} \leftarrow (\langle s \rangle, \langle e \rangle)$
8: $\mathsf{pk} \leftarrow (A, t)$

Figure 4.7: A protocol for distributed key generation for Kyber

The benefit of this method is that it is efficient since we do not have to use an algorithm for sampling from a centered binomial distribution inside the MPC protocol itself. The downside is that we end up adding the error terms sampled by the players, so we end up with more noise. This also still leaves us with the issue of dealing with malicious inputs. For this, it would be possible to employ a Zero-Knowledge (ZK) proof, where a player would prove in ZK that their input is not sampled maliciously. This would also introduce some overhead.

4.2.2 Method 2: Distributed sampling from a centered binomial distribution

Another possible technique would be to simply sample bits in a distributed manner, then run the `CBD` algorithm inside the MPC protocol to get secret vectors $\mathbf{s}$ and $\mathbf{e}$. $\mathbf{A}$ and $\mathbf{t}$ are

then once again computed as in method 1. This method is very similar to what is done for distributed key generation in Gladius.

The benefit of using this technique is that it more accurately reflects the Kyber specification. Another upside is that we can get away with a smaller q parameter since, in this case, the noise term is not larger than what it would have been in regular Kyber key generation. A downside is that this introduces an overhead for a distributed sampling of bits and for computing `CBD`.

This idea is essentially what has also been introduced in *"Actively Secure Setup for SPDZ"* by Rotaru et al. [RST+19]. Here the authors show how to sample from a centered binomial distribution, just like `CBD`, in a distributed manner.

Section 5

Implementation

In this chapter, we describe our implementation of the Threshold Public Key Encryption scheme TKyber, along with The Kyber KEM and Gladius distributed decryption protocols.

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

5.1 Implementing TKyber

In this section, 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.

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 schemes, the OW-CPA TKyber implementation, and the IND-CPA TKyber implementation.

5.1.1 LSS Schemes

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.

Replicated LSS

When using Replicated LSS, we generate all t-size subsets S_k of the elements $[0, ..., n]$, such that there are $L = \binom{n-1}{t}$ subsets. The subsets are represented as a slice of slices of int values and are generated using the `MakeCombinations` method from the `util` package. We then additively share each polynomial in the secret key using our Additive LSS implementation,

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

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

such that there are L shares, and then we distribute the k'th share to party i, if and only if $i \notin S_k$.

When we want to reconstruct the original polynomial, we use `MakeCombinations` to check which parties are supposed to have each of the shares. Then `PolyAdd` from kyber-k2so is used to add together each of the L shares to reconstruct the original polynomial.

Naive LSS

For the Naive LSS implementation, we once again use `MakeCombinations`. Here we generate all $(t+1)$-size subsets. For each subset, we use our additive LSS implementation to generate a sharing of each of the polynomials in the secret key with $t+1$ shares, which are then distributed.

For reconstruction, we use `MakeCombinations` to get all subsets of size $t+1$, then we use `PolyAdd` to recompute the secret from one of the sharings, for which all $t+1$ shares have been given as input (if one exists).

5.1.2 OW-CPA TKyber (`owcpa_TKyber.go`)

To be able to vary the LSS scheme used as well as the noise distribution, we utilize 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.

Setup

Our implementation of the Setup function looks as follows:

```
1  func Setup(params *OwcpaParams, n int, t int) ([]byte, [][]kyberk2so.
       PolyVec) {
2      // Run setup to get Kyber KeyPair
3      sk, pk, _ := kyberk2so.IndcpaKeypair(kyberk2so.ParamsK)
4      sk_unpacked := kyberk2so.IndcpaUnpackPrivateKey(sk, kyberk2so.
           ParamsK)
5
6      // Perform secret sharing
7      sk_shares := params.LSS_scheme.Share(sk_unpacked, n, t)
8
9      return pk, sk_shares
10 }
```

On line 3, we use `IndcpaKeypair` from kyber-k2so to sample a key pair. Then on line 4, 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 on line 7. 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 3.1. 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 to 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. The code for this can be seen below.

```
1  func PartDec(params *OwcpaParams, sk_i []kyberk2so.PolyVec, ct []byte,
       party int) []kyberk2so.Poly {
2    var zero kyberk2so.Poly
3
4    u, v := kyberk2so.IndcpaUnpackCiphertext(ct, kyberk2so.ParamsK)
5
6    // Instantiate d_i and put u on NTT form, which is needed for inner
         prod.
7    d_i := make([]kyberk2so.Poly, len(sk_i))
8    kyberk2so.PolyvecNtt(u, kyberk2so.ParamsK)
9
10   for j := 0; j < len(sk_i); j++ {
11     // Sample noise
12     e_i := params.D_flood_dist.SampleNoise(params, 255)
13
14     // Inner prod.
15     d_i[j] = kyberk2so.PolyvecPointWiseAccMontgomery(sk_i[j], u,
           kyberk2so.ParamsK)
16     d_i[j] = kyberk2so.PolyInvNttToMont(d_i[j])
17
18     if shouldSubV(params, party, j) {
19       d_i[j] = kyberk2so.PolySub(v, d_i[j])
20     } else {
21       d_i[j] = kyberk2so.PolySub(zero, d_i[j])
22     }
23
24     // Add noise
25     d_i[j] = kyberk2so.PolyAdd(d_i[j], e_i)
26     d_i[j] = kyberk2so.PolyReduce(d_i[j])
27   }
28
29   return d_i
30 }
```

Each iteration of the loop on line 10 computes one of the $d_{i,j}$ values. In particular, the inner product is computed by using `PolyvecNtt` to get u on NTT form, then the function `PolyvecPointWiseAccMontgomery` is used on line 15 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 additions and subtractions are then performed using `PolyAdd` and `PolySub`.

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. Thus the implementation of this function looks as follows:

```
1  func Combine(params *OwcpaParams, ct []byte, d_is [][]kyberk2so.Poly, n
       int, t int) kyberk2so.Poly {
2    y := params.LSS_scheme.Rec(d_is, n, t)
```

```
    return y
}
```

Notice that we don't have any scaling in the Combine function as done in Figure 3.1. 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.

5.1.3 IND-CPA TKyber (`indcpa_TKyber.go`)

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

Realising the Random Oracles F and G

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

Implementing F was done by converting all of the polynomials given as input to bytes using `PolyToBytes` from kyber-k2so, hashing the concatenation of these using `Shake256` to get a byte slice of size 384 bytes, which is the number of bytes used to represent a whole polynomial by the Kyber encryption scheme, then using `PolyFromBytes` from kyber-k2so to convert the resulting bytes into a polynomial of the `Poly` type. The reasoning behind using `Shake256` is that it is an Extendable-Output Function (XOF), 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.

```
func Setup(params *owcpa.OwcpaParams, n int, t int) ([]byte, [][]
    kyberk2so.PolyVec) {
    return owcpa.Setup(params, n, t)
}
```

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.

```
func Enc(params *owcpa.OwcpaParams, msg []byte, pk []byte, delta int) *
    indcpaCiphertext {
    mp := kyberk2so.PolyFromMsg(msg)
    x := make([]kyberk2so.Poly, delta)
    for i := 0; i < delta; i++ {
        x[i] = owcpa.SampleUnifPolynomial(2)
```

```
    }

    c := new(indcpaCiphertext)
    c.encyptions = make([][]byte, delta)

    c0 := kyberk2so.PolyAdd(mp, F(x))
    c.cF = c0
    for i := 0; i < delta; i++ {
    upscaled := owcpa.Upscale(x[i], 2, params.Q)
    xi_bytes := kyberk2so.PolyToMsg(upscaled)
    c.encyptions[i] = owcpa.Enc(params, xi_bytes, pk)
    }
    c.cG = G(x)

    return c
}
```

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.

```
func PartDec(params *owcpa.OwcpaParams, sk_i []kyberk2so.PolyVec, ct *
    indcpaCiphertext, party, delta int) [][]kyberk2so.Poly {
    d_i := make([][]kyberk2so.Poly, delta)
    for j := 0; j < delta; j++ {
        d_i[j] = owcpa.PartDec(params, sk_i, ct.encyptions[j], party)
    }
    return d_i
}
```

Combine

For `Combine`, we get as input a 3-dimensional matrix `d_is`, which contains all of the player's partial decryptions of the δ encryptions. The issue here is that we want to combine the partial decryptions of each of the encryptions individually, but the first dimension of `d_is` is the particular player, the second dimension is the particular encryption, while the third is L. Because of this, we first swap the first and second dimensions of the matrix so it is easier to fetch all of the players' partial decryptions for a given encryption.

After swapping the dimensions of `d_is` 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`.

Afterward, 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.

```
func Combine(params *owcpa.OwcpaParams, ct *indcpaCiphertext, d_is
    [][][]kyberk2so.Poly, n int, t int) kyberk2so.Poly {
    delta := len(ct.encyptions)

```

```
    x_prime := make([]kyberk2so.Poly, delta)
    d_is_transp := util.SwapFirstAndSecondDim(d_is)

    for j := 0; j < delta; j++ {
        combined := owcpa.Combine(params, ct.encyptions[j], d_is_transp[
            j], n, t)
        x_prime[j] = owcpa.Downscale(combined, 2, params.Q)
    }

    mp := kyberk2so.PolySub(ct.cF, F(x_prime))

    if !reflect.DeepEqual(ct.cG, G(x_prime)) {
        panic("Error: c_(delta + 1) != G(x')")
    }

    return mp
}
```

5.1.4 Testing of correctness

Testing the correctness of the TKyber implementation was done using automated tests implemented using Go's built-in "testing" package.

In particular, for the OW-CPA implementation, we use the "testing" package's subtesting functionality to test the consistency of the implementation, i.e., check that using `PartDec`, then Combine gives us the original message. Using the subtesting functionality allows us to test the consistency for a lot of parameter sets in a very condensed manner. In addition to this, we also have tests where we use a binomial noise distribution in `PartDec`, where encrypt has been made deterministic, specific tests for `Setup`, and a test for Replicated LSS where we combine manually instead of using the `Combine` function. Lastly, we also have a test covering a specific bug that was previously found in the code. The code coverage for the OW-CPA tests is 92.9% of statements.

The tests for the IND-CPA code are very similar to that of the OW-CPA tests, so we again have a mix of individual tests and consistency tests carried out through subtesting. The code coverage for the IND-CPA tests is 97.7% of statements.

5.1.5 Allowing larger modulus via the Chinese Remainder Theorem

Regular Kyber uses $q = 3329$ as the modulus for all parameter sets. For TKyber, we need more space for the extra noise that we get, so as a result, we need to increase the modulus. Unfortunately, for the kyberk-2so package that we use, the coefficients of polynomials are represented as signed 16-bit integers, and this is relied heavily upon in the implementation to get better performance through bit manipulation, so we are limited to a 15-bit modulus. This is not enough for the largest modulus used by the parameter sets displayed in Table 3.2.

To avoid having to implement Kyber from scratch, it is possible to use the Chinese Remainder Theorem (CRT) to get a modulus that is a product of multiple 15-bit moduli. This would allow us to simulate larger coefficients using multiple Kyber operations. The unfortunate part here is that we did not manage to change the modulus of Kyber-k2so since it seems that the modulus is implicit in parts of the implementation. Still, this is an interesting idea that could be used if we actually managed to change the modulus.

To use this trick, we first pick coprime numbers $q_1, ..., q_k$. Then we define $N = q_1 \cdot q_2 \cdot \cdots \cdot q_k$. Now we encrypt k times, each using a distinct of the k coprime numbers as the modulus. Now, for threshold decryption, we let all parties compute their partial decryption share of

each of the k encryptions. When combining, we only recombine using the LSS scheme, which leaves us with k polynomials with each of the n size-k pairs of coefficients satisfying the following equations

$$x_j \mod q_1$$
$$\dots$$
$$x_j \mod q_k$$

Now we use Euclid's Extended Algorithm to compute a polynomial y with coefficients satisfying

$$x_j \mod N$$

After computing y, we can then do the scaling and rounding as we usually do. By using this process, we can use larger moduli. This will have a large hit to the performance and sizes of keys and ciphertexts due to multiple Kyber operations having to be performed in the case of a large modulus, but it comes with the upside of allowing us to keep using the heavily optimized kyber-k2so implementation.

5.2 Implementing DDec for the Kyber KEM (`kyber_ddec.mpc`)

We implemented the distributed decryption protocol described in section 4.1 using MP-SPDZ [Kel20], which is a framework for MPC.

Another library, SCALE-MAMBA [ACC+18], was also considered but is no longer actively developed, so we chose to use MP-SPDZ instead.

The MP-SPDZ framework provides a high-level interface for doing MPC by being able to compile Python-like code, extended with extra types and functions for operating on secret data, into bytecode that can be run by the MP-SPDZ virtual machine. Programs for MP-SPDZ use the .mpc extension.

MP-SPDZ provides the basic types sint, sfix, sfloat, and sgf2n for representing secret data. These represent a secret integer, fixed-point number, float, and GF(2^n) value, respectively. In addition, analog types for values in the clear are also provided. These are simply prefixed by c instead of s. Opening secret values is then done by calling the .reveal() method.

For working with circuits, MP-SPDZ provides types sbit, sbitfix, sbitfixvec, sbitint, sbitintvec, sbits, sbitvec for working on secret data, and cbits for operating on data in the clear.

In addition to the types mentioned above, MP-SPDZ also provides container types MemValue, Array, Matrix, and MultiArray. These can contain any of the basic types.

For distributed decryption using Kyber, we use one of the arithmetic MPC protocols provided by MP-SPDZ that works over a field $\mathbb{F}_q$. By picking the modulus to be the Kyber modulus $q = 3329$, all of the reductions modulo q becomes implicit in the operations when working with cint and sint values.

5.2.1 Centre

For the centre subroutine, we have to be able to do bit decomposition of sint values. To do this, we use the method .BitDecFull() used internally in MP-SPDZ. This method is implemented according to the Bit-Decomposition protocol from *"Multiparty Computation for Interval, Equality, and Comparison without Bit-Decomposition Protocol"* [NO07], which is the same algorithm used by Gladius. The reason for using this as opposed to the regular method for Bit-Decomposition that MP-SPDZ provides, .bit_decompose(), is that for secret values .bit_decompose() uses comparisons requiring the field to be 2 bits larger than the amount of bits output by the Bit-Decomposition, and so since our modulus is 12 bits long the method outputs 10 bits, which is too small for the coefficients since these are in the range $[0, 3329)$.

For BitAdd MP-SPDZ likewise also has a method BitAdd() that we can use directly. For computing BitNeg, which returns the two-complement negative of a bit vector, we implemented our own function by using BitAdd(). For the BitLT comparison, MP-SPDZ includes a function BitLT(), which is implemented according to the paper *"Unconditionally Secure Constant-Rounds Multi-Party Computation For Equality, Comparison, Bits, and Exponentiation"* by Damgaard et al. [DFK+06]. Using all of these operations we then implement `centre` as follows:

```
1 def centre(x_i_angle):
2     # BitDecomp
3     b = floatingpoint.BitDecFull(x_i_angle, 12)
4 
5     # BitAdd
6     qp1_bits = cint(q + 1).bit_decompose(bit_length = 12)
7     b_prime = floatingpoint.BitAdd(flipBits(b), qp1_bits)
8 
9     # BitNeg
```

```
    b_prime_prime = bitNeg(b_prime)

    # BitLT
    q_div_2 = cint(1665).bit_decompose(bit_length = 12)
    f=floatingpoint.BITLT(b, q_div_2, 12)

    # Compute <a>
    a = []
    for i in range(len(b)):
        a.append(f * b[i] + (1 - f) * b_prime_prime[i])

    return a
```

5.2.2 Polynomials and R_q arithmetic

To represent polynomials we use the Array container type containing either sint or cint values, depending on if the polynomial is secret or not.

To be able to implement KEM Decapsulation, we also need to be able to multiply polynomials in the ring R_q. For this, we implemented the Number Theoretic Transformation (NTT) for Kyber described in Section 2.3, along with functionality for multiplying polynomials in the NTT domain. In addition to this, we also use the Montgomery domain as done for Kyber, but we do not use Barrett Reductions since the reductions modulo q are, of course, implicit in our choice of $q = 3329$ as the modulus for the MPC protocol.

The implemented NTT transformation is heavily inspired by an existing Python Kyber NTT implementation[3], which is easily adapted because MP-SPDZ uses a Python-like high-level interface, as mentioned earlier.

5.2.3 KEM Decapsulation

For the first step of KEM decapsulation, computing $v - u^T\langle s\rangle$, we have to compute an inner product, along with performing a subtraction of two vectors.

For the inner product, we use the NTT-based polynomial multiplication functionality described in the previous section. The subtraction is easily done using a loop.

For steps $2 - 5$, we use standard Python list comprehension to add $\lfloor q/4 \rceil = 832$ to each coefficient. Then we apply the centre function described earlier to each coefficient to get the list of centered Bit-Decompositions `w`. Extracting the MSB of each centered Bit-Decomposition is then simply a matter of indexing into `w`.

Thus the code for the KEM Decapsulation part of DDec looks as follows:

```
def kem_decap(u, v, s_angle):
    # v - u^T <s>
    u_ntt = polyvecNTT(u)
    x = inner_prod_NTT(u_ntt, s_angle)
    x = poly_sub(v, x)

    # Scaling and rounding
    y = [x_i + sint(832) for x_i in x]
    w = [centre(y_i) for y_i in y]
    k = [w_i[-1] for w_i in w] # get MSB of each w_i
    return k
```

[3] https://github.com/jack4818/kyber-py/blob/main/ntt_helper.py.

5.2.4 Re-encryption

For the Re-encryption step, we re-compute the coins used for re-encryption, along with a value $\overline{K}'$ used in Key Derivation. This is done by calling the hash function G, for which Kyber uses SHA3-512 in the "modern" instantiation, which we will be using. MP-SPDZ does not implement SHA3-512, but instead, it provides a function .sha3_256(), which is a SHA3-256 implementation included in the high-level interface. We modified this implementation into a general-purpose Keccak implementation, such that we can use this to compute SHA3-512 along with SHAKE256, which we will need for the IND-CPA Kyber encryption and for the key derivation.

For the CPAPKE encryption, we use the CBD algorithm with input computed by the PRF function. The PRF itself uses SHAKE-256 and was implemented using our Keccak implementation, while the CBD algorithm was implemented as described in the Kyber specification. Apart from this, we rely on R_q arithmetic explained earlier for the rest of the computations.

This gives the following implementation:

```
def CPAPKE_kyber_encryption(A_t, t, m, coins):
    N = cint(0)
    r_bold = Matrix(r, n, sint)
    for i in range(r):
        r_bold[i] = CBD(PRF(coins, N, eta1), eta1)
        N = N + 1

    e_1_bold = Matrix(r, n, sint)
    for i in range(r):
        e_1_bold[i] = CBD(PRF(coins, N, eta2), eta2)
        N = N + 1
    e_2 = CBD(PRF(coins, N, eta2), eta2)

    # Compute A^t r + e_1
    r_bold_ntt = polyvecNTT(r_bold)
    u = Matrix(r, n, sint)
    for i in range(r):
        res = inner_prod_NTT(A_t[i], r_bold_ntt)
        for j in range(n):
            u[i][j] = res[j] + e_1_bold[i][j]

    # Compute tr + e_2 + m scaled
    tr = inner_prod_NTT(t, r_bold_ntt)
    tre = poly_add(tr, e_2)
    m_scaled = scalar_mult(m, 1665)
    v = poly_add(tre, m_scaled)

    return u, v
```

By combining the CPAPKE implementation and G, we get the following complete description of the Re-encryption step:

```
def re_encryption(k_angle, hash_pk, A_t, t):
    K_bar_prime, r_prime = G(k_angle, hash_pk)
    u_prime, v_prime = CPAPKE_kyber_encryption(A_t, t, k_angle, r_prime)
    return u_prime, v_prime, K_bar_prime
```

5.2.5 Key Derivation

The Key Derivation step is fairly simple. It is implemented by comparing each coefficient of the ciphertext computed by re-encryption $c' = (u', v')$, with the original ciphertext $c = (u, v)$, by using secret integer (sint) comparisons as implemented by MP-SPDZ. If all coefficients match, then the `test` variable will have value 1, and so we use arithmetization to set the first input of the KDF hash function to either $\overline{K}'$ from the G call in the Re-encryption step, or a random value z which is a part of the Kyber KEM secret key.

The code for this step looks as follows:

```
def key_derivation(u_prime, v_prime, u, v, K_bar_prime, hash_c, z):
    test = sint(1)
    # Check u
    for i in range(r):
        for j in range(n):
            check = u_prime[i][j] == u[i][j]
            test = test * check
    # Check v
    for i in range(n):
        check = v_prime[i] == v[i]
        test = test * check

    # Compute KDF input
    in_1 = [test*sint(k_i) for k_i in K_bar_prime]
    in_2 = [(1 - test)*z_i for z_i in z]
    input = [sint(0)] * 256
    for i in range(256):
        input[i] = in_1[i] + in_2[i]

    return KDF(input, hash_c)
```

5.2.6 Distributed key generation for Kyber (`kyber_dkeygen.mpc`)

For distributed key generation, we implemented the protocol according to the first of the two methods presented, where each player privately samples a secret key from **CBD**, then adds their sampled values together inside the MPC protocol to get the shared secret key.

The implementation itself is quite simple and uses the NTT-based polynomial multiplication function described in Section 5.2.2. The full MP-SPDZ implementation can be seen below.

```
def dkeygen(s_values, e_values):
    # Sum
    s_angle = Matrix(r, n, sint)
    e_angle = Matrix(r, n, sint)
    for p in range(player_numb):
        for i in range(r):
            for j in range(n):
                s_angle[i][j] += s_values[p][i][j]
                e_angle[i][j] += e_values[p][i][j]

    A = get_rand_A()
    z = Array(256, sint)
    for i in range(256):
        z[i] = sint.get_random_bit()
```

```
15
16      t_angle = Matrix(r, n, sint)
17
18      # t = As + e
19      s_ntt = polyvecNTT(s_angle)
20      for i in range(r):
21          A[i] = polyvecNTT(A[i])
22          res = inner_prod_NTT(A[i], s_ntt)
23          for j in range(n):
24              t_angle[i][j] = res[j] + e_angle[i][j]
25
26      t = t_angle.reveal()
27
28      # Here A, s_ntt, t are on NTT form at the end
29      return A, t, s_ntt, e_angle, z
```

5.2.7 Potential optimisations

For the implementations using MP-SPDZ, there are a number of potential optimizations that we have not implemented.

One particular issue is that of hashing, which as mentioned, is hard to do inside MPC. What we could do for hashing is to use an MPC-friendly hash function such as Rescue, which is the one used for Gladius (with modifications). This would allow us to get better performance for the distributed decryption. Since Rescue works, over a large prime field, we would also be able to eliminate the need to mix binary and arithmetic MPC, which could lead to a performance improvement. This would, however, mean that we would diverge slightly from the Kyber specification.

Previously, edaBits have also been mentioned as a form of preprocessing that can be used for better performance. Currently, for fields as small as the one we are using, MP-SPDZ only implements conversions between arithmetic and binary domains using daBits and not edaBits in the high-level interface. MP-SPDZ does, however, allow generating an edaBit manually, so here, we would be able to use this to implement a conversion that can use edaBits.

Another possibility would be to use multithreading. MP-SPDZ already implements functionality to make this easier, so this could be employed to make the performance even faster.

5.3 Implementing Distributed Decryption for Gladius

For Gladius, we implemented the regular Gladius-Hispansiensis OW-CPA cryptosystem as well as the Hybrid$_1$ transformation in Go to ensure that we could generate test vectors for distributed decryption. This implementation can be found in the gladius folder of our threshold Kyber repository. In the rest of this section, we outline our implementation of Gladius-Hispansiensis distributed decryption in MP-SPDZ. Implementing it is of particular interest to us since we can then compare it to distributed decryption based on the Kyber KEM.

5.3.1 KEM Decapsulation

For the decapsulation of the Gladius KEM, we implemented vector subtraction and vector-matrix multiplication. In addition to this, we reuse the centre implementation from the Kyber KEM DDec protocol. For this step, we get the following code:

```
def kem_decap(ct1, ct2, R1):
    x = vecSub(ct2, vecMatMult(ct1, R1))
    w = [centre(x_i) for x_i in x]
    k = [w_i[nu - 1].bit_xor(w_i[nu+1 - 1]) for w_i in w] # the -1 is
        due to 0 indexing
    return k
```

5.3.2 Validity Check

The validity check is quite a long procedure, but it relies heavily on MP-SPDZ operations already explained, namely bit-decomposition, BITLT, BitNeg, and BitAdd. In addition to this, it also uses the vector/matrix operations explained earlier. We will refrain from including the actual code due to the length but will refer the interested reader to our git repository linked earlier, in which the gladius code can be found in the gladius folder.

5.3.3 Hash Check

For the hash check, we hash the concatenation of c_2 and $\langle \mathbf{k} \rangle$ using the MP-SPDZ SHA3-256 implementation, .sha3_256(), mentioned earlier.

Following this, we enter a loop where we reveal each of the bits output by hashing, then compare those to the bits of c_3 from the Hybrid$_1$ ciphertext. If any of the bits do not match, then we print "Fail".

So the Hash Check procedure ends up looking as follows:

```
def hash_check(k_angle, c_2, c_3):
    # Distributed hashing to get <t>
    k_angle_sintbit = [sintbit(x_i) for x_i in k_angle] # Convert to
        sintbit to indicate that they contain bits.
    k_angle_sbit = [sbit(x_i) for x_i in k_angle_sintbit] # Conversion
        from sintbit to sbit can be done in any domain.
    input = c_2 + k_angle_sbit

    to_hash = sbitvec.from_vec(input)
    t_angle = sha3_256(to_hash) # Note: Input and output for sha3_256
        are byte-reversed

    # Compare t and c_3
```

```
    for i in range(len(t_angle.v)):
        current_bit = cbit(t_angle.v[i].reveal())
        c_3[i].bit_xor(current_bit).print_if("Fail")
```

5.3.4 Message Extraction

For this function, we first open $\langle k \rangle$, then we simply print the result.

This does not correspond completely to Figure 3.8, since we simply output the key instead of actually hashing to get the symmetric key $\mathbf{k}$, then using this with Π_s to decrypt c_2. However, the hashing and decryption are done in the clear, so we implemented this separately outside MP-SPDZ. The reason for this choice is that MP-SPDZ only provides a method .digest() for hashing in the clear, which uses the libsodium library. Calling this method can, however, only be done on a cint. The issue is then that our k value is a vector of n bits, so in practice, we get a number that far exceeds the moduli used by Gladius, which we also use for the MPC protocol, so this would result in a modular reduction, which would mean an incorrect result.

Another choice would be to hash $\langle k \rangle$ and or decrypt c_2 before revealing, but this is clearly inefficient compared to doing the operations in the clear.

After these considerations, we end with the following piece of code for the message extraction procedure.

```
def message_extraction(ct, k_shared):
    k_bits_clear = Array(n, cint)
    for i in range(len(k_angle)):
        k_bits_clear[i] = k_angle[i].reveal()
    k_bits_clear.print_reveal_nested()
```

Section 6

Benchmarking and Analysis

In the following section, we benchmark, analyze, and compare regular Kyber, our implementation of TKyber, and our implementations of distributed decryption for Kyber and Gladius-Hispaniensis. The benchmarks were run on a Macbook M2 Pro with 32 Gb of RAM and were further run in a virtual machine using Vmware software on a kali linux distribution, having access to 28,5 Gb of RAM.

6.1 Benchmarking and analyzing distributed decryption

MP-SPDZ provides a lot of options in terms of MPC protocols that can be used. In our case, we are, of course, interested in protocols that work modulo a prime, so we can use the q prime as the modulus, so we get reduction modulo q implicitly in all computations.

One of the dishonest majority MPC protocols with malicious security that MP-SPDZ supports is the MASCOT protocol [KOS16]. If we use MASCOT as the MPC protocol, then using a small prime such as the Kyber modulus 3329 makes the field too small for any reasonable security parameter. We can instead use the MPC protocol that MP-SPDZ denotes by Mama, which is MASCOT with several MACs to allow increasing the security parameter to a multiple of the prime length. Therefore by using Mama, we can get reasonable security in the malicious dishonest majority setting. The amount of MACs used for this protocol is adjusted to be the least amount needed in order to obtain the given security level (see mama-party.cpp). For the benchmarks we use the MP-SPDZ default statistical security parameter of 40. A note here is that this means that we are using the SPDZ online phase mentioned earlier, which also implies that we are in the full-threshold setting.

For the binary domain MPC, we use the binary secret sharing-based MPC protocol that MP-SPDZ calls Tinier, which is what MP-SPDZ defaults to in the malicious, dishonest majority setting. This protocol is described in "A Unified Approach to MPC with Preprocessing using OT" by Frederiksen et al. [FKOS15].

Note that due to a lack of computing power, we have seen it necessary to benchmark the MP-SPDZ code for only two parties to limit the amount of processing power being used.

6.1.1 Running Kyber KEM distributed decryption

To run the implemented distributed decryption algorithm for the Kyber KEM, execute the following:

1. Install MP-SPDZ as described at `https://mp-spdz.readthedocs.io/en/latest/readme.html`.
2. The release or source code needs to be from after 3. April, since we use functionality introduced in commit "0bf7ad8" from this particular date.
3. Download the Basic Circuit File Keccak_f.txt file from `https://homes.esat.kuleuven.be/~nsmart/MPC/` and place the file in the "...\Programs\Circuits" folder. This is necessary to be able to use the sha3_256 implementation in the high-level interface of MP-SPDZ since it relies on a circuit in the Bristol Fashion format.

4. Download the kyber_ddec.mpc file from the ThresholdKyber repository linked to earlier.

5. Place the downloaded kyber_ddec.mpc file in "...\Programs\Source" of the MP-SPDZ folder.

6. Pick one of the test vectors from the "ThresholdKyber\ddec\test_vectors_ddec" folder of the ThresholdKyber repository. The kyber_ddec file contains the public input and must be placed in the "Programs\Public-Input" folder of the MP-SPDZ directory. The Input-P0-0 file contains the kyber key s and must be placed in the "Player-Data" folder. The top of the kyber_ddec file defines the $r = k$ and η parameters, which must be adjusted to correspond to the test vector picked. In this case, we just let the first player input s for testing, while in practice, one would either generate s in a distributed manner or have a trusted dealer generate and share it.

7. Execute the commands

```
./compile.py kyber_ddec -Y -P 3329
./Scripts/mama.sh kyber_ddec.mpc -v
```

8. The key printed to the terminal should match the one in the expected_output file of the corresponding test vector.

6.1.2 Analysis of offline phase for Kyber and Gladius DDec

By using the verbose -v argument when running a compiled .mpc program, we can get the exact number of preprocessing material consumed during program execution. When doing this, we end with the following numbers for the three main Kyber variants Kyber512, Kyber768, and Kyber1024.

Parameter set	Integer triples	Integer bits	daBits	edaBits	Input tuples	Bit Triples
Kyber512	121772	15156	6912	0	768	345600
Kyber768	145396	18732	7936	0	1024	345600
Kyber1024	169740	22740	9984	0	1280	422400

As expected, when we increase the value of the k parameter as in Kyber768 and Kyber1024, the number of integer triples, integer bits, and daBits increases. This makes sense since we increase the dimensions of the vectors and matrices of polynomials that we use, meaning that we end up with more arithmetic operations and, in some cases, arithmetic-binary domain conversions. This results in the protocol becoming more expensive, and it, therefore, has a higher communication cost.

The relationship between the number of input tuples and the k parameter is a bit more obvious, so we can theoretically verify that these are as expected. Here, the private input consists of the polynomial vector s, with $n \cdot k$ coefficients in total and the size 256 bit-string z. This means that for Kyber512 we expect $256 \cdot 2 + 256 = 768$ input tuples, for Kyber768 we expect $256 \cdot 3 + 256 = 1024$, and lastly we expect $256 \cdot 4 + 256 = 1280$ for Kyber1024. This is also what we see in practice. Note, however, that in practice, one would want to generate s and z through distributed key generation, in which case they are not input by a single party.

At first glance, the number of bit triples consumed may seem strange since it does not increase between Kyber512 and Kyber768 but increases when using Kyber1024. These bit triples are used for computing AND gates when working in the binary MPC domain, as done here when hashing. Theoretically, this does make sense, however, as we do $2k + 3$ hash functions calls, but the Kyber parameter η_1 decreases from 3 to 2 between Kyber512 and Kyber768, which means that the output length used for SHAKE256 decreases in some cases.

This results in 1 instead of 2 applications of the Keccak-f circuit for each PRF_{η_1} call, and thus also fewer AND gates computed. This turns out to exactly cancel out the increase in bit triples we get from incrementing k.

For Gladius, we get the following amount of preprocessing material of each type.

Parameter set	Integer triples	Integer bits	daBits	edaBits	Input tuples	Bit Triples
Gladius-Hisp. 1	4769062	61173	512	0	942841	38400
Gladius-Hisp. 2	5192161	64512	512	0	1048576	38400

Here the number of integer triples is significantly higher than for the Kyber KEM. This makes sense since we use regular LWR and not Module-LWR, so we have to do a lot of multiplications for computing Vector-Matrix products in the KEM decapsulation and Validity Check phases. The number of bit triples is a lot lower, which accurately reflects the fact that we have to do fewer binary domain computations due to only needing one distributed hash function call. For the Kyber KEM, we have to do $2k + 3$ distributed hash function calls, and so it makes sense that Gladius would need a lot fewer bit triples since these are used for the AND gates for the Keccak-f circuit evaluations. The low number of daBits is also due to fewer arithmetic-binary domain conversions from the low number of hash function calls. Again, we are working in a small field like for Kyber, so MP-SPDZ doesn't implement the domain conversions using edaBits, but only daBits, hence no edaBits are being generated. The higher number of integer bits compared to the Kyber KEM is likely due to a large number of arithmetic-domain comparisons having to be performed in the Validity Check phase.

6.1.3 Benchmarking the online phase of Distributed Decryption

When using MASCOT with multiple MACs as the MPC protocol, then we get the following results for the distributed decryption procedure when measuring the online phase exclusively. As can be seen, for the DDec runtime, we once again see the same pattern as for the number

Parameter set	Online time DDec	Online Bandwidth DDec
Kyber512	10.3326 sec	321.988 MB
Kyber768	10.6675 sec	322.405 MB
Kyber1024	12.9692 sec	400.017 MB

of bit triples generated in preprocessing. Namely, the runtime and bandwidth increase less between Kyber512 and Kyber768 than they do between Kyber768 and Kyber1024. This suggests that the number of Keccak-f circuits evaluated, and thus distributed hash function calls, plays a significant role in how fast the online phase of the protocol is. This likely also indicates that the number of hash function calls is indeed a bottleneck in terms of online phase runtime. To verify whether this is the case, we benchmarked the online time of Kyber DDec, where we replaced the distributed hash function calls and arithmetic-binary conversions needed for them with hardcoded constants. This resulted in an online time of 0.94886 seconds and 1.95987 MB online bandwidth. Taking this a step further, we then independently benchmarked the symmetric primitives of Kyber instantiated by hash functions. These can be seen below for the case of Kyber512.

Primitive	Hash function	Online time	Online bandwidth
G	SHA3-512	1.00722 sec	35.4292 MB
PRF_{η_1}	SHAKE-256 w/ output length $512 \cdot \eta_1$	2.05096 sec	70.7356 MB
PRF_{η_2}	SHAKE-256 w/ output length $512 \cdot \eta_2$	0.982395 sec	35.4229 MB
KDF	SHAKE-256 w/ output length 256	1.01923 sec	35.4292 MB

For Kyber512 distributed decryption we do one G call, two PRF_{η_1} calls, three PRF_{η_2} calls, and one KDF call. This results in a total time of

$$1.00722 + 2 \cdot 2.05096 + 3 \cdot 0.982395 + 1.01923 = 9.075555 \text{ sec}$$

From these experiments, it is clear that the distributed hashing is indeed the main bottleneck in the protocol and implementation. This result matches the theoretical considerations outlined in Section 4.1, where it was noted that hashing is hard to do inside MPC.

For Gladius-Hispaniensis, we get the following online phase results.

Parameter set	Avg. time DDec	Bandwidth DDec
Gladius-Hispaniensis 1	9.51517 sec	128.045 MB
Gladius-Hispaniensis 2	10.0773 sec	135.664 MB

For the parameter sets tested, the distributed decryption protocol for Gladius-Hispaniensis is thus faster and consumes less bandwidth than the distributed decryption protocol for the Kyber KEM. Still, Kyber might also be of particular interest due to it having been standardized by NIST for post-quantum usage. An important thing to note here is that using Gladius-Pompeii, which is based on Module-LWR, should give a smaller construction and would allow employing NTTs to speed up computations since the parameter sets for Gladius-Pompeii are specifically picked to be NTT-friendly. Another interesting point here is that the Gladius paper mentions that the validity check is actually the bottleneck in the Gladius-Hispaniensis distributed decryption, despite it using no hashing at all. This does, however, have to be seen in light of the fact that in the Gladius paper, they use a hash function combining Rescue and SHA3-256 to improve the efficiency of distributed hashing.

For the experiments, we also measured our distributed key generation for Kyber. This

Parameter set	Online time DKeyGen	Online Bandwidth DKeyGen
Kyber512	0.0145232 sec	0.01244 MB
Kyber768	0.0227492 sec	0.018584 MB
Kyber1024	0.014716 sec	0.024728 MB

procedure is very fast compared to the others. This is expected since the players can privately sample from CBD outside the MPC protocol, meaning that no PRF calls have to be performed to generate input for the CBD function. Therefore we end up with no distributed hash function calls, so the main bottleneck of DDec is not present here. As mentioned earlier, the downside of this is that if we add together the private key polynomial vectors s and e generated as normal, then we end up with too much noise. This would then result in the need for a larger modulus, which would mean that our DDec implementation is not compatible with DKeyGen, due to the modulus $q = 3329$ being implicit in the NTT implementation. Furthermore, we also do not have any zero-knowledge proof ensuring that the input is honestly generated. Still, the implementation is interesting as a proof of concept.

6.2 Benchmarking Golang TKyber implementation

Benchmarking our implementation of IND-CPA TKyber was done using Go's built-in benchmarking tool, which is a part of the "testing" package.

We here use the following parameter sets, where $\delta = 1$ and $\lambda = 100$ for all sets. We choose Kyber1024 since $k = 4$ is the smallest value used in the parameter sets presented in 3.1 and 3.2.

The benchmarking results when using these parameter sets then look as follows.

For the 5 first parameter sets it seems that the runtimes do not change considerably between the different variants. This is likely due to the low amount of participating parties.

Parameter set	LSS Scheme	Kyber variant	parties	t
Kyber1024-Add-3-2	Additive	Kyber1024	3	2
Kyber1024-Rep-3-2	Replicated	Kyber1024	3	2
Kyber1024-Rep-3-1	Replicated	Kyber1024	3	1
Kyber1024-Nai-3-2	Naive	Kyber1024	3	2
Kyber1024-Nai-3-1	Naive	Kyber1024	3	1
Kyber1024-Add-5-4	Replicated	Kyber1024	5	4
Kyber1024-Rep-5-2	Replicated	Kyber1024	5	2
Kyber1024-Nai-5-2	Naive	Kyber1024	5	2

Parameter set	Setup	Enc	PartDec	Combine
Kyber1024-Add-3-2	0.106 ms	0.0877 ms	0.0175 ms	0.00594 ms
Kyber1024-Rep-3-2	0.106 ms	0.0866 ms	0.0395 ms	0.00644 ms
Kyber1024-Rep-3-1	0.106 ms	0.0873 ms	0.0393 ms	0.00620 ms
Kyber1024-Nai-3-2	0.106 ms	0.0881 ms	0.0175 ms	0.00636 ms
Kyber1024-Nai-3-1	0.120 ms	0.0866 ms	0.0436 ms	0.00627 ms
Kyber1024-Add-5-4	0.132 ms	0.0862 ms	0.0171 ms	0.00629 ms
Kyber1024-Rep-5-2	0.208 ms	0.0866 ms	0.119 ms	0.00853 ms
Kyber1024-Nai-5-2	0.358 ms	0.0848 ms	0.131 ms	0.00816 ms

We then test additive secret sharing with $n = 5$, which results in an increase in the runtime of Setup as expected since we need to generate more shares. In the last two parameter sets, we test Replicated and Naive with 5 parties instead of just 3 while still keeping the threshold parameter $t = 2$. For the case of Kyber1024-Rep-5-2 we have to compute an additive sharing of size $\binom{n}{t} = 10$, where $L = \binom{n-1}{t} = 6$ shares then have to be distributed to each party. This means that we compute a larger sharing, which is more expensive. Computing which parties should receive which shares is also more expensive. For PartDec, each of the parties also has to do operations on more shares, which means that this also becomes slower. This explains the significant slowdown compared to the variants with $n = 3$. For Kyber1024-Nai-5-2, the slowdown is exacerbated since using Naive LSS with these parameters results in $\binom{n}{t+1} = 10$ different additive sharings having to be computed.

For these benchmarks, it is also important to remember that we did not manage to change the modulus in the underlying Kyber package used, as mentioned earlier. To accommodate for the lower modulus, we had to decrease the amount of noise used to be able to still decrypt. What the lower modulus means is that our benchmarks will be faster than what would otherwise have been the case. We expect that a slowdown would occur when increasing the modulus, but we do not expect this to change the comparisons to the distributed decryption in any major way since TKyber is significantly faster as is.

6.2.1 Comparing against plain Kyber

For benchmarking regular Kyber, we use the kyber-k2so implementation used for TKyber. The parameter sets benchmarked were Kyber512, Kyber768, and Kyber1024 in Figure **??**.

From the above data, it can be seen that the additional secret sharing done in Setup of TKyber results in a significant overhead compared to regular Kyber key generation. There is also a small overhead for Encryptdue to hashing using F and G and doing δ IND-CPA Kyber encryptions where $\delta = 1$ in this case. The IND-CPA Kyber Decrypt is also faster than using PartDec and Combine in IND-CPA TKyber. This makes sense since we have to do δ OW-CPA TKyber partial decryptions, and for each of these, a particular party will have to do regular decryption for each secret key share, along with sampling and adding

Parameter set	KeyGen	Encrypt	Decrypt
Kyber512	27504 ns	30589 ns	7630 ns
Kyber768	48658 ns	49365 ns	9886 ns
Kyber1024	74743 ns	76970 ns	12585 ns

Table 6.1: Average running times for plain Kyber with $q = 3329$

noise. Then recombining the partial decryption shares using the LSS scheme will also slow the implementation down further.

6.3 Comparing TKyber and Distributed Decryption

When comparing TKyber and the Kyber KEM DDec protocol for doing threshold decryption, we see that the DDec protocol is significantly slower, likely due to the large overhead associated with doing operations inside an MPC protocol.

For the security of the two schemes, it is important to note that the TKyber TPKE scheme is only IND-CPA secure, whereas the Kyber KEM is IND-CCA2 secure. Concretely, this means that in some applications, such as for example TLS servers, it might be enough to use an IND-CPA TPKE, while in other applications, something stronger might be needed. So for some applications, one might disregard the efficiency limitations of the distributed decryption, for example, in order to get IND-CCA2 security.

As for the techniques used themselves, the construction used for the TKyber scheme is specifically designed for LWE schemes, as it requires, among other things, a public-key encryption (PKE) scheme with linear decryption. The technique presented in the Gladius paper, which we used for the Kyber KEM, is more generic and does not put such limitations on the encryption scheme used. It only requires that we can carry out the operations required for decryption inside an MPC protocol.

Section 7

Conclusion

This work examined different means of attaining threshold decryption for the post-quantum KEM Kyber, one of the algorithms recently standardized by NIST for post-quantum usage. The goal of the thesis was then to answer the question:

> *"Is it possible to adapt and implement a distributed decryption procedure for threshold decryption using Kyber, similar to the one for Gladius by Cong et al.?* [CCMS21]. *If it is possible, how does it compare to a specialized approach for threshold decryption from LWE by Boudgoust et al.* [BS23]."

To be able to answer this question, we first implemented the TKyber construction from "Simple Threshold (Fully Homomorphic) Encryption From LWE With Polynomial Modulus" by Boudgoust et al. [BS23]. Following this, we used the techniques from "Gladius: LWR based efficient hybrid public key encryption with distributed decryption" by Cong et al. [CCMS21] to construct distributed decryption and key generation protocols for the Kyber KEM. In addition, we implemented these protocols, along with a variant of Gladius called Gladius-Hispaniensis, in the MPC framework MP-SPDZ. We then benchmarked the implementations of TKyber and distributed decryption for Kyber and Gladius-Hispaniensis, to compare these against each other. For distributed decryption using Kyber and Gladius-Hispaniensis, this included analyzing the amount of preprocessing material generated in the offline phase, as well as the time and bandwidth used in the online phase. For TKyber, this involved measuring the run time of the Setup, Enc, PartDec, and Combine algorithms.

From our benchmarks, we found that the online phase of distributed decryption using Kyber takes 10.3326, 10.6675, and 12.9692 seconds for Kyber512, Kyber768, and Kyber1024, respectively. We also found that the online phase of our implementation of distributed decryption for Gladius-Hispaniensis took 9.5152 and 10.0773 seconds for the two-parameter sets tested. When using TKyber, we found that for the parameter sets tested, Setup uses between 0.106 and 0.358 ms, Enc uses between 0.0848 and 0.0881 ms, PartDec uses between 0.0171 and 0.131 ms and finally Combine uses between 0.00594 and 0.00853 ms.

From our findings, we can answer yes to the first part of the research question, it is possible to use the techniques from Gladius to implement a distributed decryption algorithm for Kyber. For the second part of the question, we found that TKyber is significantly faster than the implementation of distributed decryption for Kyber and that the bottleneck in Kyber DDec was distributed hashing. This suggests that for distributed decryption for Kyber, improving the efficiency of distributed hashing could result in a large improvement in runtime. On the other hand, as mentioned previously, the slowest phase in Gladius is the validity check, which does not use hashing at all. An additional difference between TKyber and DDec for Kyber is that TKyber is an IND-CPA TPKE, while DDec for Kyber provides an IND-CCA2 KEM. The implication of this is that in some applications, such as for a TLS server, it might be enough to use an IND-CPA scheme, while in other applications, an IND-CCA2 secure scheme might be necessary. Therefore one might be favorable over the other without regard for efficiency.

One of the main limitations with regard to the benchmarks performed was that we did not manage to change the modulus of the TKyber implementation, so we had to resort to using

the default modulus of $q = 3329$. The issue here is that this does not allow enough space for the noise needed, so we had to decrease the standard deviation of the noise distribution for decryption to work. We hypothesize that increasing the modulus will slow the implementation down somewhat, but that it will still be significantly faster than DDec for Kyber. A second limitation is that we had to run the DDec protocol for Kyber for two parties only, as the benchmark PC had an insufficient amount of RAM for more players. However, we expect the results to change by a very small margin as two parties were used consistently for all MP-SPDZ benchmarks.

The main impact of this work is to provide a protocol for distributed decryption using Kyber, along with an implementation using MP-SPDZ of said protocol. In addition to this, we also provide an implementation and benchmarks of Gladius-Hispaniensis using MP-SPDZ. We also provide an implementation of TKyber in Go, along with benchmarks and comparisons to the aforementioned distributed decryption protocol for Kyber.

7.0.1 Future work

Here we present a few ideas for future work.

Optimizing MP-SPDZ implementation

Currently, MP-SPDZ seems to draw only on daBits for doing the binary-arithmetic domain conversions, but as D. Escudero et al. have shown [EGK+20], it is more efficient to use edaBits when converting between binary and arithmetic domains.

This can be seen in commit $0bf7ad8$ where one of the authors of MP-SPDZ implemented a way of converting secret integers in $\{0, 1\}$ into secret bits for small fields, such as the one we are using due to an issue we raised on the GitHub repository.

Making Kyber rigid

If Kyber can be changed to satisfy the rigidity requirement from Section 4.1.3, we could avoid doing the re-encryption from the Fujisaki-Okamoto transformation, which in turn, avoids some of the distributed hashing by instead using the variant of the Cramer-Shoup transformation used by Gladius. This should improve the efficiency of the protocol significantly, as argued earlier.

MPC-friendly Kyber variant

Additionally, MPC-friendly hash functions such as Rescue could be introduced to speed up the implementation, although this would break compatibility with standard Kyber. To do this, a new instantiation of the symmetric primitives could be made. We assume that such a change would decrease the runtime since these hash functions are specifically designed for MPC, as opposed to those from the FIPS-202 standard used in the modern instantiation of Kyber.

Allow changing modulus in TKyber

Currently, we use the kyber-k2so package for our TKyber implementation. As explained earlier, we did not manage to change the modulus of kyber-k2so, which is an issue for our TKyber implementation. Allowing different moduli for TKyber through either modifying kyber-k2so or using an entirely different library.

Bibliography

[AABS+20] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Transactions on Symmetric Cryptology*, pages 1–45, 2020.

[ACC+18] Abdelrahaman Aly, Karl Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P Smart, Titouan Tanguy, et al. SCALE-MAMBA, 2018. `https://homes.esat.kuleuven.be/~nsmart/SCALE/`.

[AOR+19] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. *IACR Cryptol. ePrint Arch.*, page 974, 2019.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings 30*, pages 169–188. Springer, 2011.

[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`.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[Div14] NIST Computer Security Division. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Number 202, May 2014.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*, pages 1–18. Springer, 2013.

[EGK+20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. Cryptology ePrint Archive, Paper 2020/338, 2020. `https://eprint.iacr.org/2020/338`.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to mpc with preprocessing using ot. Cryptology ePrint Archive, Paper 2015/901, 2015. https://eprint.iacr.org/2015/901.

[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.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III*, pages 158–189. Springer, 2018.

[LD21] John M. Schanck Léo Ducas. Kyber parameter security estimator, March 2021. https://github.com/pq-crystals/security-estimates.

[LPR12] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Paper 2012/230, 2012. https://eprint.iacr.org/2012/230.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[NO07] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Public Key Cryptography–PKC 2007: 10th International Conference on Practice and Theory in Public-Key Cryptography Beijing, China, April 16-20, 2007. Proceedings 10*, pages 343–360. Springer, 2007.

[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.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery.

[RST+19] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. *IACR Cryptol. ePrint Arch.*, page 1300, 2019.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, oct 1997.

Section A

Kyber algorithms

Algorithm 1 $\mathsf{Parse}\colon \mathcal{B}^* \to R_q^n$

Input: Byte stream $B = b_0, b_1, b_2 \cdots \in \mathcal{B}^*$
Output: NTT-representation $\hat{a} \in R_q$ of $a \in R_q$

$i \coloneqq 0$
$j \coloneqq 0$
while $j < n$ **do**
  $d_1 \coloneqq b_i + 256 \cdot (b_{i+1} \bmod^+ 16)$
  $d_2 \coloneqq \lfloor b_{i+1}/16 \rfloor + 16 \cdot b_{i+2}$
  if $d_1 < q$ **then**
    $\hat{a}_j \coloneqq d_1$
    $j \coloneqq j + 1$
  end if
  if $d_2 < q$ **and** $j < n$ **then**
    $\hat{a}_j \coloneqq d_2$
    $j \coloneqq j + 1$
  end if
  $i \coloneqq i + 3$
end while
return $\hat{a}_0 + \hat{a}_1 X + \cdots + \hat{a}_{n-1} X^{n-1}$

Algorithm 2 $\mathsf{CBD}_\eta\colon \mathcal{B}^{64\eta} \to R_q$

Input: Byte array $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
Output: Polynomial $f \in R_q$

$(\beta_0, \ldots, \beta_{512\eta-1}) \coloneqq \mathsf{BytesToBits}(B)$
for i from 0 to 255 **do**
  $a \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
  $b \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
  $f_i \coloneqq a - b$
end for
return $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$

Algorithm 3 $\mathsf{Decode}_\ell\colon \mathcal{B}^{32\ell} \to R_q$

Input: Byte array $B \in \mathcal{B}^{32\ell}$
Output: Polynomial $f \in R_q$

$(\beta_0, \ldots, \beta_{256\ell-1}) \coloneqq \mathsf{BytesToBits}(B)$
for i from 0 to 255 **do**
  $f_i \coloneqq \sum_{j=0}^{\ell-1} \beta_{i\ell+j} 2^j$
end for
return $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$

Algorithm 4 KYBER.CPAPKE.KeyGen(): key generation

Output: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$

$d \leftarrow \mathcal{B}^{32}$
$(\rho, \sigma) \coloneqq \mathrm{G}(d)$
$N \coloneqq 0$
for i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  for j from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}[i][j] \coloneqq \mathsf{Parse}(\mathsf{XOF}(\rho, j, i))$
  end for
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{s} \in R_q^k$ from B_{η_1}
  $\mathbf{s}[i] \coloneqq \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
  $N \coloneqq N + 1$
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e} \in R_q^k$ from B_{η_1}
  $\mathbf{e}[i] \coloneqq \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
  $N \coloneqq N + 1$
end for
$\hat{\mathbf{s}} \coloneqq \mathsf{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} \coloneqq \mathsf{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} \coloneqq \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
$pk \coloneqq (\mathsf{Encode}_{12}(\hat{\mathbf{t}} \bmod^+ q) \| \rho)$ ▷ $pk \coloneqq \mathbf{As} + \mathbf{e}$
$sk \coloneqq \mathsf{Encode}_{12}(\hat{\mathbf{s}} \bmod^+ q)$ ▷ $sk \coloneqq \mathbf{s}$
return (pk, sk)

Algorithm 5 KYBER.CPAPKE.Enc(pk, m, r): encryption

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Random coins $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

$N \coloneqq 0$
$\hat{\mathbf{t}} \coloneqq \mathsf{Decode}_{12}(pk)$
$\rho \coloneqq pk + 12 \cdot k \cdot n/8$
for i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  for j from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}^T[i][j] \coloneqq \mathsf{Parse}(\mathsf{XOF}(\rho, i, j))$
  end for
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{r} \in R_q^k$ from B_{η_1}
  $\mathbf{r}[i] \coloneqq \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, N))$
  $N \coloneqq N + 1$
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e}_1 \in R_q^k$ from B_{η_2}
  $\mathbf{e}_1[i] \coloneqq \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
  $N \coloneqq N + 1$
end for
$e_2 \coloneqq \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$ ▷ Sample $e_2 \in R_q$ from B_{η_2}
$\hat{\mathbf{r}} \coloneqq \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} \coloneqq \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ $\mathbf{u} \coloneqq \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
$v \coloneqq \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$ ▷ $v \coloneqq \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
$c_1 \coloneqq \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
$c_2 \coloneqq \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
return $c = (c_1 \| c_2)$ ▷ $c \coloneqq (\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$

Algorithm 6 KYBER.CPAPKE.Dec(sk, c): decryption

Input: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Message $m \in \mathcal{B}^{32}$

$\mathbf{u} \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
$v \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
$\hat{\mathbf{s}} \coloneqq \mathsf{Decode}_{12}(sk)$
$m \coloneqq \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1))$ ▷ $m \coloneqq \mathsf{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1))$
return m

Algorithm 7 KYBER.CCAKEM.KeyGen()

Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8 + 32}$
Output: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$

$z \leftarrow \mathcal{B}^{32}$
$(pk, sk') \coloneqq$ KYBER.CPAPKE.KeyGen()
$sk \coloneqq (sk' \| pk \| \mathrm{H}(pk) \| z)$
return (pk, sk)

Algorithm 8 KYBER.CCAKEM.Enc(pk)

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8 + 32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Shared key $K \in \mathcal{B}^*$

$m \leftarrow \mathcal{B}^{32}$
$m \leftarrow \mathrm{H}(m)$ ▷ Do not send output of system RNG
$(\bar{K}, r) \coloneqq \mathrm{G}(m \| \mathrm{H}(pk))$
$c \coloneqq$ KYBER.CPAPKE.Enc(pk, m, r)
$K \coloneqq \mathsf{KDF}(\bar{K} \| \mathrm{H}(c))$
return (c, K)

Algorithm 9 KYBER.CCAKEM.Dec(c, sk)

Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Input: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$
Output: Shared key $K \in \mathcal{B}^*$

$pk \coloneqq sk + 12 \cdot k \cdot n/8$
$h \coloneqq sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
$z \coloneqq sk + 24 \cdot k \cdot n/8 + 64$
$m' \coloneqq$ KYBER.CPAPKE.Dec$(\mathbf{s}, (\mathbf{u}, v))$
$(\bar{K}', r') \coloneqq \mathrm{G}(m' \| h)$
$c' \coloneqq$ KYBER.CPAPKE.Enc(pk, m', r')
if $c = c'$ **then**
    return $K \coloneqq \mathsf{KDF}(\bar{K}' \| \mathrm{H}(c))$
else
    return $K \coloneqq \mathsf{KDF}(z \| \mathrm{H}(c))$
end if
return K