From Middle-Earth to the Galaxy: SMAUG vs. Kyber

Correa Merlino, Jorge

Award date:
2023

From Middle-Earth to the Galaxy: SMAUG vs. Kyber

Jorge Correa Merlino
1629883

Supervisor
dr. Sven Schäge

Eindhoven, August 2023

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Applied Mathematics
Coding and Cryptography

Abstract

The goal of this report is to provide a comparison between two state-of-the-art lattice-based encryption algorithms in terms of efficiency and security. The report is largely self-contained providing all the background information on understanding the design rationale. The goal is accomplished by examining a recent proposal submitted to the South Korean competition (SMAUG) and comparing it to the NIST competition winner (Kyber). The key generation security of both SMAUG and Kyber rely on the MLWE assumption. However, SMAUG relies on MLWR for the security of its encryption system while Kyber relies on MLWE. This allows SMAUG to perform faster operations when encrypting and decrypting data. Moreover, SMAUG decides to use a sparse secret key, which reduces the amount of storage required for the key. This is also useful for some efficiency gains in polynomial multiplication, but it is still slower than how Kyber does it. The reason for this is that Kyber can benefit from using NTT while SMAUG cannot, and this is because of how both schemes choose the parameter q. However, SMAUG has some other benefits from its choice of q since being a power of two means that it is friendly with scaling and rounding operations, leading to some efficiency gains over Kyber in other areas, such as during encryption.

Contents

1 Introduction

The arrival of quantum computing promises to revolutionize many fields and sectors, one of which is cryptography. Quantum computers have unmatched computational capacity and are capable of cracking several widely-used encryption algorithms as conventional computing capabilities continue to hit their limits. This threat has prompted significant concerns over the long-term security of digital communications, which gave birth to the NIST (National Institute of Standards and Technology) Post-Quantum Cryptography Standardization Project [24].

The NIST competition marks a pivotal moment in the field of cryptography, as researchers and cryptographers worldwide join forces designing, evaluating, and standardizing post-quantum cryptographic algorithms. These novel algorithms aim to provide robust security against quantum adversaries, ensuring that sensitive information remains safeguarded in a quantum-powered world. As part of this competition, numerous post-quantum cryptographic candidates from different families were submitted for evaluation. Researchers presented schemes with different approaches, including lattice-based, code-based, hash-based, and multivariate polynomial-based, among others. Each candidate underwent rigorous scrutiny over the last 6 years, with NIST analyzing their security, efficiency, and practicality to ensure they could be used in real-world applications. Among the notable candidates, lattice-based cryptographic schemes such as Kyber [5] and Saber [6] offered strong security guarantees and efficiency.

However, this would not be the only competition considering such schemes, since South Korea launched their own post-quantum competition in 2022 through the Center for Quantum Resistant Cryptography [22]. The name of this competition is the Korean post-quantum Competition (KpqC) and their goal is to develop world-class quantum resistant cryptography by expanding their domestic technological facilities and enhancing competitiveness. While some candidates were quickly proven to be insecure, some other candidates show promise, one of which is SMAUG, and this scheme is the one in which this report is based in.

The goal of this report is to provide a comprehensive understanding of post-quantum cryptography, with a focus on lattice-based problems. This is accomplished by examining a recent proposal submitted to the South Korean competition and comparing it to the NIST competition winner. The report is structured as follows: It starts providing the reader with preliminary information to help them understand the topics that will be discussed later on. Then, a thorough explanation of SMAUG is provided, as well as for Kyber, the NIST competition winner. Afterwards, a section dedicated to the comparison of the differences in the schemes is presented. Finally, the security proofs for the core element of both schemes are produced, with some common attacks described.

2 Preliminaries

In this section, the concepts used in the implementation of SMAUG [10] will be explained.

2.1 Rings

A ring is defined by a set $\mathcal{R}$ together with two operations, addition $(+)$ and multiplication $(\cdot)$. It satisfies the following properties:

1. $[\mathcal{R}; +]$ is commutative under addition. In other words, for all a, b in $\mathcal{R}$, $a + b = b + a$.
2. Multiplication is associative on $\mathcal{R}$. This means that for all a, b, c in $\mathcal{R}$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ holds.
3. Multiplication is distributive over addition. That is, for all $a, b, c \in \mathcal{R}$, $a \cdot (b + c) = a \cdot b + a \cdot c$, and $(b + c) \cdot a = b \cdot a + c \cdot a$.

In this report, the interest lies in the quotient ring of integer polynomials modulo q. This ring will be defined incrementally.

First consider the ring of integers, $[\mathbb{Z}; +, \cdot]$, then this ring is infinite since all integers fulfill the properties above. Next, consider the ring of integers modulo q, $[\mathbb{Z}_q; +_q, \times_q]$ then this ring only contains integers in $[0, q-1]$. Note that the addition and multiplication operations are done modulo q, so if an operation results in a number exceeding q, it is subtracted by q until it is in $[0, q-1]$.

The next ring to consider is the integer polynomial ring $[\mathbb{Z}[X]; +, \cdot]$, where the operators are the polynomial addition and multiplication operators. This ring can also be defined as an integer polynomial ring modulo q, $[\mathbb{Z}_q[X]; +_q, \times_q]$. Then this ring describes all the polynomials with coefficients in $[0, q-1]$, with addition and multiplication still being polynomial but the resulting coefficients are done modulo q.

Finally, the ring of interest can be explained. Let the quotient ring of integer polynomials modulo q be $[\mathbb{Z}_q[X] \setminus \{x^n + 1\}, +^*, \times^*]$. Then in this ring, the polynomials are no larger than degree $n - 1$. This can be better illustrated with an example. Let $q = 3$ and $n = 2$ then the elements are:

$$0, 1, 2, x, x + 1, x + 2.$$

Larger degree polynomials can be reduced to take this form in the following way: take $x^3 + x + 1$. Then, this polynomial is reduced with the quotient $x^2 + 1$. Since $x^2 + 1 \equiv 0$ in this ring, then it can be multiplied with appropriate terms to cancel out larger degrees. For example:

$$\begin{aligned} x^3 + x + 1 &\equiv x^3 + x^2 + x + 3 - x \cdot (x^2 + 1) \\ &\equiv x^3 + x^2 + x + 3 - x^3 - x \\ &\equiv x^2 + 3 \\ &\equiv x^2 + 3 - (x^2 + 1) \\ &\equiv 2. \end{aligned}$$

Therefore, $x^3 + x^2 + x^3 \equiv 2$ in this ring. If the addition and multiplication operators are obvious, rings are usually referred to as the set that defines them.

2.2 Notation

Matrices are represented with bold and upper case letters $\boldsymbol{A}$ and vectors with bold and lower case letters $\boldsymbol{b}$. A polynomial ring $\mathcal{R}$ is defined as $\mathcal{R} = \mathbb{Z}[X]/\{x^n + 1\}$. The quotient ring by $\mathcal{R}_q = \mathbb{Z}[X]/\{q, x^n + 1\} = \mathbb{Z}_q[X]/\{x^n + 1\}$ for a positive integer q. For an integer η, the set of polynomials of degree less than n with coefficients in $[-\eta, \eta] \in \mathbb{Z}$ is defined as $\mathcal{S}$.

2.3 Lattices

A lattice $\mathcal{L}$ is the set of all integer linear combinations of (linearly independent) basis vectors. Basis vectors are a collection of vectors that can be used to reproduce any point in the lattice. Lattices can be visualized as a grid or a repeating pattern of points that extend infinitely in all directions. Figure 1 shows a visualization of some example lattices. Another way to define a lattice is as a discrete additive subgroup of $\mathbb{R}^n$.

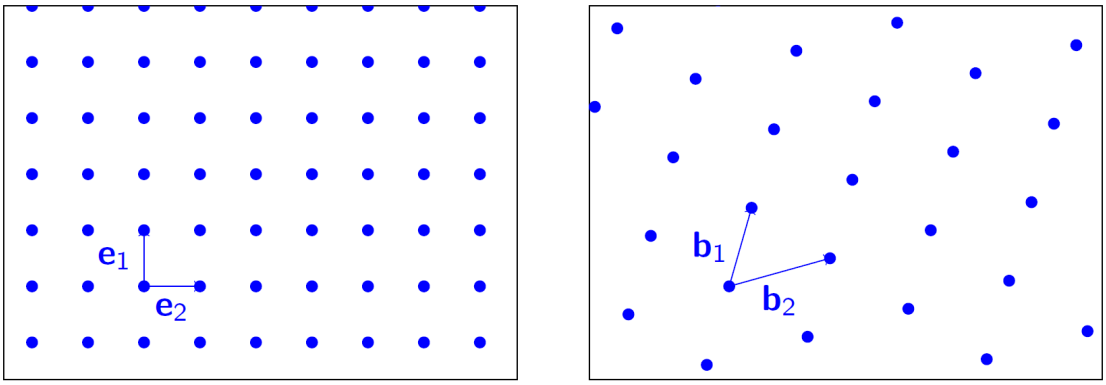

Figure 1: Examples of two different lattices with their basis vectors

The distance between points can be calculated using Euclidean distance, even the points that are not part of the lattice (the points that cannot be reached with the basis vectors). Therefore the distance between two points $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ and $\boldsymbol{y} = (y_1, \ldots, y_n) \in \mathbb{R}^n$ is calculated in the following way:

$$\begin{aligned} d(\boldsymbol{x}, \boldsymbol{y}) &= \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \\ &= \|\boldsymbol{x} - \boldsymbol{y}\| \end{aligned}$$

The most interesting feature of a lattice in cryptography is the shortest vector in it. The shortest distance between the origin of the lattice and the closest point to it is called the minimum distance and is defined as:

$$\lambda = \min_{\boldsymbol{x} \in \mathcal{L}, \boldsymbol{x} \neq 0} \|\boldsymbol{x}\|$$

Any vector in lattice $\mathcal{L}$ of length λ is a shortest vector. It turns out that finding a shortest vector for a given lattice is computationally hard and no methods are known that find them efficiently. This

problem is known as the Shortest Vector Problem (SVP) which is to, given a lattice $\mathcal{L}(\boldsymbol{B})$ with basis $\boldsymbol{B}$, find a (nonzero) lattice vector $\boldsymbol{Bx}$ with $\boldsymbol{x} \in \mathbb{Z}^k$ of length at most $\|\boldsymbol{Bx}\| \leq \lambda$.

Another special type of lattice is the dual lattice. The dual of a lattice $\mathcal{L}$ is the set of all vectors $x \in \operatorname{span}(\mathcal{L})$ such that $\langle x, v \rangle \in \mathbb{Z}$ for all $v \in \mathcal{L}$. They are useful since if the SVP can be solved in the dual lattice, the vector that is found can be transformed back to the original lattice. Therefore, with a carefully selected dual lattice, computations can be made easier than if they happened on the original one.

2.4 Gram-Schmidt Process

The Gram–Schmidt process is a method for orthonormalizing a set of vectors in an inner product space, the most common choice being $\mathbb{R}^n$. Orthonormalizing is the process of turning a set of vectors into orthogonal (perpendicular vectors) and normal (vectors of length 1). Let $S = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ be the set of vectors to be orthonormalized, and define the the projection vectors by

$$\operatorname{proj}_{\boldsymbol{u}}(\boldsymbol{v}) = \frac{\langle \boldsymbol{v}, \boldsymbol{u} \rangle}{\langle \boldsymbol{u}, \boldsymbol{u} \rangle} \cdot \boldsymbol{u}$$

where $\langle \boldsymbol{v}, \boldsymbol{u} \rangle$ is the inner product of $\boldsymbol{u}$ and $\boldsymbol{v}$. Then the Gram-Schmidt process is executed in the following way:

$$\begin{aligned}
\boldsymbol{u}_1 &= \boldsymbol{v}_1, & \boldsymbol{e}_1 &= \frac{\boldsymbol{u}_1}{\|\boldsymbol{u}_1\|} \\
\boldsymbol{u}_2 &= \boldsymbol{v}_2 - \operatorname{proj}_{\boldsymbol{u}_1}(\boldsymbol{v}_2), & \boldsymbol{e}_2 &= \frac{\boldsymbol{u}_2}{\|\boldsymbol{u}_2\|} \\
\boldsymbol{u}_3 &= \boldsymbol{v}_3 - \operatorname{proj}_{\boldsymbol{u}_1}(\boldsymbol{v}_3) - \operatorname{proj}_{\boldsymbol{u}_2}(\boldsymbol{v}_3), & \boldsymbol{e}_3 &= \frac{\boldsymbol{u}_3}{\|\boldsymbol{u}_3\|} \\
&\vdots & &\vdots \\
\boldsymbol{u}_n &= \boldsymbol{v}_n - \sum_{i=1}^{n-1} \operatorname{proj}_{\boldsymbol{u}_i}(\boldsymbol{v}_n), & \boldsymbol{e}_n &= \frac{\boldsymbol{u}_n}{\|\boldsymbol{u}_n\|}
\end{aligned}$$

This process ensures that each new calculated vector is orthogonal to all the previously calculated vectors.

2.5 Hash Functions

Hash functions are mathematical algorithms that take an input of any size and produce a fixed-size string of characters, often referred to as a hash or digest. The primary purpose of hash functions is to quickly and efficiently map data to a unique output value. A good hash function should have two important properties: Firstly, for an arbitrary sized input it should provide a fixed output. Secondly, it should be collision-resistant, this means that it should be computationally infeasible to find two distinct inputs that produce the same output.

Since (most) hash functions output a set number of characters, if the input is larger than this, it essentially gives you a way to compress data. During this project the following hash functions are used:

1. SHA3-256 - Produces a fixed output of 256 bits.
2. SHA3-512 - Produces a fixed output of 512 bits.

3. SHAKE-128 - Can produce output of different lengths but typically around 128 bits.
4. SHAKE-256 - Can produce output of different lengths but typically around 256 bits.

Note that for the SHAKE functions, SHAKE-256 is better optimized to produce large outputs compared to SHAKE-128, but SHAKE-128 is faster in doing smaller outputs.

2.6 Cryptography

Cryptography is a way of making information secure by using special techniques to turn it into a secret code. It involves using mathematical algorithms and methods to scramble the information so that only those who have the special "key" can understand it. It's like putting your message inside a locked box that can only be opened with the right key. This helps keep important information, like passwords or private messages, safe from unauthorized access or tampering. This process is accomplished by using encryption systems.

There are two important operations that are used in encryption systems: encryption and decryption. Encryption takes the message that you want to send (sometimes called the plaintext) and scrambles it so that nobody can unscramble it (usually called the ciphertext) if they do not have the key for it. Decryption uses the secret key to unscramble the ciphertext so that it can retrieve the plaintext. There are two main types of cryptography: symmetric key encryption and asymmetric key encryption, and they differ in how the keys are generated and used.

2.6.1 Symmetric Key Encryption

In symmetric key encryption, only one key is used. Both communicating parties must have the same key in order for it to work. Figure 2 shows how the key is used in this type of encryption. Imagine Alice and Bob are trying to communicate with each other, then, if Alice wants to send Bob a message, she will use the secret key to encrypt the message and send it. When Bob receives the message, he will also use the secret key to decrypt the message, recovering the original message.

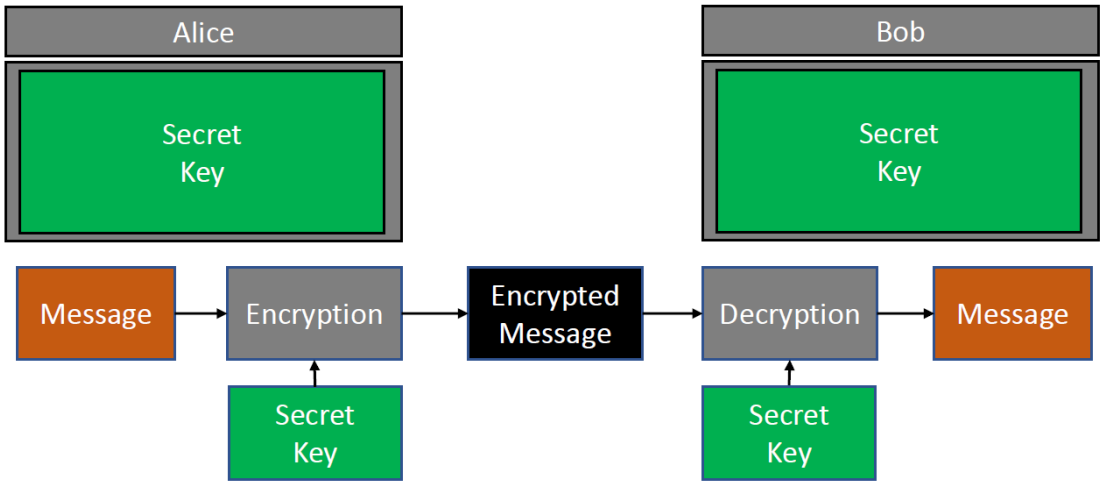

Figure 2: Example of how symmetric key encryption works.

There is one main weakness of this system, namely, how do you ensure both Alice and Bob have the

same secret key? If they send it without encrypting it, malicious attackers could intercept the key and use it to decrypt all the messages that are exchanged between Alice and Bob. Asymmetric key encryption provides an answer to this, and is most commonly known as public key encryption.

2.6.2 Public Key Encryption (PKE)

This scheme was invented by Whitfield Diffie and Martin Hellman in 1976 [11], and in 2015 won the Turing Award for bridging the world of cryptography from secrecy to the public sphere [15]. In public key encryption systems, each user has two keys. One is the public key which is known by everyone that wants to communicate with a user, and the other is the private key which is only known to the user it belongs to. The special feature of these keys is that they inverse each others operations, therefore if you encrypt something with a public key, it can only be decrypted by its corresponding private key. Figure 3 shows how the keys are used in this type of encryption. Suppose again that Alice and Bob are trying to communicate with each other, then, if Alice wants to send Bob a message, she will use Bob's public key to encrypt the message and send it. When Bob receives the message, he will use his secret key to decrypt the message, recovering the original message. The keys are related in such a way that it is computationally infeasible to recover the private key from the public key.

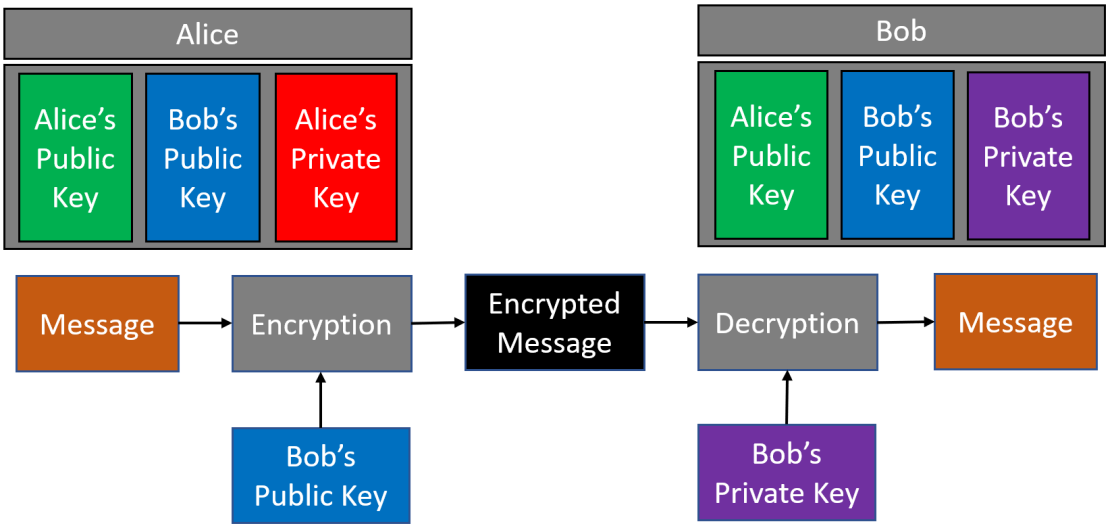

Figure 3: Example of how public key encryption works.

Now a formal definition for PKE is provided as given in [10].

Public Key Encryption (PKE). A *public key encryption* scheme is a set of algorithms (KeyGen, Enc, Dec) with the following specifications:

- KeyGen: a probabilistic algorithm that outputs a public key pk and a secret key sk.
- Enc: a probabilistic algorithm that takes as input a public key pk and a message μ and outputs a ciphertext $\boldsymbol{c}$.
- Dec: a deterministic algorithm that takes as input a secret key sk and a ciphertext $\boldsymbol{c}$ and outputs a message μ.

Additionally, let $0 < \delta < 1$. Then a PKE is $(1-\delta)$-correct if for any (pk, sk) generated from KeyGen and μ,

$$\mathbb{P}[\mathsf{Dec}(sk, \mathsf{Enc}(pk, \mu)) \neq \mu] \leq \delta,$$

where the probability is taken over the randomness of the encryption algorithm. We call the above probability decryption failure probability (DFP). Indeed, this is the probability that, after encrypting a message μ and later decrypting it, the message retrieved from decryption is not the same as the original message.

While public key encryption has its security advantages over symmetric encryption, it has some disadvantages regarding speed and size. PKE has slower operations and larger ciphertexts as compared to symmetric encryption, which means that in some applications, such as live communications, symmetric encryption is more efficient and preferable. Hence, a system that is somewhere between symmetric and asymmetric encryption is needed which has the speed of symmetric encryption and the security of asymmetric encryption. A hybrid of both can be used to accomplish this, which is aptly referred to as hybrid encryption.

2.6.3 Key Encapsulation Mechanism (KEM)

In short, a KEM is a public key encryption system that supports hybrid encryption. It generates a symmetric key that is encrypted using public key encryption to send to another party. The ciphertext of the actual message is generated using symmetric encryption. The receiver decrypts the symmetric key using their asymmetric secret key, and then uses the symmetric key to decrypt the message sent. This can be seen illustrated in the following example.

Suppose once again that Alice and Bob are trying to communicate with each other, then Alice will first generate a symmetric key K_s and encrypt the message she wants to send to Bob with it, generating ciphertext c_s. This process is called encapsulation. Finally, Alice encrypts K_s with Bob's public key, producing ciphertext c_{pk} and sends both ciphertexts to Bob. When Bob receives the ciphertexts, it first decrypts c_{pk} using his secret key to retrieve K_s, and this process is called decapsulation. Then, he uses K_s to decrypt c_s to retrieve the message that Alice sent her. Figure 4 shows how the keys are used in this type of encryption.

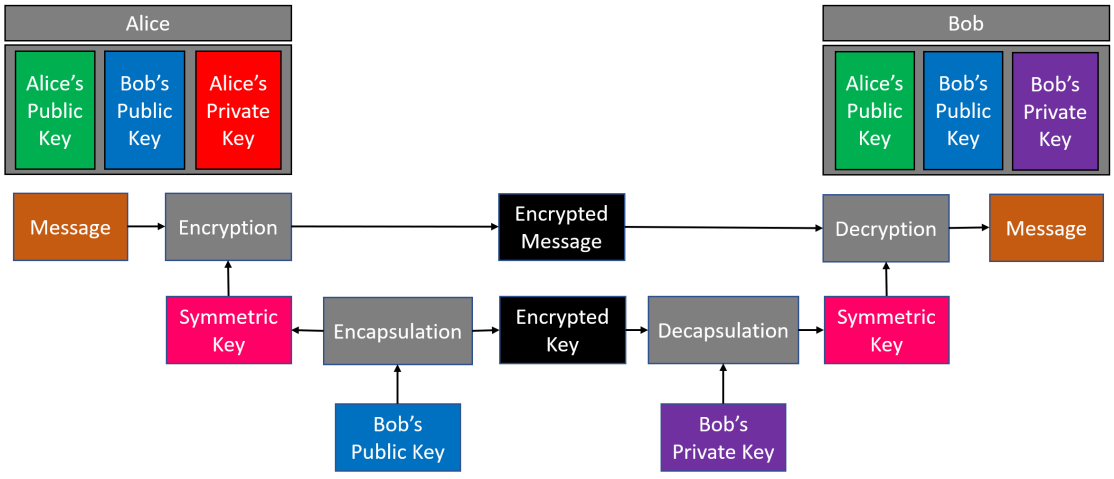

Figure 4: Example of how a KEM works.

Now a formal definition for KEM is provided as given in [10].

Key Encapsulation Mechanism (KEM). A *key encapsulation mechanism* scheme is a set of algorithms (KeyGen, Encap, Decap) with the following specifications:

- KeyGen: a probabilistic algorithm that outputs a public key pk and a secret key sk.
- Encap: a probabilistic algorithm that takes as input a public key pk and outputs a sharing key K and a ciphertext $\boldsymbol{c}$.
- Decap: a deterministic algorithm that takes as input a secret key sk and a ciphertext $\boldsymbol{c}$ and outputs a sharing key K.

The correctness of KEM is defined similarly to that of PKE.

2.7 Security

2.7.1 IND-CPA

IND-CPA (Indistinguishability under Chosen Plaintext Attack) is a desirable property that encryption schemes should have. It is used to evaluate the security of an encryption algorithm and assess the vulnerabilities to chosen plaintext attacks. It can be defined as a game between an adversary (attacker) and a challenger (the system). Let pk be the public key and sk the secret key of the challenger, then:

1. Challenger: Generates pk and sk, and publishes pk to the adversary and keeps sk.
2. Adversary: May perform some encryptions or other operations. Then, submits two different plaintexts, m_0 and m_1, to the challenger.
3. Challenger: Selects a bit $b \in 0, 1$ uniformly at random, and sends the ciphertext $c = \mathsf{E}(pk, m_b)$ back to the adversary.
4. Adversary: May perform some encryptions or other operations. Then, guesses which plaintext was encrypted by guessing the value of b.

A cryptographic scheme is IND-CPA secure if the advantage the adversary has is negligible, i.e. the attacker has roughly a 50% chance to guess the plaintext correctly. Therefore, attackers aim to find ways of creating attacks where the success probability is better than 50%. The PKE version of both SMAUG and Kyber are IND-CPA secure under a set of complexity theoretic security assumptions. The advantage that an adversary has in a PKE setting is formally described in the next definition.

IND-CPA security of PKE. *For a (quantum) adversary $\mathcal{A}$ against a public key encryption scheme PKE = (KeyGen, Enc, Dec), the IND-CPA advantage of $\mathcal{A}$ is defined as follows for all messages m_0 and m_1:*

$$\mathsf{Adv}_{\mathsf{PKE}}^{\mathsf{IND\text{-}CPA}}(\mathcal{A}) = \left| \mathbb{P}\left[b' = b \mid b \leftarrow \{0,1\}, c \leftarrow \mathsf{Enc}(pk, m_b)\right] - \frac{1}{2} \right|$$

It should be noted that the attacker only has access to the encryption method, so they can call the encryption function as many times as they want but do not have access to either the secret key or the decryption method.

2.7.2 IND-CCA

IND-CCA (Indistinguishability under Chosen Ciphertext Attack) uses a definition similar to IND-CPA. It has two variants, IND-CCA1 and IND-CCA2, both SMAUG and Kyber are IND-CCA2 secure so that will be the focus of this part. Once again, it can be explained as a game between an adversary and a challenger. Let pk be the public key and sk the secret key of the challenger, then:

1. Challenger: Generates pk and sk, and publishes pk to the adversary and keeps sk.
2. Adversary: May perform some encryptions, ask decryption queries or other operations. Then, submits two different plaintexts, m_0 and m_1, to the challenger.
3. Challenger: Selects a bit $b \in 0, 1$ uniformly at random, and sends the ciphertext $c = \mathsf{E}(pk, m_b)$ back to the adversary.
4. Adversary: May perform some encryptions, decryptions or other operations. However, they may not send c back for decryption. Then, guesses which plaintext was encrypted by guessing the value of b.

Same as before, a scheme is said to be IND-CCA2 secure if the advantage the adversary has is negligible. The KEM version of both SMAUG and Kyber are IND-CCA2 secure. The advantage that an adversary has in a KEM setting is formally described in the next definition.

IND-CCA2 security of KEM. *For a (quantum) adversary $\mathcal{A}$ against a key encapsulation mechanism scheme KEM = (KeyGen, Encap, Decap), the IND-CCA2 advantage of $\mathcal{A}$ is defined as follows for all messages m_0 and m_1:*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM}}(\mathcal{A}) = \left| \mathbb{P}\left[b = b' \mid b \leftarrow \{0,1\}, c \leftarrow \mathsf{Encap}(pk, m_b), \mathsf{Decap}\right] - \frac{1}{2} \right|$$

2.8 Fujisaki-Okamoto Transform

The Fujisaki-Okamoto (FO) transform can be used to build KEMs that are IND-CCA secure from PKEs that are IND-CPA secure. This is accomplished by performing two transformations: T and U. In this section these two transformations will be explained.

The simplest way to transform a PKE into a KEM using the FO transform requires that the PKE is OW-CPA (One-Way against Plaintext Checking Attacks) secure. OW-CPA provides the adversary with a plaintext checking oracle that returns 1 if and only if the decryption of the ciphertext returns the original message. Therefore, the first step is to transform the IND-CPA secure schemes into OW-PCA schemes (OW-CPA schemes are actually the requirement, but IND-CPA is a stronger notion therefore it also works). This transformation will be transformation T and is defined in the following manner.

Transformation T. To a public-key encryption scheme PKE = (KeyGen, Enc, Dec) and random oracle G, associate $\mathsf{PKE}_1 = \mathsf{T}[\mathsf{PKE}, G]$. The algorithms of $\mathsf{PKE}_1 = (\mathsf{KeyGen}, \mathsf{Enc}_1, \mathsf{Dec}_1)$ are defined below.

- KeyGen: a probabilistic algorithm that outputs a public key pk and a secret key sk.
- Enc_1: a deterministic algorithm that takes as input a public key pk and a message μ and outputs a ciphertext $\boldsymbol{c}$. It is deterministic because it calls Enc with pk, μ and $G(m)$, where $G(m)$ is used as a seed such that the random oracles in Enc always produce the same value for the same seed.

- Dec: a deterministic algorithm that takes as input a secret key sk and a ciphertext $\boldsymbol{c}$ and outputs a message μ if decryption is successful, and $\perp$ otherwise.

After obtaining PKE_1 which is OW-PCA secure, transformation U is applied to transform PKE_1 into a IND-CCA secure KEM. There are four variations to this transformation, however only the two that are used by SMAUG and Kyber will be discussed. The one SMAUG uses is $\mathsf{U}^{\not\perp}_m$ and the one Kyber uses is $\mathsf{U}^{\not\perp}$, and are defined as follows.

Transformation $\mathsf{U}^{\not\perp}_m$. To a public-key encryption scheme $\mathsf{PKE}_1 = (\mathsf{KeyGen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ and a hash function $H : \{0,1\}^* \to \{0,1\}^n$, associate $\mathsf{KEM}^{\not\perp}_m = \mathsf{U}^{\not\perp}_m[\mathsf{PKE}_1, H]$. The algorithms of $\mathsf{KEM}^{\not\perp}_m = (\mathsf{KeyGen}^{\not\perp}, \mathsf{Encap}_m, \mathsf{Decap}^{\not\perp}_m)$ are defined below.

- $\mathsf{KeyGen}^{\not\perp}$: a probabilistic algorithm that outputs a public key pk and a secret key sk, which is defined as the secret key sk' produced by KeyGen_1 together with a random secret s.
- Encap_m: a probabilistic algorithm that takes as input a public key pk and outputs a sharing key K, which is the hash $H(m)$ of a random message m, and a ciphertext $\boldsymbol{c}$.
- $\mathsf{Decap}^{\not\perp}_m$: a deterministic algorithm that takes as input a secret key sk and a ciphertext $\boldsymbol{c}$ and outputs the sharing key $K = H(m)$ if decryption is successful, and $H(s, c)$ otherwise.

Transformation $\mathsf{U}^{\not\perp}$. To a public-key encryption scheme $\mathsf{PKE}_1 = (\mathsf{KeyGen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ and a hash function $H : \{0,1\}^* \to \{0,1\}^n$, associate $\mathsf{KEM}^{\not\perp} = \mathsf{U}^{\not\perp}[\mathsf{PKE}_1, H]$. The algorithms of $\mathsf{KEM}^{\not\perp} = (\mathsf{KeyGen}^{\not\perp}, \mathsf{Encap}, \mathsf{Decap}^{\not\perp})$ are defined below.

- $\mathsf{KeyGen}^{\not\perp}$: a probabilistic algorithm that outputs a public key pk and a secret key sk, which is defined as the secret key sk' produced by KeyGen_1 together with a random secret s.
- Encap: a probabilistic algorithm that takes as input a public key pk and outputs ciphertext $\boldsymbol{c}$, and a sharing key K, which is the hash $H(m, \boldsymbol{c})$ of a random message m and $\boldsymbol{c}$.
- $\mathsf{Decap}^{\not\perp}$: a deterministic algorithm that takes as input a secret key sk and a ciphertext $\boldsymbol{c}$ and outputs the sharing key $K = H(m, \boldsymbol{c})$ if decryption is successful, and $H(s, \boldsymbol{c})$ otherwise.

To finish formally defining the full FO transform on each scheme, the KEM of each PKE scheme is built in the following manner:

- $\mathsf{SMAUG.KEM}^{\not\perp}_m = \mathsf{FO}^{\not\perp}_m[\mathsf{PKE}, \mathsf{G}, \mathsf{H}] = \mathsf{U}^{\not\perp}_m[\mathsf{T}[\mathsf{SMAUG.PKE}, \mathsf{G}], \mathsf{H}] = (\mathsf{KeyGen}^{\not\perp}, \mathsf{Encap}_m, \mathsf{Decap}^{\not\perp}_m)$.
- $\mathsf{Kyber.KEM}^{\not\perp} = \mathsf{FO}^{\not\perp}[\mathsf{PKE}, \mathsf{G}, \mathsf{H}] = \mathsf{U}^{\not\perp}[\mathsf{T}[\mathsf{Kyber.PKE}, \mathsf{G}], \mathsf{H}] = (\mathsf{KeyGen}^{\not\perp}, \mathsf{Encap}, \mathsf{Decap}^{\not\perp})$.

2.9 Learning with Errors/Rounding

Learning with Errors/Rounding (LWE/R) are security assumptions. Public keys and secret keys are setup via noisy linear operations that use errors from carefully chosen distributions. The LWE/R assumptions now state that the public keys are indistinguishable from random values. To understand better how these work and why they are secure, an example of learning without errors/rounding will be given.

2.9.1 Initial Setup

Let $\boldsymbol{A} \in \mathbb{Z}^{k \times l}$ be an integer matrix of dimensions $k \times l$, $\boldsymbol{s} \in \mathbb{Z}^l$ be an integer l-dimensional vector and $\boldsymbol{b} \in \mathbb{Z}^k$ be an integer k-dimensional vector. The relationship between these is as follows:

$$\boldsymbol{A} \cdot \boldsymbol{s} = \boldsymbol{b}. \tag{1}$$

Then $\boldsymbol{A}$ and $\boldsymbol{b}$ can be kept as the public key, and $\boldsymbol{s}$ as the secret key. However, this is not a hard mathematical problem, since if someone has $\boldsymbol{A}$ and $\boldsymbol{b}$, it is easy to decipher $\boldsymbol{s}$ by using Gaussian elimination. This can be easily seen in the following example:

$\boldsymbol{A}$, $\boldsymbol{s}$ and $\boldsymbol{b}$ are defined as follows:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \qquad \boldsymbol{s} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad \boldsymbol{b} = \begin{pmatrix} 19 \\ 21 \end{pmatrix}.$$

Equation 1 can be rewritten such that it results in the following system of equations:

$$\begin{aligned} x + 2y + 3z &= 19 \\ 3x + 2y + z &= 21. \end{aligned}$$

And using Gaussian elimination retrieves the solutions:

$$\begin{aligned} x &= z + 1 \\ y &= 9 - 2z. \end{aligned}$$

Now, even though a unique solution cannot be found, and therefore the original secret key, it is not needed to break encryption. In fact, any set of numbers that fulfill the above solutions will work, therefore, an additional step needs to be introduced to make this more robust.

2.9.2 Learning with Errors (LWE)

To improve on the previous system, LWE introduces some random "noise" that will make it harder to find the secret key. Additionally it takes the values of $\boldsymbol{A}$, $\boldsymbol{s}$ and $\boldsymbol{b}$ modulo q, the reason for this is that it will be used for error correcting as it will discussed later.

Let $\boldsymbol{A} \in \mathbb{Z}_q^{k \times l}$ be an integer matrix of dimensions $k \times l$, $\boldsymbol{s} \in \mathbb{Z}_q^l$ be an integer l-dimensional vector, $\boldsymbol{e} \in \mathbb{Z}^k$ be an integer k-dimensional error vector and $\boldsymbol{b} \in \mathbb{Z}_q^k$ be an integer k-dimensional vector. The relationship between these is as follows:

$$\boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e} = \boldsymbol{b}. \tag{2}$$

Similar to before, $\boldsymbol{A}$ and $\boldsymbol{b}$ will be our public key and $\boldsymbol{s}$ will be the secret key. Gaussian elimination cannot be used to solve this equation, and since the error $\boldsymbol{e}$ is never stored, it generally becomes quite difficult to break. The error vector should have random small values, such that it only perturbs the result slightly which is also the reason it does not need to be done modulo q. Moreover, generally it is desired that there exists a unique secret key for the public key used.

2.9.3 Learning with Rounding (LWR)

There is another way of doing the previous setup, but instead of small random errors, rounding is used which corresponds to a deterministic error. LWR introduces two integers $q \geq p \geq 2$,which are used to scale the multiplication $\boldsymbol{A} \cdot \boldsymbol{s}$.

Let $\boldsymbol{A} \in \mathbb{Z}_q^{k \times l}$ be an integer matrix of dimensions $k \times l$, $\boldsymbol{s} \in \mathbb{Z}_q^l$ be an integer l-dimensional vector and $\boldsymbol{b} \in \mathbb{Z}_q^k$ be an integer k-dimensional vector. The relationship between these is as follows:

$$\left\lfloor \frac{p}{q} \cdot \boldsymbol{A} \cdot \boldsymbol{s} \right\rceil = \boldsymbol{b}. \tag{3}$$

Once again, A and b will be our public key and s will be the secret key. Similarly to the error variant, Gaussian Elimination cannot be used to solve this equation. Given the conditions on p and q, this method makes $A \cdot s$ smaller and then rounds every value to the nearest integer. Similarly to LWE, generally it is desired that there exists a unique secret key for the public key used.

2.10 Module Learning with Errors/Rounding

Based on the same ideas as LWE and LWR, Module Learning with Errors/Rounding (MLWE/R) builds upon them by making the elements of the matrices go from integers to polynomials in a polynomial quotient ring. For example, as it is the case in SMAUG, the elements go from being in $\mathbb{Z}_q$ to $\mathcal{R}_q$. MLWE/R incorporates a modular structure into the equations. This structure has several advantages over LWE/R, such as increased efficiency, which is why it is preferred in cryptographic applications. MLWE/R is actually a predecessor to Ring Learning with Errors/Rounding (R-LWE/R) instead of LWE/R. An informal definition of R-LWE/R is that it is MLWE/R of module rank 1, and it provides better efficiency at the cost of security. This is because essentially $\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{s}$ and $\boldsymbol{e}$ are only one polynomial each, meaning that there are fewer polynomial multiplications. However, this means that the scheme is less secure so longer polynomials will be needed to achieve the same level of security. MLWE/R was designed such that it had better security than RLWE/R, but better performance than LWE/R [1].

To prove the security of such assumptions, reductions from one problem to another are used. For starters, there exists a reduction from a worst-case lattice problem to an average-case LWE problem, like the SVP [21][23]. This means solving a random LWE problem is at least as hard as solving the worst case in the the lattice problem. Another way this can be interpreted is that if LWE is solved, then the lattice problem is solved too through this reduction. This gives confidence on the security assumption of LWE. Moreover, there exist reductions from module lattices to MLWE, where problems in module lattices are proven to be equally hard as the generic counterpart [18]. Further work also proved there to be a reduction from MLWE to MLWR [8]. Therefore, MLWE could be a problem that is harder to solve than MLWR. Additionally, so far there are no attacks that specifically target the module structure of MLWE/R.

The modular structure of MLWE enables the efficient utilization of techniques such as the Number Theoretic Transform (NTT), leading to faster encryption and decryption operations.

Now an example of how encryption works under the MLWE assumption is presented. This example is taken from [16], and is based off Kyber. Imagine that Alice is trying to send Bob an encrypted message using a public key encryption system. Let $(\boldsymbol{A}, \boldsymbol{b})$ be the public key and $\boldsymbol{s}$ the secret key of Bob. Then Alice only has access to the values of $\boldsymbol{A}$ and $\boldsymbol{b}$. Let $n = 4$ and $q = 17$ such that $\mathcal{R}_q = \mathbb{Z}_{17}/\{x^4 + 1\}$. Then let $\boldsymbol{A}$, $\boldsymbol{s}$, $\boldsymbol{b}$ and $\boldsymbol{e}$ take the following values:

$$\boldsymbol{A} = \begin{pmatrix} 6x^3 + 16x^2 + 16x + 11 & 9x^3 + 4x^2 + 6x + 3 \\ 5x^3 + 3x^2 + 10x + 1 & 6x^3 + x^2 + 9x + 15 \end{pmatrix} \qquad \boldsymbol{e} = \begin{pmatrix} x^2 \\ x^2 - x \end{pmatrix}$$

$$\boldsymbol{b} = \begin{pmatrix} 16x^3 + 15x^2 + 7 \\ 10x^3 + 12x^2 + 11x + 6 \end{pmatrix} \qquad \boldsymbol{s} = \begin{pmatrix} -x^3 - x^2 + x \\ -x^3 - x \end{pmatrix}.$$

For encryption two additional vectors and a polynomial are needed, and these values are freshly generated every time the encryption process is started. Vector $\boldsymbol{r}$ will be a random polynomial vector, vector $\boldsymbol{e}_1$ will be a small error polynomial vector and e_2 will be a small error polynomial.

$$\boldsymbol{r} = \begin{pmatrix} -x^3 + x^2 \\ x^3 + x^2 - 1 \end{pmatrix} \qquad \boldsymbol{e}_1 = \begin{pmatrix} x^2 + x \\ x^2 \end{pmatrix} \qquad e_2 = -x^3 - x^2$$

Now to encrypt a message in bits, the bits are turned into the coefficients of the polynomials. For example, say that $(11)_{10} = (1011)_2$ is what should be sent, then the polynomial representation will be:

$$m_b = 1x^3 + 0x^2 + 1x^1 + 1x^0 = x^3 + x + 1.$$

The final step before starting to encrypt is to scale up the coefficients by $\lfloor \frac{q}{2} \rceil = 9$, the reason for this is so that it can correct the error caused by the noise in $\boldsymbol{e}$. Therefore the message to be sent will be:

$$m = 9x^3 + 9x + 9.$$

The process of encryption calculates two values, $\boldsymbol{u}$ and v, and they are calculated in the following way:

$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{A}^\intercal \cdot \boldsymbol{r} + \boldsymbol{e}_1 \\ v &= \boldsymbol{b}^\intercal \cdot \boldsymbol{r} + e_2 + m \end{aligned}$$

Therefore substituting the values:

$$\begin{aligned} \boldsymbol{u} &= (11x^3 + 11x^2 + 10x + 3, 4x^3 + 4x^2 + 13x + 11) \\ v &= 7x^3 + 6x^2 + 8x + 15. \end{aligned}$$

This is what Alice will send to Bob, the ciphertext. Then to decrypt the message, Bob simply has to solve this equation:

$$m_n = v - \boldsymbol{s}^\intercal \cdot \boldsymbol{u}.$$

Note that this is a noisy message because of the errors that were introduced. Which means that the coefficients will not be 0 or 9 exactly. The reason this works is because if the terms are expanded, the following derivation is obtained:

$$\begin{aligned} m_n &= v - \boldsymbol{s}^\intercal \cdot \boldsymbol{u} \\ &= \boldsymbol{b}^\intercal \cdot \boldsymbol{r} + e_2 + m - \boldsymbol{s}^\intercal \cdot (\boldsymbol{A}^\intercal \cdot \boldsymbol{r} + \boldsymbol{e}_1) \\ &= (\boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e})^\intercal \cdot \boldsymbol{r} + e_2 + m - \boldsymbol{s}^\intercal \cdot \boldsymbol{A}^\intercal \cdot \boldsymbol{r} - \boldsymbol{s}^\intercal \cdot \boldsymbol{e}_1 \\ &= \boldsymbol{s}^\intercal \cdot \boldsymbol{A}^\intercal \cdot \boldsymbol{r} + \boldsymbol{e}^\intercal \cdot \boldsymbol{r} + e_2 + m - \boldsymbol{s}^\intercal \cdot \boldsymbol{A}^\intercal \cdot \boldsymbol{r} - \boldsymbol{s}^\intercal \cdot \boldsymbol{e}_1 \\ &= \boldsymbol{e}^\intercal \cdot \boldsymbol{r} + e_2 + m - \boldsymbol{s}^\intercal \cdot \boldsymbol{e}_1. \end{aligned}$$

Therefore m is a noisy variant of m, however, it perturbs the original m so little that the real message can be figured out. The result of solving this equation is:

$$m_n = 8x^3 + 14x^2 + 8x + 6.$$

The coefficients that are close to 9 are the coefficients that represented the bit 1, and the ones close to 0 represented bit 0. Therefore the original message is indeed $(1011)_2 = (11)_{10}$. To encrypt with MLWR a similar process is used, but instead of adding small errors, the equations are scaled and rounded.

2.11 Number-Theoretic Transform (NTT)

The Number-Theoretic Transform is an efficient way to perform multiplications in $\mathcal{R}_q$. However, it needs q to be of a certain form, namely $q = p \cdot 2^k + 1$ where q and p are primes and $k \in \mathbb{Z}$. Since

SMAUG uses q as powers of 2 it cannot benefit from this, nonetheless, Kyber does have q in such a way. In Kyber, $q = 3329 = 13 \cdot 2^8 + 1$ so NTT can be used. The following is an explanation of how NTT works in Kyber's case.

Thanks to the special properties of q, the defining polynomial of $\mathcal{R}_q$, $X^{256} + 1$, can be factored into 128 polynomials of degree 2 modulo q:

$$X^{256} + 1 = \prod_{i=0}^{127} (X^2 - 17^{2i+1})$$

Then the NTT of $f \in \mathcal{R}_q$ is given by:

$$(f \mod X^2 - 17^{2(0)+1}, \ldots, f \mod X^2 - 17^{2(127)+1}).$$

So each term in the NTT representation is a degree 1 polynomial. Alternatively, the NTT of f can be seen as:

$$\mathsf{NTT}(f) = \hat{f} = (\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, \ldots, \hat{f}_{254} + \hat{f}_{255} X)$$

where,

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j} \cdot 17^{(2i+1)j}$$
$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1} \cdot 17^{(2i+1)j}.$$

Therefore multiplication of $f \cdot g$ can be accomplished by $\mathsf{NTT}^{-1}(\mathsf{NTT}(f) \cdot \mathsf{NTT}(g))$ where $\mathsf{NTT}(f) \cdot \mathsf{NTT}(g) = \hat{f} \cdot \hat{g} = \hat{h}$ and NTT^{-1} is the inverse operation. So each term in $\hat{h}$ takes the form:

$$\hat{h}_{2i} + \hat{h}_{2i+1} X = (\hat{f}_{2i} + \hat{f}_{2i+1} X) \cdot (\hat{g}_{2i} + \hat{g}_{2i+1} X) \mod X^2 - 17^{2i+1}.$$

This means that $\hat{h}$ is calculated by multiplying many little linear polynomials which is a cheaper operation that multiplying two big polynomials of higher degrees. Applying $\mathsf{NTT}^{-1}(\hat{h})$ recovers the actual result of $f \cdot g$. Using fast algorithms, the time complexity of NTT-based polynomial multiplication is $\mathcal{O}(n \log n)$ [19].

3 SMAUG

In this section, SMAUG's algorithms for both its PKE and KEM variants are explored. SMAUG PKE is IND-CPA secure as proven in Section 6.1 , and SMAUG KEM is IND-CCA2 secure as proven in [10]. The parameters that are mentioned throughout this section are instantiated in Table 1. The rings from which the polynomials are taken from are defined in Section 2.2.

3.1 Design Overview

3.1.1 MLWE and MLWR

The security of the key generation relies on the MLWE assumption and the security of the encryption relies on the MLWR assumption. Since LWE-based problems have been studied for longer, many schemes choose to use these problems for the security of both their key generation and encryption. The modular variant of LWE in particular allows for better efficiency and fine-tuning security better than its ring and standard counterparts. However, MLWR is considered as hard as MLWE unless the same secret is used to create too many samples [9]. This is because the rounding operations tend to leak more data about the secret key. Therefore, SMAUG uses it in encryption only, and this makes the ciphertexts smaller and the encryption/decryption more efficient. The reasons LWE/R and RLWE/R are not used are described in Section 2.10.

3.1.2 Choice of Moduli

All moduli in SMAUG are in powers of 2. This choice is made since binary information is stored in base 2, therefore operations such as bit shifting and scaling become easier. Bit shifting is when you shift all the bits in a number to the left or right. For example, if you have the number $6_{10} = (0110)_2$ you can shift the bits to the left by multiplying by 2. Indeed, $12_{10} = (1100)_2$. Similarly, you can shift the bits to the right by dividing by 2, since $3_{10} = (0011)_2$. However, since the moduli are powers of two, it means that they cannot benefit from using NTT. To combat this, SMAUG introduces a polynomial multiplication method that leverages the fact that in all multiplications, there is a sparse polynomial.

3.1.3 Sparse Secret and Ephemeral Key

SMAUG does not use the traditional MLWE problem, and instead makes use of MLWE with a sparse secret. This means that the secret key ends up taking less space which is an advantage for storage limited devices. The coefficients only take three different values $(-1,\ 0,\ 1)$ and this structure is what allows SMAUG to take advantage of faster multiplications. For sparse secrets, instead of storing the coefficients directly, the degrees at which nonzero coefficients exist are stored. First the degrees of the terms with coefficient 1 are stored, then the ones for coefficient -1 while making note of where the negative coefficients start. For example, if you had the polynomial $x^4 + x^3 - x$ then, instead of storing the polynomial as $[0, -1, 0, 1, 1]$, it is stored $[3, 4, 1]$ with a counter neg_start $= 2$ (note that it is 2 and not 3 since list positions start at 0 instead of 1).

Algorithm 1: poly_mult_add

Input: $a \in \mathcal{R}_q, b \in \mathcal{S}_\eta$
For i from 0 to neg_start-1 **do**
    degree $= b[i]$
    For j from 0 to $n-1$ **do**
        $x[\text{degree} + j]$ += $a[j]$
    end for
end for
For i from neg_start to len$(b)-1$ **do**
    degree $= b[i]$
    For j from 0 to $n-1$ **do**
        $x[\text{degree} + j]$ -= $a[j]$
    end for
end for
For i from 0 to $n-1$ **do**
    $y[j] = x[j] - x[n+j]$
end for
Return: y

Algorithm 1 shows how polynomial multiplication happens in SMAUG. As stated in Section 2.2, for an integer η, the set of polynomials of degree less than n with coefficients in $[-\eta, \eta] \in \mathbb{Z}$ is defined as $\mathcal{S}$. In this case, since it describes the coefficients of a sparse polynomial, $\eta = 1$ and therefore the coefficients are in $\{-1, 0, 1\}$. Additionally, n takes the value $n = 256$ in all variants of SMAUG, with $q = 1024$ for SMAUG-128 and $q = 2028$ for the others. This algorithm leverages the fact that in every matrix or vector multiplication (and corresponding polynomial multiplications), at least one of the terms will be sparse. As it will be seen in the following sections, this is always the case. It is easily seen that the time complexity of this algorithm is $\mathcal{O}(\text{len}(b) \cdot (1+n) + n)$. This is because the first two for loops cost the same to run and they happen in total len(b) times. Then the inner part of the loop runs one operation to retrieve the degree and n operations adding the list elements. Similarly for the last for loop, one instruction is carried out n times, hence $\mathcal{O}(\text{len}(b) \cdot (1+n) + n)$.

3.1.4 Hash Functions

The following hash functions are used:

- H - Is instantiated with SHA3-256 [13].
- G - Is instantiated with SHAKE-256 [13].
- XOF - Is instantiated with SHAKE-128 (XOF stands for eXtendable-Output Function) [13].

3.1.5 Sampling Algorithms

Each variant of SMAUG uses three different sampling algorithms: expandA, HWT and dGaussian. expandA is used to generate $\boldsymbol{A}$ from a seed and the process can be seen in Algorithm 2. It uses the seed as input for XOF, which creates further pseudorandomness, and the output is stored in buf. Then, for each position in $\boldsymbol{A}$ it populates the polynomials by transforming the bytes into coefficients for the polynomial. Note that i and j are passed as parameters in this function and this is done to offset the bytes used in buf for each position. For example, in SMAUG-128 a polynomial requires 320 bytes to populate a polynomial, because each coefficient needs 10 bits (since it is mod 1024) for representation, there are 256 coefficients and each byte has 8 bits ($(256 \cdot 10)/8 = 320$). Then

bytes_to_Rq could take the first $320 + i \cdot \text{offset}_i + j \cdot \text{offset}_j$ bytes of buf for $i, j \in [0, 1]$. This way it ensures that different bytes in buf are used for each position. The function expandA is adopted from Saber which can be found at Algorithm 15 in [6]. The time complexity in terms of instructions in Algorithm 2 is $\mathcal{O}(k^2)$. This is due to the for loops, however, this might be larger depending on how many operations happen in bytes_to_Rq function. Here k refers to the dimension of $\boldsymbol{A}$ which is 2, 3 and 5 for SMAUG-128, SMAUG-192, SMAUG-256 respectively.

Algorithm 2: expandA: uniform matrix sampler

Input: seed $\in \{0,1\}^{256}$
buf $\leftarrow$ XOF(seed)
For i from 0 to $k-1$ **do**
    For j from 0 to $k-1$ **do**
        $\boldsymbol{A}[i][j] =$ bytes_to_Rq(buf, i, j)
    end for
end for
Return: $\boldsymbol{A}$

HWT_h (Hamming Weight Sampler) is used to generate the secret key $\boldsymbol{s}$ and the ephemeral secret $\boldsymbol{r}$ used in encryption, and it is illustrated in Algorithm 3. Similar to expandA, it first populates buf with pseudorandom bytes using the seed passed as input. This function also has a parameter h which determines how many nonzero coefficients the resulting polynomials will have. It should also be mentioned that res is a list that has 0 as an initial value for each position, and its length is the total sum of the number of coefficients between all polynomials in $\boldsymbol{s}$. In other words, in SMAUG-128 $\boldsymbol{s}$ has two polynomials (since $k = 2$) each with 256 coefficients, therefore res is a list of length 512. So in this example $n = 512$, even though as it can be seen in the parameter sets in Table 1, $n = 256$. This is because in this algorithm n is actually $n \cdot k$. Here h takes the value of h_s or h_r depending on its use, with the instantiated values defined in Table 1.

The high level explanation of what happens in this algorithm can be explained as follows. With all the values of res been set to 0, first pick a random degree that is smaller than i. Variable i essentially becomes the last h biggest degrees. Putting SMAUG-128 as an example again, then $i \in [372, 511]$. Then, it copies the coefficient at position "degree" into position i. Next, it randomly assigns 1 or -1 to position "degree". Finally, after all the nonzero coefficients have been generated, convToIdx separates res into the amount of polynomials needed and then stores the degrees at which the nonzero coefficients happen as explained before.

Algorithm 3: HWT_h: hamming weight sampler

Input: seed $\in \{0,1\}^{256}$
buf $\leftarrow$ XOF(seed)
idx $\leftarrow 0$
For i from $n-h$ to $n-1$ **do**
    Repeat
        degree $=$ buf[idx] $\wedge$ mask
        idx $+= 1$
    until degree $< i$
    res[i] $=$ res[degree]
    res[degree] $= ((\text{buf[idx-1]} \gg 14) \wedge 0\text{x}02) - 1$
end for
Return: convToIdx(res)

In a bit more detail, the mask works as a modulo operator which takes the value $q-1$, therefore ensuring that all values of degree are below q. However, q equals 1024 and 2048 in different variants of SMAUG, therefore the value of degree could be larger than the totals sum of number of degrees in a polynomial since the maximum is 511. This is the reason the loop repeats until it finds a smaller degree. Line 10 is another complicated line, but essentially what it does is bit shift the value of buf at idx-1 14 places to the right (similar to dividing the number by 2^{14}). This value is then used in a bit AND operation with the number 2, the result of this is that it produces either a 0 or a 2. If a 0 is produced, the -1 makes it so that the coefficient at that degree is -1, and if a 2 is produced, it makes the coefficient at that degree 1.

The function HWT is adapted from SampleInBall, which can be found in Figure 2 of [12]. The time complexity of this algorithm is at least $\mathcal{O}(h)$, and it could be larger depending on how many times the repeat loop runs for.

The algorithm dGaussian is a discrete Gaussian sampler which is used to generate the coefficients of the error polynomial during the key generation. So this algorithm is called for each coefficient of each polynomial in $\boldsymbol{e}$. There are two variants of this algorithm, since SMAUG-192 uses a different one as opposed to the other variants. The two algorithms are distinguished by their standard deviation and the most remarkable result is that the one with $\sigma = 1.0625$ gives coefficients in $[-3,3]$ and the one with $\sigma = 1.0625$ gives coefficients in $[-7,7]$. Algorithms 4 and 5 illustrate how the sampling happens.

It should be noted that, except for the lines that have the $\in \{0,1\}^c$, $x_0x_1x_2 + x_2x_3\overline{x_4}$ should be read as (x_0 AND x_1 AND x_2) OR (x_2 AND x_3 AND (NOT x_4)). This algorithm has a constant time complexity.

Algorithm 4: dGaussian: discrete Gaussian sampler with $\sigma = 1.0625$

1 **Input:** $x = x_0x_1x_2x_3x_4x_5x_6x_7x_8x_9 \in \{0,1\}^{10}$
2 $s_0 = x_0x_1x_2x_3x_4x_5x_7\overline{x_8}$
3 $s_0 \mathrel{+}= (x_0x_3x_4x_5x_6x_8) + (x_1x_3x_4x_5x_6x_8) + (x_2x_3x_4x_5x_6x_8)$
4 $s_0 \mathrel{+}= (\overline{x_2}x_3\overline{x_6}x_8) + (\overline{x_1}x_3\overline{x_6}x_8)$
5 $s_0 \mathrel{+}= (x_6x_7\overline{x_8}) + (\overline{x_5}x_6x_8) + (\overline{x_4}x_6x_8) + (\overline{x_7}x_8)$
6 $s_1 = (x_1x_2x_4x_5x_7x_8) + (x_3x_4x_5x_7x_8) + (x_6x_7x_8)$
7 $s = s_1s_0 \in \{0,1\}^2$
8 $s = (-1)^{x_9} \cdot s$
9 **Return:** s

Algorithm 5: dGaussian: discrete gaussian sampler with $\sigma = 1.453713$

1 **Input:** $x = x_0x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10} \in \{0,1\}^{11}$
2 $s_0 = (x_0x_1x_2x_3x_5x_7x_8) + (x_1x_2x_3x_5\overline{x_6}x_7x_9) + (\overline{x_1x_2x_3}x_6x_7x_8)$
3 $s_0 \mathrel{+}= (\overline{x_1}x_2x_3x_5x_8x_9) + (\overline{x_0}x_2x_3x_5x_8x_9)$
4 $s_0 \mathrel{+}= (x_4x_5\overline{x_6}x_7x_9) + (x_3x_4x_8\overline{x_9}) + (\overline{x_5}x_6x_7x_8) + (\overline{x_4}x_6x_7x_8) + (\overline{x_4x_5x_8}x_9)$
5 $s_0 \mathrel{+}= (x_5x_8\overline{x_9}) + (x_6x_8\overline{x_9}) + (x_6x_8\overline{x_9}) + (x_7x_8\overline{x_9}) + (\overline{x_7x_8}x_9) + (\overline{x_6x_8}x_9)$
6 $s_1 = (x_0x_1x_4\overline{x_5}x_6x_7x_9) + (x_2x_4\overline{x_5}x_6x_7x_9) + (x_3x_4\overline{x_5}x_6x_7x_9) + (x_5x_6x_7\overline{x_8}x_9)$
7 $s_1 \mathrel{+}= (\overline{x_1x_2x_3}x_8x_9) + (\overline{x_7}x_8x_9) + (\overline{x_6}x_8x_9) + (\overline{x_5}x_8x_9) + (\overline{x_4}x_8x_9)$
8 $s_2 = (x_1x_4x_5x_6x_7x_8x_9) + (x_2x_4x_5x_6x_7x_8x_9) + (x_3x_4x_5x_6x_7x_8x_9)$
9 $s = s_2s_1s_0 \in \{0,1\}^3$
10 $s = (-1)^{x_{10}} \cdot s$
11 **Return:** s

3.2 Specification of SMAUG.PKE

3.2.1 Key Generation

Algorithm 6: SMAUG.PKE.KeyGen: key generation

1 $\mathsf{seed} \leftarrow \{0,1\}^{256}$
2 $(\mathsf{seed}_{\boldsymbol{A}}, \mathsf{seed}_{\mathsf{sk}}) \leftarrow \mathsf{XOF}(\mathsf{seed})$
3 $\boldsymbol{A} \leftarrow \mathsf{expandA}(\mathsf{seed}_{\boldsymbol{A}}) \in \mathcal{R}_q^{k \times k}$
4 $\boldsymbol{s} \leftarrow \mathsf{HWT}_{h_s}(\mathsf{seed}_{\mathsf{sk}}) \in \mathcal{S}_\eta^k$
5 $\boldsymbol{e} \leftarrow \mathsf{dGaussian}_\sigma(\mathsf{seed}_{\mathsf{sk}}) \in \mathcal{R}^k$
6 $\boldsymbol{b} \leftarrow -\boldsymbol{A}^\intercal \cdot \boldsymbol{s} + \boldsymbol{e} \in \mathcal{R}_q^k$
7 **Return:** $pk = (\mathsf{seed}_{\boldsymbol{A}}, \boldsymbol{b})$, $sk = \boldsymbol{s}$

Algorithm 6 generates a key-pair of a public and secret key. The public key is composed of $\mathsf{seed}_{\boldsymbol{A}}$ and $\boldsymbol{b}$ while the secret key is only composed of $\boldsymbol{s}$. The public key contains $\mathsf{seed}_{\boldsymbol{A}}$ instead of $\boldsymbol{A}$ to make the public key smaller, however this means that every time $\boldsymbol{A}$ needs to be used it must be recomputed. The secret key is $\boldsymbol{s}$, which is initialized using Algorithm 3 (HWT_{h_s}). This algorithm will make sure that there are only h_s nonzero coefficients in the polynomials of $\boldsymbol{s}$. In the case of SMAUG-128 $h_s = 140$ and $k = 2$, which means that $\boldsymbol{s}$ will have two polynomials with a combined total of 140 nonzero coefficients. Additionally, those nonzero coefficients will take either 1 or -1 as their value. Since the coefficients are small and there are not many nonzero coefficients, this is what makes the secret sparse.

Moreover, even though the usual format of LWE is that $\boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e}$, here they choose to use the negative transpose of $\boldsymbol{A}$. While it is never explained explicitly why they chose to do this, it does allow for the correct encryption and decryption operations to happen. Finally, although the polynomials in $\boldsymbol{e}$ are not made modulo some number, the coefficients are kept small enough to not perturb the result too much (as seen before either in $[-3, 3]$ or $[-7, 7]$), reducing the probability of decryption failure.

The time complexity of this algorithm is $\mathcal{O}(n \cdot h \cdot k)$, because of the polynomial multiplication. The functions expandA and HWT_{h_s} have time complexities $\mathcal{O}(k^2)$ and $\mathcal{O}(h)$ respectively. Moreover, there are k^2 polynomial multiplications with time complexity $\mathcal{O}(\mathsf{len}(b) \cdot (1+n) + n)$. This time complexity can be reworked so that it no longer has the term $\mathsf{len}(b)$, and this is because $\mathsf{len}(b)$ is the number of nonzero coefficients in one polynomial of $\boldsymbol{s}$. However, the sum of all of these is h. Therefore, there are k polynomial multiplications with time complexity $\mathcal{O}(h \cdot (1+n) + n)$. Introducing k means that we get the terms $kh + khn + kn$, from which khn is clearly the largest, even against k^2 and h. Therefore, the time complexity of this algorithm is $\mathcal{O}(n \cdot h \cdot k)$ as stated before.

3.2.2 Encryption

Algorithm 7: SMAUG.PKE.Enc: encryption

1 **Input:** $pk = (\mathsf{seed}_{\boldsymbol{A}}, \boldsymbol{b})$, $\mu \in \mathcal{R}_t$, (optional) seed_r
2 $\boldsymbol{A} \leftarrow \mathsf{expandA}(\mathsf{seed}_{\boldsymbol{A}}) \in \mathcal{R}_q^{k \times k}$
3 **If** seed_r is not given **then** $\mathsf{seed}_r \leftarrow \{0,1\}^{256}$
4 $\boldsymbol{r} \leftarrow \mathsf{HWT}_{h_r}(\mathsf{seed}_{\boldsymbol{r}}) \in \mathcal{S}_\eta^k$
5 $\boldsymbol{c}_1 \leftarrow \lfloor p/q \cdot \boldsymbol{A} \cdot \boldsymbol{r} \rceil \in \mathcal{R}_p^k$
6 $c_2 \leftarrow \lfloor p'/q \cdot \langle \boldsymbol{b}, \boldsymbol{r} \rangle + p'/t \cdot \mu \rceil \in \mathcal{R}_{p'}$
7 **Return:** $\boldsymbol{c} = (\boldsymbol{c}_1, c_2)$

Algorithm 7 encrypts a message μ and outputs a vector $\boldsymbol{c}$ which contains the ciphertext. The vector $\boldsymbol{r}$ is used to introduce randomness when encrypting, and is sparse like $\boldsymbol{s}$ to ensure that it perturbs the result as little as possible. Additionally, it is also scaled using p, p', q and t, and then it is rounded. While no explicit reasons were given for the use of p and p' instead of q, it probably has to do with a balance on security and storage efficiency. A scheme ideally should produce small ciphertexts, but if they are too small, they might be less secure since they have to represent a larger set of plaintexts. Additionally, reducing it too much can increase the decryption failure probability by increasing the amount of plaintexts mapped to the same ciphertext.

c_2 is used to encrypt the message, however $\boldsymbol{c}_1$ is needed so that the secret key can be used to cancel out $\boldsymbol{b}$. Note that the message μ is interpreted as a polynomial in $\mathcal{R}_t$, and since $t = 2$ for every SMAUG specification, it allows a 256-bit number to be represented in polynomial form. To do this, set the 0s and 1s in the coefficients of the appropriate degrees.

The time complexity of this algorithm is $\mathcal{O}(n \cdot h \cdot k)$, because of the polynomial multiplication for the same reason as before. The polynomial multiplication in c_2 does not contribute to this time complexity since it only multiplies two vectors together, while in $\boldsymbol{c}_1$ the matrix $\boldsymbol{A}$ ensures there are multiple multiplications of two vectors together.

3.2.3 Decryption

Algorithm 8: SMAUG.PKE.Dec: decryption

Input: $sk = \boldsymbol{s}$, $\boldsymbol{c} = (\boldsymbol{c}_1, c_2)$
$\mu' \leftarrow \lfloor t/p \cdot \langle \boldsymbol{c}_1, \boldsymbol{s} \rangle + t/p' \cdot c_2 \rceil \in \mathcal{R}_t$
Return: μ'

Algorithm 8 decrypts the ciphertext received with the help of the secret key. The time complexity of this algorithm is $\mathcal{O}(n \cdot h)$, because of the polynomial multiplication. It will now be shown that this does indeed recover the message.

$$\begin{aligned}
\mu' &= \lfloor t/p \cdot \langle \boldsymbol{c}_1, \boldsymbol{s} \rangle + t/p' \cdot c_2 \rceil \\
&= \lfloor t/p \cdot \langle \lfloor p/q \cdot \boldsymbol{A} \cdot \boldsymbol{r} \rceil, \boldsymbol{s} \rangle + t/p' \cdot \lfloor p'/q \cdot \langle \boldsymbol{b}, \boldsymbol{r} \rangle + p'/t \cdot \mu \rceil \rceil \\
&= t/p \cdot \langle p/q \cdot \boldsymbol{A} \cdot \boldsymbol{r} - \boldsymbol{\epsilon_1}, \boldsymbol{s} \rangle + t/p' \cdot (p'/q \cdot \langle \boldsymbol{b}, \boldsymbol{r} \rangle + p'/t \cdot \mu - \epsilon_2) - \epsilon_3
\end{aligned}$$

Here the rounding operators are removed since for any $k, m \in \mathbb{Z}$ it holds that

$$\frac{k}{m} = \left\lfloor \frac{k}{m} \right\rceil + \epsilon,$$

where $\boldsymbol{\epsilon_1} \in [-0.5, 0.5)^k$ and $\epsilon_2, \epsilon_3 \in [-0.5, 0.5)$.

$$\begin{aligned}
\mu' &= t/p \cdot \langle p/q \cdot \boldsymbol{A} \cdot \boldsymbol{r}, \boldsymbol{s} \rangle - t/p \cdot \langle \boldsymbol{\epsilon_1}, \boldsymbol{s} \rangle + t/q \cdot \langle \boldsymbol{b}, \boldsymbol{r} \rangle + \mu - t/p' \cdot \epsilon_2 - \epsilon_3 \\
&= t/q \cdot \langle \boldsymbol{A} \cdot \boldsymbol{r}, \boldsymbol{s} \rangle + t/q \cdot \langle -\boldsymbol{A}^\intercal \cdot \boldsymbol{s} + \boldsymbol{e}, \boldsymbol{r} \rangle + \mu - t/p \cdot \langle \boldsymbol{\epsilon_1}, \boldsymbol{s} \rangle - t/p' \cdot \epsilon_2 - \epsilon_3 \\
&= t/q \cdot \langle \boldsymbol{A} \cdot \boldsymbol{r}, \boldsymbol{s} \rangle + t/q \cdot \langle -\boldsymbol{A}^\intercal \cdot \boldsymbol{s}, \boldsymbol{r} \rangle + \mu + \underline{t/q \cdot \langle \boldsymbol{e}, \boldsymbol{r} \rangle - t/p \cdot \langle \boldsymbol{\epsilon_1}, \boldsymbol{s} \rangle - t/p' \cdot \epsilon_2 - \epsilon_3}
\end{aligned}$$

For now, the error terms will be ignored, therefore proceeding to the following equation:

$$\mu' = t/q \cdot (\langle \boldsymbol{A} \cdot \boldsymbol{r}, \boldsymbol{s} \rangle + \langle -\boldsymbol{A}^\intercal \cdot \boldsymbol{s}, \boldsymbol{r} \rangle) + \mu.$$

Now all that is left to prove is that:

$$\langle \boldsymbol{A} \cdot \boldsymbol{r}, \boldsymbol{s} \rangle = \langle \boldsymbol{A}^\intercal \cdot \boldsymbol{s}, \boldsymbol{r} \rangle. \tag{4}$$

First, $\boldsymbol{Ar}$ is calculated:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,k} \\ A_{2,1} & A_{2,2} & \dots & A_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k,1} & A_{k,2} & \dots & A_{k,k} \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_k \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} A_{1,i} \cdot r_i \\ \sum_{i=1}^{k} A_{2,i} \cdot r_i \\ \vdots \\ \sum_{i=1}^{k} A_{k,i} \cdot r_i \end{bmatrix}.$$

Therefore $\langle \boldsymbol{A} \cdot \boldsymbol{r}, \boldsymbol{s} \rangle = \sum_{j=1}^{k} s_j \sum_{i=1}^{k} A_{j,i} \cdot r_i = \sum_{j=1}^{k} \sum_{i=1}^{k} s_j \cdot A_{j,i} \cdot r_i$.

Next, $\boldsymbol{A}^\intercal \boldsymbol{s}$ is computed:

$$\begin{bmatrix} A_{1,1} & A_{2,1} & \dots & A_{k,1} \\ A_{1,2} & A_{2,2} & \dots & A_{k,2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{1,k} & A_{2,k} & \dots & A_{k,n} \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_k \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} A_{i,1} s_i \\ \sum_{i=1}^{k} A_{i,2} s_i \\ \vdots \\ \sum_{i=1}^{k} A_{i,k} s_i \end{bmatrix}.$$

Therefore $\langle \boldsymbol{A}^\intercal \cdot \boldsymbol{s}, \boldsymbol{r} \rangle = \sum_{j=1}^{k} r_j \sum_{i=1}^{k} A_{i,j} \cdot s_i = \sum_{j=1}^{k} \sum_{i=1}^{k} s_i \cdot A_{i,j} \cdot r_j$.

Indeed, Equation 4 has been proven. So μ' can be computed as:

$$\begin{aligned} \mu' &= t/q \cdot (\langle \boldsymbol{A} \cdot \boldsymbol{r}, \boldsymbol{s} \rangle - \langle \boldsymbol{A}^\intercal \cdot \boldsymbol{s}, \boldsymbol{r} \rangle) + \mu \\ &= t/q \cdot (0) + \mu \\ &= \mu. \end{aligned}$$

So decryption works as intended if the errors are ignored. The effect of the errors is later discussed in Section 6.2.1.

3.3 Specification of SMAUG.KEM

3.3.1 Key Generation

Algorithm 9: SMAUG.KEM.KeyGen: key generation

1 $(pk, sk') \leftarrow$ SMAUG.PKE.KeyGen()
2 $d \leftarrow \{0,1\}^{256}$
3 **Return:** pk, $sk = (sk', d)$

The only difference in this algorithm compared to the PKE version is that a random 256-bit seed is added to the secret key. It has the same time complexity as the PKE version, which is $\mathcal{O}(n \cdot h \cdot k)$.

3.3.2 Encapsulation

Algorithm 10: SMAUG.KEM.Encap: encapsulation

Input: $pk = (\text{seed}_{\boldsymbol{A}}, \boldsymbol{b})$
$\mu \leftarrow \{0,1\}^{256}$
$(K, \text{seed}) \leftarrow G(\mu, H(pk))$
$\boldsymbol{c} \leftarrow$ SMAUG.PKE.Enc$(pk, \mu; \text{seed})$
Return: $\boldsymbol{c}$, K

Algorithm 10 creates a shared secret key K by hashing together a message μ and the hash of the public key of the recipient. Only the ciphertext corresponding to μ is sent. This will be used later to, not only recover μ but also, verify the integrity of the shared key. Note that when μ is passed as a parameter in the encryption function, it is converted to a polynomial in $\mathcal{R}_t$. This algorithm has the same time complexity as the PKE version, which is $\mathcal{O}(n \cdot h \cdot k)$.

3.3.3 Decapsulation

Algorithm 11: SMAUG.KEM.Decap: decapsulation

Input: $pk = (\text{seed}_{\boldsymbol{A}}, \boldsymbol{b})$, $sk = (sk', d)$, $\boldsymbol{c}$
$\mu' \leftarrow$ SMAUG.PKE.Dec$(sk', \boldsymbol{c})$
$(K', \text{seed'}) \leftarrow G(\mu', H(pk))$
$\boldsymbol{c'} \leftarrow$ SMAUG.PKE.Enc$(pk, \mu'; \text{seed'})$
If $\boldsymbol{c} \neq \boldsymbol{c'}$ **then**
    $(K', \cdot) \leftarrow G(d, H(\boldsymbol{c}))$
end if
Return: K'

Algorithm 11 decrypts the ciphertext $\boldsymbol{c}$ to retrieve the message μ', and then uses it to reconstruct the shared key K'. It then does its own computation of the ciphertext $\boldsymbol{c}'$ and compares it to $\boldsymbol{c}$. If they are the same then the shared key is correct, if they are not, then use the seed d from the secret key to generate a different key. In the case that the latter happens, that key cannot be used for communication since the sender will have a different key. The reason for returning this fake key instead of nothing at all is discussed in 5.4, where it is found that this has been proven to be more secure. This algorithm has the same time complexity as the PKE version, which is $\mathcal{O}(n \cdot h \cdot k)$.

4 Kyber

In this section, Kyber's algorithms for both its PKE and KEM variants are explored. Kyber PKE is IND-CPA secure as proven in 6.1, and Kyber KEM is IND-CCA2 secure as proven in [5]. It should be noted that the notation used is different from the one in the original paper. The notation has been changed to match SMAUG's for easier comparison. For further simplicity the Encode and Decode operations are omitted used in the original paper, since their purpose is to transform polynomials into bytes. The parameters that are mentioned throughout this section are instantiated in Table 2. The rings from which the polynomials are taken from are defined in Section 2.2.

4.1 Design Overview

4.1.1 MLWE

Kyber relies on MLWE for all its operations, key generation and encryption. MLWE is used instead of LWE or R-LWE (Ring-LWE) because of the advantages is has over them. Even though LWE is more easily scalable it comes at the cost of efficiency, therefore making MLWE an interesting variant since it provides a trade-off between the two extremes. Moreover, with the parameters that Kyber uses it achieves a reduced structure and much better scalability than R-LWE, and for encrypting messages of 256 bits the performance is quite similar.

4.1.2 Choice of Modulo

The only number that is used for modulo operations is q which, as stated before, takes the value 3329. The reason for this is that it is a prime number which is friendly in NTT computations, meaning that the multiplications done in NTT gain efficiency, as mentioned in Section 2.11.

4.1.3 Compress/Decompress Functions

Kyber uses a modulo that requires 12 bits to be represented, and subsequently means each coefficient modulo q takes 12 bits of storage, which is more than the 10 bits SMAUG uses. In order to combat this extra size, Kyber uses a compress and decompress function which are able to discard some low-order bits in the ciphertext which do not affect the correctness probability of decryption too much. The functions are defined below.

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

When the functions are applied to polynomials, they do the operations to each coefficient individually. These operations contribute to errors when decrypting, and these effects are discussed in Section 6.2.2.

4.1.4 Hash Functions

The following hash functions are used:

- H - Is instantiated with SHA3-256 [13].
- G - Is instantiated with SHA3-512 [13].
- XOF - Is instantiated with SHAKE-128 [13].
- PRF(s, b) - Is instantiated with SHAKE-256($s \| b$) [13].
- KDF - Is instantiated with SHAKE-256 [13].

4.1.5 Sampling Algorithms

Kyber uses two different sampling algorithms: Parse and CBD. The Parse algorithm is used to generate the polynomials in $\boldsymbol{A}$ and can be seen in Algorithm 12. The algorithm is built in such a way that the polynomial that is generated is already in the NTT domain. The reason behind this is that the output polynomial is statistically close to a uniformly random element if $\mathcal{R}_q$, if the input byte array is statistically close to a uniformly random byte array [5]. This is the case since the byte stream is generated by a hash function that is assumed to have close to uniformly random outputs. On average, the time complexity of this algorithm is $\mathcal{O}(n)$ because of the while loop.

Algorithm 12: Parse

Input: Byte stream $B = b_0, b_1, b_2, \cdots \in \mathcal{B}^*$
$i \leftarrow 0$
$j \leftarrow 0$
While $j < n$ **do**
    $d_1 = b_i + 256 \cdot (b_{i+1} \mod 16)$
    $d_2 = \lfloor b_{i+1}/16 \rfloor + 16 \cdot b_{i+2}$
    If $d_1 < q$ **then**
        $a_j = d_1$
        $j \mathrel{+}= 1$
    end if
    If $d_2 < q$ **and** $j < n$ **then**
        $a_j = d_2$
        $j \mathrel{+}= 1$
    end if
    $i \mathrel{+}= 3$
end while
Return: $a_0 + a_1 X + \cdots + a_{n-1} X^{n-1}$

The CBD (Central Binomial Noise) algorithm is used to populate the polynomials in the secret key $\boldsymbol{s}$ and the error polynomials $\boldsymbol{e}$ and $\boldsymbol{r}$. For each coefficient it takes a sample $(a_1, \ldots, a_\eta, b_1, \ldots, b_\eta) \leftarrow \{0,1\}^{2\eta}$ and calculates $\sum_{i=1}^{n}(a_i - b_i)$. Since η is either 2 or 3, this means that the coefficients produced by this function are either in the range $[-2, 2]$ or $[-3, 3]$, respectively. The time complexity of this algorithm is $\mathcal{O}(n)$ because of the for loop runs 256 times, which was chosen to match the value of n.

Algorithm 13: CBD$_\eta$: central binomial distribution

Input: Byte array $B = (b_0, b_1, b_2, \ldots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
$(\beta_0, \ldots, \beta_{512\eta-14}) \leftarrow$ BytesToBits(B)
For i from 0 to 255 **do**
    $a = \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
    $b = \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
    $f_i = a - b$
end for
Return: $f_0 + f_1 X + \cdots + f_{255} X^{255}$

4.2 Specification of Kyber.CPAPKE

4.2.1 Key Generation

Algorithm 14: Kyber.CPAPKE.KeyGen: key generation

$\text{seed} \leftarrow \{0,1\}^{256}$
$(\text{seed}_{\boldsymbol{A}}, \text{seed}_{\text{sk}}) \leftarrow G(\text{seed})$
$N \leftarrow 0$
For i from 0 to $k-1$ **do**
  For j from 0 to $k-1$ **do**
    $\boldsymbol{A}[i][j] \leftarrow \text{Parse}(\text{XOF}(\text{seed}_{\boldsymbol{A}}, j, i)) \in \mathcal{R}_q$
  end for
end for
For i from 0 to $k-1$ **do**
  $\boldsymbol{s}'[i] \leftarrow \text{CBD}_{\eta_1}(\text{PRF}(\text{seed}_{\text{sk}}, N)) \in \mathcal{R}_q$
  $N \leftarrow N+1$
end for
For i from 0 to $k-1$ **do**
  $\boldsymbol{e}'[i] \leftarrow \text{CBD}_{\eta_1}(\text{PRF}(\text{seed}_{\text{sk}}, N)) \in \mathcal{R}_q$
  $N \leftarrow N+1$
end for
$\boldsymbol{s} \leftarrow \text{NTT}(\boldsymbol{s}') \in \mathcal{R}_q^k$
$\boldsymbol{e} \leftarrow \text{NTT}(\boldsymbol{e}') \in \mathcal{R}_q^k$
$\boldsymbol{b} \leftarrow \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e} \in \mathcal{R}_q^k$
Return: $pk = (\text{seed}_{\boldsymbol{A}}, \boldsymbol{b})$, $sk = \boldsymbol{s}$

Algorithm 14 generates a key-pair of a public and secret key. The public key is composed of $\text{seed}_{\boldsymbol{A}}$ and $\boldsymbol{b}$ while the secret key is only composed of $\boldsymbol{s}$. The public key contains $\text{seed}_{\boldsymbol{A}}$ instead of $\boldsymbol{A}$ to make the public key smaller, however this means that every time $\boldsymbol{A}$ needs to be used it must be recomputed. The function Parse samples $\boldsymbol{A}$ from an NTT domain. This means that any matrices or vectors that are used in computations with $\boldsymbol{A}$ must be transformed using NTT. The secret key is $\boldsymbol{s}$, which is initialized with CBD_{η_1}. CBD is a function that samples polynomials in $\mathcal{R}_q$ from a centered binomial distribution. η_1 takes the value 3 in Kyber-512 and 2 in the other variants. This determines the range of coefficients of the polynomial which are in $[-\eta_1, \eta_1]$. This also applies for CBD instantiated with η_2, which equals 2 for all versions of Kyber.

NTT is applied to both $\boldsymbol{s}$ and $\boldsymbol{e}$ to make computations with $\boldsymbol{A}$ possible. Note that because of the NTT operations on $\boldsymbol{A}$, $\boldsymbol{s}$ and $\boldsymbol{e}$; $\boldsymbol{b}$ is also in the NTT domain. The time complexity of this algorithm is $\mathcal{O}(k^2 \cdot n \log n)$ because of the k^2 polynomial multiplications in $\boldsymbol{b}$. The time complexities of the sampling algorithms are smaller than this, so they are ignored.

4.2.2 Encryption

Algorithm 15: Kyber.CPAPKE.Enc: encryption

Input: $pk = (\text{seed}_{\boldsymbol{A}}, \boldsymbol{b})$, $\mu \in \mathcal{R}_2$, seed_r
$N \leftarrow 0$
For i from 0 to $k-1$ **do**
  For j from 0 to $k-1$ **do**
    $\boldsymbol{A}^\intercal[i][j] \leftarrow \text{Parse}(\text{XOF}(\text{seed}_{\boldsymbol{A}}, j, i)) \in \mathcal{R}_q^{k \times k}$
  end for
end for
For i from 0 to $k-1$ **do**
  $\boldsymbol{r}'[i] \leftarrow \text{CBD}_{\eta_1}(\text{PRF}(\text{seed}_r, N)) \in \mathcal{R}_q^k$
  $N \leftarrow N + 1$
end for
For i from 0 to $k-1$ **do**
  $\boldsymbol{e}_1[i] \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(\text{seed}_r, N)) \in \mathcal{R}_q^k$
  $N \leftarrow N + 1$
end for
$e_2 \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(\text{seed}_r, N)) \in \mathcal{R}_q$
$\boldsymbol{r} \leftarrow \text{NTT}(\boldsymbol{r}')$
$\boldsymbol{u} \leftarrow \text{NTT}^{-1}(\boldsymbol{A}^\intercal \cdot \boldsymbol{r}) + \boldsymbol{e}_1$
$v \leftarrow \text{NTT}^{-1}(\boldsymbol{b}^\intercal \cdot \boldsymbol{r}) + e_2 + \text{Decompress}_q(\mu, 1)$
$c_1 \leftarrow \text{Compress}_q(\boldsymbol{u}, d_u)$
$c_2 \leftarrow \text{Compress}_q(v, d_v)$
Return: $\boldsymbol{c} = (c_1, c_2)$

Algorithm 15 above encrypts a message μ and outputs a vector $\boldsymbol{c}$ which contains the ciphertext. The vector $\boldsymbol{r}$ is used to introduce randomness when encrypting, and uses the vector together with a small error vector $\boldsymbol{e}_1$ to produce $\boldsymbol{u}$. v is used to encrypt the message, however $\boldsymbol{c}_1$ is needed so that the secret key can be used to cancel out $\boldsymbol{b}$. Here μ is compressed to scale the polynomial as it was explained in the MLWE example in Section 2.10. Finally, $\boldsymbol{u}$ and v are compressed to reduce the size of the ciphertext.

The parameters d_u and d_v take the values 10 or 11 and 4 or 5 respectively for different variants of Kyber. This means that c_1 is compressed to a 10 or 11 bit number, while c_2 is compressed to a 4 or 5 bit number. The time complexity of this algorithm is $\mathcal{O}(k^2 \cdot n \log n)$ because there are k^2 polynomial multiplications in $\boldsymbol{u}$.

4.2.3 Decryption

Algorithm 16: Kyber.CPAPKE.Dec: decryption

Input: $sk = \boldsymbol{s}$, $\boldsymbol{c} = (c_1, c_2)$
$\boldsymbol{u} \leftarrow \text{Decompress}_q(c_1, d_u)$
$v \leftarrow \text{Decompress}_q(c_2, d_v)$
$\mu' \leftarrow \text{Compress}_q(v - \text{NTT}^{-1}(\boldsymbol{s}^\intercal \cdot \text{NTT}(\boldsymbol{u})), 1)$
Return: μ'

Algorithm 16 above decrypts the ciphertext received with the help of the secret key. The time complexity of this algorithm is $\mathcal{O}(n \log n)$ because of the single polynomial multiplication. It will now be shown that this algorithm does indeed recover the message when the errors are ignored. C_q and D_q are used as shorthand notation of Compress_q and Decompress_q respectively.

$$\begin{aligned}\mu' =& C_q(v - \text{NTT}^{-1}(\boldsymbol{s}^\intercal \cdot \text{NTT}(\boldsymbol{u})), 1) \\ =& C_q(D_q(C_q(\text{NTT}^{-1}(\boldsymbol{b}^\intercal \cdot \boldsymbol{r}) + e_2 + D_q(\mu, 1), d_v), d_v) \\ & - \text{NTT}^{-1}(\boldsymbol{s}^\intercal \cdot \text{NTT}(D_q(C_q(\text{NTT}^{-1}(\boldsymbol{A}^\intercal \cdot \boldsymbol{r}) + \boldsymbol{e}_1, d_u), d_u))), 1)\end{aligned}$$

The compress and decompress operations are ignored for now, but they introduce some errors in decryption.

$$\begin{aligned}\mu' &= \text{NTT}^{-1}(\boldsymbol{b}^\intercal \cdot \boldsymbol{r}) + e_2 + \mu - \text{NTT}^{-1}(\boldsymbol{s}^\intercal \cdot \text{NTT}(\text{NTT}^{-1}(\boldsymbol{A}^\intercal \cdot \boldsymbol{r}) + \boldsymbol{e}_1)) \\ &= \text{NTT}^{-1}(\boldsymbol{b}^\intercal \cdot \boldsymbol{r}) + \mu - \text{NTT}^{-1}(\boldsymbol{s}^\intercal \cdot \boldsymbol{A}^\intercal \cdot \boldsymbol{r}) - \text{NTT}^{-1}(\boldsymbol{s}^\intercal \cdot \boldsymbol{e}_1) + e_2 \\ &= \text{NTT}^{-1}(\boldsymbol{b}^\intercal \cdot \boldsymbol{r} - \boldsymbol{s}^\intercal \cdot \boldsymbol{A}^\intercal \cdot \boldsymbol{r}) + \mu - \boldsymbol{s}'^\intercal \cdot \text{NTT}^{-1}(\boldsymbol{e}_1) + e_2 \\ &= \text{NTT}^{-1}(\text{NTT}(\boldsymbol{b}'^\intercal \cdot \boldsymbol{r}') - \text{NTT}(\boldsymbol{s}'^\intercal \cdot \boldsymbol{A}'^\intercal \cdot \boldsymbol{r}')) + \mu - \boldsymbol{s}'^\intercal \cdot \text{NTT}^{-1}(\boldsymbol{e}_1) + e_2\end{aligned}$$

Note that there were no $\boldsymbol{A}'$ and $\boldsymbol{b}'$ in the key generation process, however since both $\boldsymbol{A}$ and $\boldsymbol{b}$ were by definition in the NTT domain, $\boldsymbol{A}'$ and $\boldsymbol{b}'$ are created to demonstrate that these are the non-NTT versions of $\boldsymbol{A}$ and $\boldsymbol{b}$ respectively. Therefore $\boldsymbol{A}' = \text{NTT}^{-1}(\boldsymbol{A})$ and $\boldsymbol{b}' = \text{NTT}^{-1}(\boldsymbol{b}) = \boldsymbol{A}' \cdot \boldsymbol{s}' + \boldsymbol{e}'$.

$$\begin{aligned}\mu' &= \boldsymbol{b}'^\intercal \cdot \boldsymbol{r}' - \boldsymbol{s}'^\intercal \cdot \boldsymbol{A}'^\intercal \cdot \boldsymbol{r}' + \mu - \boldsymbol{s}'^\intercal \cdot \text{NTT}^{-1}(\boldsymbol{e}_1) + e_2 \\ &= (\boldsymbol{A}' \cdot \boldsymbol{s}' + \boldsymbol{e}')^\intercal \cdot \boldsymbol{r}' - \boldsymbol{s}'^\intercal \cdot \boldsymbol{A}'^\intercal \cdot \boldsymbol{r}' + \mu - \boldsymbol{s}'^\intercal \cdot \text{NTT}^{-1}(\boldsymbol{e}_1) + e_2 \\ &= \boldsymbol{s}'^\intercal \cdot \boldsymbol{A}'^\intercal \cdot \boldsymbol{r}' + \boldsymbol{e}'^\intercal \cdot \boldsymbol{r}' - \boldsymbol{s}'^\intercal \cdot \boldsymbol{A}'^\intercal \cdot \boldsymbol{r}' + \mu - \boldsymbol{s}'^\intercal \cdot \text{NTT}^{-1}(\boldsymbol{e}_1) + e_2 \\ &= \mu + \underline{\boldsymbol{e}'^\intercal \cdot \boldsymbol{r}' - \boldsymbol{s}'^\intercal \cdot \text{NTT}^{-1}(\boldsymbol{e}_1) + e_2}\end{aligned}$$

Note that the remaining terms except for the message are small by definition, therefore perturbing slightly the message. A method like the one from Section 2.10 can be used to retrieve the original μ', so decryption works as intended. The effect of these little errors and the errors introduced in the compress and decompress functions are discussed in Section 6.2.2.

4.3 Specification of Kyber.CCAKEM

4.3.1 Key Generation

Algorithm 17: Kyber.CCAKEM.KeyGen: key generation

1 $(pk, sk') \leftarrow$ Kyber.CPAPKE.KeyGen()
2 $d \leftarrow \{0,1\}^{256}$
3 **Return:** pk, $sk = (sk', pk, H(pk), d)$

The only difference in this algorithm compared to the PKE version is what is stored in the private key. Aside from the original value of the secret key, it now stores the public key, the hash of the public key and the random 256-bit seed d. The reason for this will be discussed later. The time complexity of this algorithm is the same as its PKE version, therefore it is $\mathcal{O}(k^2 \cdot n \log n)$.

4.3.2 Encapsulation

Algorithm 18: Kyber.CCAKEM.Encap: encapsulation

Input: $pk = (\text{seed}_{\boldsymbol{A}}, \boldsymbol{b})$
$\mu \leftarrow \{0,1\}^{256}$
$\mu \leftarrow H(\mu)$
$(K, \text{seed}) \leftarrow G(\mu, H(pk))$
$\boldsymbol{c} \leftarrow$ Kyber.CPAPKE.Enc$(pk, \mu; \text{seed})$
$K \leftarrow \text{KDF}(K, H(\boldsymbol{c}))$
Return: $\boldsymbol{c}$, K

Algorithm 18 creates a provisional shared secret key K by hashing together the hash of a message μ and the hash of the public key of the recipient. Only the ciphertext corresponding to μ is sent. The ciphertext serves multiple purposes, its used to recover the message μ by the receiver, used to create the final shared secret key K and to verify the integrity of the shared key by the receiver. Note that when μ is passed as a parameter in the encryption function, it is converted to a polynomial in $\mathcal{R}_t$. The time complexity of this algorithm is the same as its PKE version, therefore it is $\mathcal{O}(k^2 \cdot n \log n)$.

4.3.3 Decapsulation

Algorithm 19: Kyber.CCAKEM.Decap: decapsulation

Input: $sk = (sk', pk, H(pk), d)$, $\boldsymbol{c}$
$\mu' \leftarrow$ Kyber.CPAPKE.Dec$(sk', \boldsymbol{c})$
$(K', \text{seed'}) \leftarrow G(\mu', H(pk))$
$\boldsymbol{c'} \leftarrow$ Kyber.CPAPKE.Enc$(pk, \mu'; \text{seed'})$
If $\boldsymbol{c} = \boldsymbol{c'}$ **then**
    $(K', \cdot) \leftarrow \text{KDF}(K, H(\boldsymbol{c}))$
else
    $(K', \cdot) \leftarrow \text{KDF}(d, H(\boldsymbol{c}))$
end if
Return: K'

Algorithm 19 decrypts the ciphertext $\boldsymbol{c}$ to retrieve the message μ', and then uses it to reconstruct the provisional shared key K'. It then does its own computation of the ciphertext $\boldsymbol{c}'$ and compares it to $\boldsymbol{c}$. If they are the same then it computes the final shared key, if they are not, then it uses the seed d from the secret key to generate a different key. In the case that the latter happens, that key cannot be used for communication since the sender will have a different key. The time complexity of this algorithm is the same as its PKE version, therefore it is $\mathcal{O}(n \log n)$.

5 SMAUG vs. Kyber

In this section, the differences between SMAUG and Kyber will be studied. The design rationale behind their differences will also be explained. A comparison between the algorithms from the previous sections will be performed, as well as a comparison on the parameter sets of both systems.

5.1 Parameters Sets

Tables 1 and 2 show the parameter sets of SMAUG and Kyber as well as other interesting information. Their different approaches might make it seem that comparison may be futile, however there are some interesting insights that can be obtained regardless.

Firstly, the focus will be in parameter q which is common to both schemes. In Kyber, it is used throughout the entire PKE scheme, however in SMAUG it has more relevance in the key generation. The value of q is bigger in Kyber than in SMAUG, which means more storage must be dedicated for Kyber. In order to represent numbers in $[0, 3328]$ 12 bits are needed as opposed to SMAUG's 10 and 11 bits for $[0, 1023]$ and $[0, 2047]$ respectively.

Secondly, a comparison between parameters q, p and p' is made. Their importance lies in the encryption process and the subsequent sizes of the ciphertext. Even though both p and p' are much smaller than q, the ciphertext size is not that different. The reason for this is that Kyber uses a compress function with parameters d_u and d_v, and this function essentially turns each coefficient into an integer in $[0, 2^d - 1]$. For example, $\boldsymbol{c}_1$ in SMAUG has coefficients each with 8-bit numbers given the value of p. For Kyber-512 and Kyber-768, it would have been a 12-bit number given the value of q, however because of the compress function with parameter d_u the coefficient will be in $[0, 1023]$. This means that the coefficients can be represented with a 10-bit number. A similar argument holds for p' and d_v for c_2, however, in this case Kyber has the smaller size since it only requires 4 bits for Kyber-512 and Kyber-768 and 5 bits for Kyber-1024. This is opposed to SMAUG's need for 5, 8 and 6 bits of storage for c_2 in SMAUG-128, SMAUG-192 and SMAUG-256 respectively.

Finally, the decryption failure probabilities (DFP) are higher in SMAUG than in Kyber. When decryption fails, it suggests a correlation between the secret key and the randomness used for encryption, which inadvertently discloses information about the key. This is because the secret key is part of the errors in decryption and the secret key is accompanied by the random noise, like $\boldsymbol{e}$, as seen in Section 6.2.1 and 6.2.2. In other words, decryption failure may leak sensitive details about the secret key or the encryption randomness, compromising the security of the encryption scheme [3]. However the main problem is the secret key, since the encryption randomness is sampled freshly each time, while the secret key is fixed and knowledge about it accumulates. The results in the table are presented with a log base 2, so in reality the probabilities are 2^{DFP} and all probabilities can be considered practically insignificant. There are multiple factors that explain why the DFPs of SMAUG are higher. The main one is that, as it is seen in Section 6.2, Kyber's scaling of μ using the compress and decompress functions allows for more flexibility and erases some errors. However, in SMAUG this is not the case so all the error terms affect the DFP.

Parameter sets Security level	SMAUG-128 I	SMAUG-192 III	SMAUG-256 V
n	256	256	256
k	2	3	5
q	1024	2048	2048
p	256	256	256
p'	32	256	64
t	2	2	2
h_s	140	198	176
h_r	132	151	160
σ	1.0625	1.453713	1.0625
DFP	-119.6	-136.1	-167.2
Secret key	176	236	218
Public key	672	1088	1792
Ciphertext	672	1024	1472

Table 1: Parameter sets and other information from SMAUG

Parameter sets Security level	Kyber-512 I	Kyber-768 III	Kyber-1024 V
n	256	256	256
k	2	3	4
q	3329	3329	3329
η_1	3	2	2
η_2	2	2	2
d_u	10	10	11
d_v	4	4	5
DFP	-139	-164	-174
Secret key	1632	2400	3168
Public key	800	1184	1568
Ciphertext	768	1088	1568

Table 2: Parameter sets and other information from Kyber

5.2 Key Generation (PKE)

To see the related algorithms please refer to 3.2.1 and 4.2.1.
The first difference that can be seen is in how $\boldsymbol{A}$ is generated. While both SMAUG and Kyber use algorithms that sample from a uniformly random distribution over $\mathcal{R}_q^{k\times k}$, Kyber's algorithm Parse does so in a way that the elements of $\boldsymbol{A}$ are in NTT representation. Kyber uses NTT to benefit from faster polynomial multiplication as explained in Section 2.11, and can do so because of how they choose q. SMAUG however, given how they chose q, cannot benefit from NTT, since their moduli are in powers of two. Therefore, SMAUG ideally should use a multiplication system which is of similar efficiency. A closer comparison of these techniques is studied in Section 5.5, where it was found that SMAUG's polynomial multiplication is much slower than Kyber's.

The next difference is how their secret keys are generated. SMAUG uses a sparse secret instead of a traditional secret like Kyber does. This means that SMAUG's private keys are considerably smaller

than Kyber's. This is very beneficial storage wise, however it may compromise its security as it was pointed out by Daniel Bernstein in [7]. The paper that he makes reference to, and the attack it contains is explained in Section 6.4. Sparse-LWE is a less studied problem than LWE (and its modular counterparts), therefore more research must be made in order to verify its long term security.

Errors are also sampled differently, with SMAUG using a discrete gaussian distribution and Kyber using a centered binomial distribution. The exact distribution that is used for sampling noise is not important for attacks, but the standard deviation is. This is because the standard deviation of the noise distribution determines the amount of uncertainty that an attacker has about the original data. Therefore it should be chosen appropriately to maximize this uncertainty, while still being efficient. There is no specific mention in SMAUG in the reason for the values for the standard deviation they use, but they are probably chosen taking the considerations above.

5.3 Encryption and Decryption (PKE)

To see the related algorithms please refer to 3.2.2 and 4.2.2 for encryption and, 3.2.3 and 4.2.3 for decryption.
The main change between Kyber and SMAUG in the encryption process is that Kyber uses MLWE and SMAUG MLWR. SMAUG claims that using MLWR for encryption allows for faster operations than MLWE. For both MLWE and MLWR, a vector $\boldsymbol{r}$ is created to be multiplied by $\boldsymbol{A}$ and used in the inner product with $\boldsymbol{b}$. However, in MLWE an additional vector $\boldsymbol{e}_1$ and polynomial e_2 have to be created to use in the encryption and then added. On the other hand, MLWR only needs to do one scalar multiplication and rounding operation.

Furthermore, SMAUG uses powers-of-two moduli, which means that the scalar multiplication can be accomplished by bit shifting operations. In SMAUG-128 $p = 256$ and $q = 1024$, therefore the scalar multiplication is with $\frac{1}{4}$ which means shifting the bits two places to the right. Additionally, the coefficients of the polynomials can be easily computed by using bit AND operations. For example, imagine a coefficient is $777_{10} = (1100001001)_2$, then to do this mod $p = 256$, the binary representation of 777 needs to be used with the AND bitwise operator with the binary representation of $255_{10} = (11111111)_2$:

$$\begin{aligned} &1100001001 \\ \text{AND } &0011111111 \\ &0000001001 \end{aligned}$$

which is $(1001)_2 = 9_{10}$ and indeed 777 mod 256 is 9. Therefore, not only is MLWR typically faster than MLWE, but SMAUG's choice of parameters make it particularly more efficient. Not much can be said about the decryption processes in terms of differences since they are entirely dependent on how the encryption process were constructed.

5.4 KEM Algorithms

To see the related algorithms please refer to 3.3.1 and 4.3.1 for key generation, 3.3.2 and 4.3.2 for encapsulation, and 3.3.3 and 4.3.3 for decapsulation.
Both SMAUG and Kyber use an almost identical Fujisaki Okamoto transform [14] to turn their PKE schemes into KEMs. During key generation, Kyber chooses to store the public key and its hash in the secret key. The reason for this is to save computation time and additional calls to improve efficiency. The downside of this is that it takes more space but it means that in this aspect Kyber is more efficient than SMAUG in terms of speed.

SMAUG uses a special type of FO transform $\mathsf{U}^{\not\perp}_{m}$ which has implicit rejection ($\not\perp$) and does not use ciphertext in generating the symmetric key (m). Kyber uses FO transform $\mathsf{U}^{\not\perp}$ which also has implicit rejection, but does use ciphertext in generating the symmetric key. Implicit rejection means that when the ciphertext comparison fails, instead of returning a failure symbol ($\perp$), a fake symmetric key is returned using the extra bits appended to the secret key during key generation. This has allowed for higher security in the past [17]. One of the differences between these two transforms is that SMAUG's FO transform allows for faster operations, since the ciphertext is not computed into the shared key.

Additionally, Kyber does an additional hash to the shared secret key K and hashes together the previous value of K with the hash of the ciphertext. Doing this still means that both SMAUG and Kyber are IND-CCA2 secure, however it could make Kyber more secure. This is because the shared secret key is built upon a full "view" of the protocol, which means that it uses all the data that was created or sent during this process. The disadvantage to this is loss in efficiency, therefore even though Kyber's KEM might be more secure, SMAUG's KEM is more efficient.

There is another difference in the encapsulation and decapsulation processes. When Kyber creates the secret message that is going to be sent, it first hashes it before encrypting it. While it is never explicitly explained why they chose to do that, it might add robustness by adding more randomness. SMAUG not doing this is not a concern since the actual value of the secret is not what is important. What is important is that the value of the secret changes sufficiently between encapsulations.

5.5 Polynomial Multiplication

Both systems have different approaches to polynomial multiplication, Kyber takes advantage of its modulo choice to use NTT while SMAUG takes advantage of its sparse secret structure. In previous sections, it was stated that the time complexity for each polynomial multiplication is $\mathcal{O}(\text{len}(b) \cdot (1 + n) + n)$ for SMAUG and $\mathcal{O}(n \log n)$ for Kyber. However, this makes it challenging for comparison since $\text{len}(b)$ changes depending on how many nonzero coefficients each polynomial has. Since it is known how many nonzero coefficients an entire vector has, the entire multiplication of vectors and matrices will be considered for comparison. For simplicity, SMAUG-128 and Kyber-512 are considered.

$\boldsymbol{A} \cdot \boldsymbol{s}$. This multiplication performs 4 polynomial multiplications, making Kyber's running time $4 \cdot n \log n$. The SMAUG multiplications can be seen in groups of 2: $\boldsymbol{A}_{11} \cdot \boldsymbol{s}_1 + \boldsymbol{A}_{12} \cdot \boldsymbol{s}_2$ and $\boldsymbol{A}_{21} \cdot \boldsymbol{s}_1 + \boldsymbol{A}_{22} \cdot \boldsymbol{s}_2$. For each group, h can be used instead of $\text{len}(b)$ since the lengths of both $b's$ in each $\boldsymbol{s}_i$ will add up to h. Therefore, the running time for SMAUG would be $2 \cdot (h \cdot (1 + n) + 2 \cdot n)$. Substituting in $h = 140$ and $n = 256$ give the results below.

$$\begin{aligned} \text{SMAUG:} &= 2 \cdot (140 \cdot (1 + 256) + 2 \cdot 256) \\ &= 72984 \\ \text{Kyber:} &= 24 \cdot 256 \cdot \log(256) \\ &= 5678.26\ldots \end{aligned}$$

Therefore, Kyber is more than one order of magnitude faster than SMAUG.

$\boldsymbol{A} \cdot \boldsymbol{r}$. A similar methodology is used, however in this case $h = 132$, so the results change but not in a significant way.

$$\begin{aligned} \text{SMAUG:} &= 2 \cdot (132 \cdot (1 + 256) + 2 \cdot 256) \\ &= 68872 \\ \text{Kyber:} &= 4 \cdot 256 \cdot \log(256) \\ &= 5678.26\ldots \end{aligned}$$

Even though it is faster than before, there is still an order of magnitude of difference.

$\langle \boldsymbol{b}, \boldsymbol{r} \rangle$ or $\boldsymbol{b}^\intercal \cdot \boldsymbol{r}$. Both describe the same operation, and it includes two polynomial multiplications. In SMAUG this means it is one polynomial multiplication, but using h instead of len(b) as explained before with $h = 132$. Then, the resulting running times are as follows.

$$\begin{aligned} \text{SMAUG:} &= (132 \cdot (1 + 256) + 2 \cdot 256) \\ &= 34436 \\ \text{Kyber:} &= 256 \cdot \log(256) \\ &= 1419.57\ldots \end{aligned}$$

Once again, even though this is much faster than previous examples, there is still an order of magnitude of difference. This is a puzzling result since SMAUG claims to be much faster than Kyber. A few reasons can explain this disparity. The first one is that this is based in theoretical time complexities from the multiplication algorithm. It could be that the practical implementation of the algorithm is faster than what the pseudo-code of the algorithm suggests. Another reason is that all the other changes that SMAUG has compensate for this lack of efficiency. However, right now it is unclear what operations exactly provide this efficiency gain over Kyber.

6 Security

In this section, the proofs for the IND-CPA security of both SMAUG and Kyber are provided. Then, attacks that target the SVP are stated in a lattice and its dual. Finally, a combinatorial attack that aims to find the secret vector $\boldsymbol{s}$ and its developments are explained.

6.1 IND-CPA Security Proof

The proofs of the IND-CPA security of SMAUG.PKE and Kyber.PKE are the only provided proofs of security, since the proofs for KEMs constructed from FO transforms are essentially the same for all of them. Additionally, the security of such a proof relies on the security of its underlying PKE scheme. The following proof for SMAUG.PKE can be found as Theorem 4 in Section 4.3 in [10].

IND-CPA security of SMAUG.PKE. *Assuming pseudorandomness of the underlying sampling algorithms, the IND-CPA security of SMAUG.PKE can be tightly reduced to the decisional MLWE and MLWR problems. Specifically, for any IND-CPA-adversary $\mathcal{A}$ of SMAUG.PKE, there exist adversaries $\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2$, and $\mathcal{B}_3$ attacking the pseudorandomness of XOF and the sampling algorithms, MLWE, and MLWR problems, such that,*

$$\begin{aligned}\mathsf{Adv}^{\text{IND-CPA}}_{\text{SMAUG.PKE}}(\mathcal{A}) \leq & \mathsf{Adv}^{\text{PR}}_{\text{XOF}}(\mathcal{B}_0) + \mathsf{Adv}^{\text{PR}}_{\text{expandA,HWT,dGaussian}}(\mathcal{B}_1) \\ & + \mathsf{Adv}^{\text{MLWE}}_{n,q,k,k,\text{HWT}_{h_s},\text{dGaussian}}(\mathcal{B}_2) + \mathsf{Adv}^{\text{MLWR}}_{n,p,q,k+1,k,\text{HWT}_{h_r}}(\mathcal{B}_3)\end{aligned}$$

Proof. Define a sequence of hybrid games from G_0 to G_4 as follows:

- G_0: the genuine IND-CPA game,
- G_1: identical to G_0, except that the public key is changed into $(\boldsymbol{A}, \boldsymbol{b})$,
- G_2: identical to G_1, except that the sampling algorithms are changed into truly random samplings,
- G_3: identical to G_2, except that $\boldsymbol{b}$ is randomly chosen from $\mathcal{R}_q^k$,
- G_4: identical to G_3, except that the ciphertext is randomly chosen from $\mathcal{R}_p^k \times \mathcal{R}_{p'}$. As a result, the public key and the ciphertexts are truly random.

Furthermore, define the advantage of the adversary on each game G_i as Adv_i, where $\mathsf{Adv}_0 = \mathsf{Adv}^{\text{IND-CPA}}_{\text{SMAUG.PKE}}(\mathcal{A})$ and $\mathsf{Adv}_4 = 0$. Then, it holds that

$$|\mathsf{Adv}_0 - \mathsf{Adv}_1| \leq \mathsf{Adv}^{\text{PR}}_{\text{XOF}}(\mathcal{B}_0)$$

for some adversary $\mathcal{B}_0$ against some pseudorandomness (PR) of the hash function (XOF). The reason for this is that since the difference between G_0 and G_1 is how the public key is defined, the focus is the function XOF that initializes the seed of $\boldsymbol{A}$, the secret $\boldsymbol{s}$ and the error $\boldsymbol{e}$, which are subsequently used to define $\boldsymbol{b}$. Next, it also holds that

$$|\mathsf{Adv}_1 - \mathsf{Adv}_2| \leq \mathsf{Adv}^{\text{PR}}_{\text{expandA,HWT,dGaussian}}(\mathcal{B}_1)$$

for some adversary, $\mathcal{B}_1$ attacking the pseudorandomness of at least one of the samplers. This is because the only difference between G_1 and G_2 is that instead of using the SMAUG sampling algorithms, truly random sampling algorithms are used. The next statement also holds,

$$|\mathsf{Adv}_2 - \mathsf{Adv}_3| \leq \mathsf{Adv}^{\text{MLWE}}_{n,q,k,k,\text{HWT}_{h_s},\text{dGaussian}}(\mathcal{B}_2)$$

where $\mathcal{B}_2$ is an adversary against the decisional MLWE problem, distinguishing the MLWE samples from random. Indeed, this holds since the difference from G_2 and G_3 is that $\boldsymbol{b}$ is sampled as an MLWE sample in G_2 and randomly in G_3. Finally, it holds that

$$|\mathsf{Adv}_3 - \mathsf{Adv}_4| \leq \mathsf{Adv}^{\mathsf{MLWR}}_{n,p,q,k+1,k,\mathsf{HWT}_{h_r}}(\mathcal{B}_3)$$

where $\mathcal{B}_3$ is an adversary distinguishing the MLWR sample from random. This comes from the difference in G_3 and G_4 being how the ciphertexts are generated; as $(\boldsymbol{c}_1, \lfloor p'/p \cdot c_2 \rceil)$ in G_3 versus randomly over $\mathcal{R}_p^k \times \mathcal{R}_{p'}$ in G_4, where

$$\begin{bmatrix} \boldsymbol{c}_1 \\ c_2 \end{bmatrix} = \left\lfloor \frac{p}{q} \cdot \begin{pmatrix} \boldsymbol{A} \\ \boldsymbol{b}^\mathsf{T} \end{pmatrix} \cdot \boldsymbol{r} \right\rceil + \frac{p}{t} \cdot \begin{bmatrix} 0 \\ \mu \end{bmatrix}.$$

Using all the definitions above the original statement can be proven as:

$$\begin{aligned}
\mathsf{Adv}_0 &= \mathsf{Adv}_0 - \mathsf{Adv}_4 \\
&\leq |\mathsf{Adv}_0 - \mathsf{Adv}_4| \\
&\leq |\mathsf{Adv}_0 - \mathsf{Adv}_1 + \mathsf{Adv}_1 - \mathsf{Adv}_2 + \mathsf{Adv}_2 - \mathsf{Adv}_3 + \mathsf{Adv}_3 - \mathsf{Adv}_4| \\
&\leq |\mathsf{Adv}_0 - \mathsf{Adv}_1| + |\mathsf{Adv}_1 - \mathsf{Adv}_2| + |\mathsf{Adv}_2 - \mathsf{Adv}_3| + |\mathsf{Adv}_3 - \mathsf{Adv}_4| \\
\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{SMAUG.PKE}}(\mathcal{A}) &\leq \mathsf{Adv}^{\mathsf{PR}}_{\mathsf{XOF}}(\mathcal{B}_0) + \mathsf{Adv}^{\mathsf{PR}}_{\mathsf{expandA,HWT,dGaussian}}(\mathcal{B}_1) \\
&\quad + \mathsf{Adv}^{\mathsf{MLWE}}_{n,q,k,k,\mathsf{HWT}_{h_s},\mathsf{dGaussian}}(\mathcal{B}_2) + \mathsf{Adv}^{\mathsf{MLWR}}_{n,p,q,k+1,k,\mathsf{HWT}_{h_r}}(\mathcal{B}_3)
\end{aligned}$$

This concludes the proof. □

Next, the proof for Kyber.PKE is provided, however this proof is not provided in [5] but can be constructed in a similar way as SMAUG did.

IND-CPA security of Kyber.PKE. *Suppose XOF and G are random oracles. For any adversary $\mathcal{A}$, there exists adversaries $\mathcal{B}$ and $\mathcal{C}$ with roughly the same running time as that of $\mathcal{A}$ such that,*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{Kyber.CPAPKE}}(\mathcal{A}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{MLWE}}_{k+1,k,\eta}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{PR}}_{\mathsf{PRF}}(\mathcal{C})$$

Proof. Define a sequence of hybrid games from G_0 to G_3 as follows:

- G_0: the genuine IND-CPA game,
- G_1: identical to G_0, except that the public key is changed into $(\boldsymbol{A}, \boldsymbol{b})$,
- G_2: identical to G_1, except that $\boldsymbol{b}$ is randomly chosen from $\mathcal{R}_q^k$,
- G_3: identical to G_2, except that the ciphertext is randomly chosen from $\mathcal{R}_p^k \times \mathcal{R}_{p'}$. As a result, the public key and the ciphertexts are truly random.

Furthermore, define the advantage of the adversary on each game G_i as Adv_i, where $\mathsf{Adv}_0 = \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{Kyber.CPAPKE}}(\mathcal{A})$ and $\mathsf{Adv}_3 = 0$. Then, it holds that

$$|\mathsf{Adv}_0 - \mathsf{Adv}_1| \leq \mathsf{Adv}^{\mathsf{PR}}_{\mathsf{prf}}(\mathcal{C})$$

for some adversary $\mathcal{C}$ against some pseudorandomness of the hash function (PRF). The reason for this is that since the difference between G_0 and G_1 is how the public key is defined, the focus is the function PRF that initializes the seed of the secret $\boldsymbol{s}$ and error $\boldsymbol{e}$, which is subsequently used to define $\boldsymbol{b}$. Next, it also holds that

$$|\mathsf{Adv}_1 - \mathsf{Adv}_2| \leq \mathsf{Adv}^{\mathsf{MLWE}}_{k+1,k,\eta}(\mathcal{B})$$

where $\mathcal{B}$ is an adversary against the decisional MLWE problem, distinguishing the MLWE samples from random. Indeed, this holds since the difference from G_1 and G_2 is that $\boldsymbol{b}$ is sampled as an MLWE sample in G_1 and randomly in G_2. Finally, it holds that

$$|\mathsf{Adv}_2 - \mathsf{Adv}_3| \leq \mathsf{Adv}^{\mathsf{MLWE}}_{k+1,k,\eta}(\mathcal{B})$$

where again $\mathcal{B}$ is an adversary distinguishing the MLWE sample from random. This comes from the difference in G_2 and G_3 being how the ciphertexts are generated; as $(\boldsymbol{c}_1, c_2)$ in G_2 versus randomly over $\mathcal{R}_q^k \times \mathcal{R}_q$ in G_3, where

$$\begin{bmatrix} \boldsymbol{c}_1 \\ c_2 \end{bmatrix} = \mathsf{NTT}^{-1}\left(\begin{pmatrix} \boldsymbol{A}^\intercal \\ \boldsymbol{b}^\intercal \end{pmatrix} \cdot \boldsymbol{r}\right) + \begin{bmatrix} \boldsymbol{e}_1 \\ e_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \mu \end{bmatrix}.$$

Using all the definitions above the original statement can be proven as:

$$\begin{aligned}
\mathsf{Adv}_0 &= \mathsf{Adv}_0 - \mathsf{Adv}_3 \\
&\leq |\mathsf{Adv}_0 - \mathsf{Adv}_3| \\
&\leq |\mathsf{Adv}_0 - \mathsf{Adv}_1 + \mathsf{Adv}_1 - \mathsf{Adv}_2 + \mathsf{Adv}_2 - \mathsf{Adv}_3| \\
&\leq |\mathsf{Adv}_0 - \mathsf{Adv}_1| + |\mathsf{Adv}_1 - \mathsf{Adv}_2| + |\mathsf{Adv}_2 - \mathsf{Adv}_3| \\
&\leq \mathsf{Adv}^{\mathsf{PR}}_{\mathsf{prf}}(\mathcal{C}) + \mathsf{Adv}^{\mathsf{MLWE}}_{k+1,k,\eta}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{MLWE}}_{k+1,k,\eta}(\mathcal{B}) \\
\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{Kyber.CPAPKE}}(\mathcal{A}) &\leq 2 \cdot \mathsf{Adv}^{\mathsf{MLWE}}_{k+1,k,\eta}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{PR}}_{\mathsf{prf}}(\mathcal{C})
\end{aligned}$$

This concludes the proof. □

There are only two differences in the approach of this proof compared to SMAUG. Firstly, there is one game less to consider since the sampling algorithms in Kyber are all based on the hash functions. Secondly, PRF is the only hash function considered in the proof since, even though XOF is used in the generation of $\boldsymbol{A}$, it is assumed in the statement as a random oracle, so it is not pseudorandom.

6.2 Decryption Failure Probability

The encryption and decryption processes carry inherent errors in their operations, which means that in "unlucky" situations decryption will fail. The errors for both SMAUG and Kyber are revisited now, while giving a bound for those errors.

6.2.1 SMAUG

The error in decryption for SMAUG is majorly determined by the errors mentioned in Section 3.2.3 which are shown below.

$$E_S = t/q \cdot \langle \boldsymbol{e}, \boldsymbol{r} \rangle - t/p \cdot \langle \boldsymbol{\epsilon_1}, \boldsymbol{s} \rangle - t/p' \cdot \epsilon_2 - \epsilon_3$$

where $\boldsymbol{\epsilon_1} \in [-0.5, 0.5)^k$ and $\epsilon_2, \epsilon_3 \in [-0.5, 0.5)$. An attempt was made to calculate the individual bounds of each term, but that would require calculating what would be the worst case for the coefficients in polynomial multiplication which is a lengthy task. Due to the time constraints of this project, this was not completed. Whatever the result of this bound would have been, it would define how much the message would get distorted in decryption. If the message sent was 45, this error could be small enough to decrypt to 44 which, even though it is a small difference, is still incorrect. Therefore, in order to avoid a decryption failure, E_S should be the zero polynomial or the coefficients should be very close to 0.

6.2.2 Kyber

Determining the bound for the error introduced in decryption for Kyber is even more complicated than in SMAUG. This is because aside from the error terms, the compress and decompress functions introduce further errors. The equation below describes the errors in Kyber without the compress and decompress functions, as seen in Section 4.2.3.

$$E_K = \boldsymbol{e}'^{\intercal} \cdot \boldsymbol{r}' - \boldsymbol{s}'^{\intercal} \cdot \mathsf{NTT}^{-1}(\boldsymbol{e}_1) + e_2$$

As in SMAUG, it is very challenging to get numerical bounds from this equation, however, the loss in the compress and decompress functions can be discussed.

The compress and decompress functions allow a number to have a representation using fewer numbers. The fewer numbers used in this representation, the less precise the retrieval of the original number. For example, if a 12 bit number is mapped to a 10 bit number, then the accuracy loss will be little. However, if a 12 bit number is mapped to a 4 bit number, the accuracy loss is great.

Take for example 2979. Then compressing it to a 10 bit number returns 916, and decompressing it again returns 2978. However, if it is compressed to a 4 bit number, the compress function returns 14 and decompressing it back returns 2913. So it can be seen that there is a big loss of accuracy. For smaller cases like the first one, this will not result in a decryption failure like in SMAUG. This is because the compress and decompress function that are used on the message μ ensure that small variations like those are not noticeable, but it cannot solve big alterations. Therefore the shrinking parameters have to be chosen carefully to reduce the decryption failure probability.

6.3 Core-SVP Methodology

The core-SVP methodology evaluates the core-SVP hardness of mathematical problems that are based around the SVP problem, such as LWE (and its variants). The core-SVP hardness is defined as the cost of one call to an SVP oracle in dimension β [4], where an SVP oracle allows an attacker to efficiently solve the SVP on a given lattice. The dimension β refers to the block size used in the BKZ algorithm, which is an algorithm used as the SVP oracle for core-SVP hardness measurements. The next subsections provide descriptions on some common attacks considered in the context of lattices.

6.3.1 Primal Attack - BKZ variant

Consider an LWE instance $(\boldsymbol{A}, \boldsymbol{b}) \in \mathbb{Z}_q^{k \times k} \times \mathbb{Z}_q^k$ and define the lattice as $\mathcal{L}_m = \{\boldsymbol{v} \in \mathbb{Z}^{k+m+1} : \boldsymbol{B}\boldsymbol{v} = 0 \mod q\}$, where $\boldsymbol{B} = (\boldsymbol{A}_{[m]} \mid \boldsymbol{I}_m \mid \boldsymbol{b}_{[m]}) \in \mathbb{Z}_q^{m \times k+m+1}$, where $\boldsymbol{A}_{[m]}$ is the uppermost $m \times k$ sub-matrix of $\boldsymbol{A}$, $\boldsymbol{I}_m$ the identity matrix of dimension m and $\boldsymbol{b}_{[m]}$ is the uppermost length m sub-vector of $\boldsymbol{b}$ for $m \leq k$. The BKZ (Blockwise Korkine-Zolotarev) algorithm can be used with this setup to try and find a short vector for the lattice $\mathcal{L}$. This algorithm is a blockwise generalization of the LLL (Lenstra-Lenstra-Lovász) algorithm, which calculates a short and nearly orthogonal lattice basis with the Gram-Schmidt process in polynomial time fulfilling certain conditions.

The LLL algorithm begins with a basis $\boldsymbol{B}$. The first step applies the Gram-Schmidt process to produce an orthogonal basis $\boldsymbol{B'}$. In the second step, it reduces each basis vector with respect to the preceding ones and applies transformations to satisfy the LLL condition. The LLL condition ensures that each vector in the basis is not much longer than the previous vector and, if it is, reduces the longer vector. The third step checks the Lovász condition, which checks that each basis vector is not much shorter than the previous vector and, if it is, swaps the vectors. The algorithm repeats the second and third steps until no more swaps are needed, resulting in the reduced basis $\boldsymbol{B'}$.

The BKZ algorithm does essentially the same thing, the main difference being the dimension of the basis $\boldsymbol{B}$. Instead of considering the whole basis, it considers a reduced view of $\boldsymbol{B}$ of size β. It then runs the LLL algorithm a number of times, increasing the block size β each time. The algorithm ends when the termination criteria is met, which can be that a reduced lattice of a desired quality is found or β has reached the size of the original basis $\mathcal{B}$.

Tables 3 and 4 show what are the costs of this attack in each security assumption. Interestingly, it can be seen that, in general, the attack costs are higher for SMAUG in the classical setting, but higher for Kyber in the quantum setting. Therefore, for this types of attacks it would seem that Kyber is stronger since the main interest is the quantum setting. Note that the quantum attack cost for SMAUG-512 is the highest in the quantum setting and considerably larger than SMAUG-256. The reason is probably due to SMAUG having very aggressive parameters at this level compared to Kyber. For example, SMAUG has matrices of dimension 5, while Kyber's are of size 4.

Parameter sets	Classical core-SVP	Quantum core-SVP
SMAUG-128	120.0	105.6
SMAUG-192	182.8	160.9
SMAUG-256	300.5	264.5
Kyber-512	118	107
Kyber-768	182	165
Kyber-1024	256	232

Table 3: Core-SVP security of SMAUG and Kyber in primal (BKZ) attacks for MLWE

Parameter sets	Classical core-SVP	Quantum core-SVP
SMAUG-128	120.0	105.6
SMAUG-192	188.9	160.9
SMAUG-256	322.7	264.5

Table 4: Core-SVP security of SMAUG in primal (BKZ) attacks for MLWR

6.3.2 Primal Attack - BDD variant

Given a lattice $\mathcal{L}$ with basis $\boldsymbol{B}$, a vector $\boldsymbol{t}$, and a parameter $\alpha > 0$ such that the Euclidean distance between $\boldsymbol{t}$ and the lattice (or in other words, the shortest distance between $\boldsymbol{t}$ and any other vector in the lattice) is

$$d(\boldsymbol{t}, \boldsymbol{B}) < \alpha \cdot \lambda_1(\mathcal{L}(\boldsymbol{B})),$$

where λ_1 is the norm of a shortest vector in $\mathcal{L}$, find the lattice vector $\boldsymbol{v} \in \mathcal{L}(\boldsymbol{B})$ which is closest to $\boldsymbol{t}$. Therefore, instead of aiming to find the shortest vector, the BDD (Bounded Distance Decoding) variant aims to find a short vector within a bounded distance from the origin. The bound is carefully selected such that both s and e are within the search radius. For a unique solution to be guaranteed, α should be smaller than 0.5. However, when $\alpha \in [0.5, 1)$ a unique solution is expected with high probability [2].

Tables 5 and 6 show what are the costs of this attack in each security assumption. No comparison can be made with Kyber since they did not calculate the costs of this attack. However, from these results it can be seen why they may have chosen to do so. For SMAUG, the attack costs in every instance

are higher than in the BKZ variant, therefore the BDD variant attack would never be preferable over the BKZ variant attack.

Parameter sets	Classical core-SVP	Quantum core-SVP
SMAUG-128	120.9	106.5
SMAUG-192	184.4	162.4
SMAUG-256	302.4	265.9

Table 5: Core-SVP security of SMAUG in primal (BDD) attacks for MLWE

Parameter sets	Classical core-SVP	Quantum core-SVP
SMAUG-128	121.5	107.0
SMAUG-192	191.9	168.9
SMAUG-256	329.5	290.0

Table 6: Core-SVP security of SMAUG in primal (BDD) attacks for MLWR

6.3.3 Dual Attack

Consider an LWE instance $(\boldsymbol{A}, \boldsymbol{b}) \in \mathbb{Z}_q^{k \times k} \times \mathbb{Z}_q^k$ and define the lattice as $\mathcal{L'}_m = \{(\boldsymbol{u}, \boldsymbol{v}) \in \mathbb{Z}^m \times \mathbb{Z}^k : \boldsymbol{A}_{[m]}^\intercal \boldsymbol{u} + \boldsymbol{v} = 0 \mod q\}$, where $\boldsymbol{A}_{[m]}$ is the uppermost $m \times k$ sub-matrix of $\boldsymbol{A}$ for $m \leq k$. Then, the aim is to find the shortest vector in this lattice, and transform it back to the original lattice. The BKZ algorithm can be used again with this setup to try and find a short vector for the lattice $\mathcal{L}'$.

While dual attacks are considered in both SMAUG and Kyber, there are results only for SMAUG. Kyber decided to omit their results since they found that the cost of dual attacks is much larger than primal attacks, therefore not worth mentioning. Tables 7 and 8 show the attack costs of the dual attack.

Parameter sets	Classical core-SVP	Quantum core-SVP
SMAUG-128	125.9	110.8
SMAUG-192	190.4	167.6
SMAUG-256	311.0	273.7

Table 7: Core-SVP security of SMAUG in dual attacks for MLWE

Parameter sets	Classical core-SVP	Quantum core-SVP
SMAUG-128	125.9	110.8
SMAUG-192	197.1	173.5
SMAUG-256	334.9	294.8

Table 8: Core-SVP security of SMAUG in dual attacks for MLWR

6.3.4 Core-SVP Security

Table 9 shows what the core-SVP security for each scheme is. These represent the lowest attack cost of any attack on the SVP, hence a lower number means that it is less difficult to break. From the results in the table, it can be seen that Kyber is slightly more robust than SMAUG for security levels

1 and 3. In level 5, SMAUG has a higher attack cost, but once again this is probably due to SMAUG having very aggressive parameters at this level.

Parameter sets	Classical core-SVP	Quantum core-SVP
SMAUG-128	120.0	105.6
SMAUG-192	181.7	160.9
SMAUG-256	264.5	245.2
Kyber-512	118	107
Kyber-768	182	165
Kyber-1024	256	232

Table 9: Core-SVP security of SMAUG and Kyber

It should be noted that some of the values of this table do not show up in the previous tables. The values 181.7, 264.5 and 245.2 do not show up in the other tables since they correspond to attack costs of hybrid dual attacks. A hybrid dual attack is an attack that uses a dual attack together with another type of attack, such as a combinatorial or algebraic attack. These need a closer inspection which is not done in this report due to time restraints.

6.4 Beyond Core-SVP Methodology

There are more attacks that target weaknesses in systems that do not necessarily target the SVP, combinatorial attacks are also a threat for systems with certain characteristics, like SMAUG with its sparse secret. There are no known practical algorithms that can take advantage of the sparse-LWE structure of SMAUG. However, work is progressing in the field and more efficient algorithms are being developed. The simplest algorithm to break sparse-LWE is brute force which is executed in the following way:

Algorithm 20: Brute-Force s-LWE

Input: $\boldsymbol{A} \in \mathbb{Z}_q^{n \times n}, \boldsymbol{b} \in \mathbb{Z}_q^n$
For all $\boldsymbol{s} \in \{0, \pm 1\}^n$
  If $\boldsymbol{As} - \boldsymbol{b} \in \{0, \pm 1\}^n$
    Return: $\boldsymbol{s}$
  end if
end for

Note that this only returns potential candidates for $\boldsymbol{s}$ and not a unique solution, unless it only returns a single candidate. This works if $\boldsymbol{e}$ is in $\{0, \pm 1\}^n$, which, even though it is not the case for SMAUG or Kyber, it can be adapted to the range that $\boldsymbol{e}$ takes. The search space is $\mathcal{S} = 3^n$ since it tries all possible combinations of arranging 3 different numbers in n spots.

A refinement can be accomplished using the Meet-in-the-Middle technique, which splits $\boldsymbol{s}$ in two to probe smaller search spaces. Instead of using the traditional $\boldsymbol{As} + \boldsymbol{e} = \boldsymbol{b}$ equation, this method breaks up $\boldsymbol{A}$ and $\boldsymbol{s}$ to give the following equation:

$$\boldsymbol{A}_1 \boldsymbol{s}_1 = -\boldsymbol{A}_2 \boldsymbol{s}_2 + \boldsymbol{b} - \boldsymbol{e}$$

which can be also approximated as

$$\boldsymbol{A}_1 \boldsymbol{s}_1 \approx -\boldsymbol{A}_2 \boldsymbol{s}_2 + \boldsymbol{b}.$$

Then Algorithm 21 can be used to improve efficiency when searching for $\boldsymbol{s}$.

Algorithm 21: Meet-in-the-Middle

Input: $\boldsymbol{A} = (\boldsymbol{A}_1 | \boldsymbol{A}_2) \in \mathbb{Z}_q^{n \times n}, \boldsymbol{b} \in \mathbb{Z}_q^n$
For all $\boldsymbol{s}_1 \in \{0, \pm 1\}^{\frac{n}{2}}$
    Construct L_1 with entries $(\boldsymbol{s}_1, h(\boldsymbol{A}_1 \boldsymbol{s}_1))$
end for
For all $\boldsymbol{s}_2 \in \{0, \pm 1\}^{\frac{n}{2}}$
    Construct L_2 with entries $(\boldsymbol{s}_2, h(-\boldsymbol{A}_2 \boldsymbol{s}_2 + \boldsymbol{b}))$
end for
Return: $(\boldsymbol{s}_1 \| \boldsymbol{s}_2)$ with $h(\boldsymbol{A}_1 \boldsymbol{s}_1) = h(-\boldsymbol{A}_2 \boldsymbol{s}_2 + \boldsymbol{b})$

The reason this works is because h is a Locality-Sensitive Hash (LSH) function. This means that inputs which are, in this case, numerically close produce the same output with high probability, which allows for the flexibility of not considering $\boldsymbol{e}$. Moreover, because of the split of $\boldsymbol{s}$, the search space is reduced to half of its original size which means that the running time is $\mathcal{S}^{\frac{1}{2}} = 3^{\frac{n}{2}}$. In reality, the actual time complexity (ignoring linear and polynomial terms) is $2 \cdot 3^{\frac{n}{2}}$, because the reduced search space is explored twice. However, as n becomes larger, this scalar factor is of very little significance.

Further refinements were suggested by Alexander May [20], and can be implemented by changing the representation of vectors using the REP-$0, 1, 2$ representations. REP-0 represents 1 as $1 = 1+0 = 0+1$ and -1 as $-1 = (-1) + 0 = 0 + (-1)$, REP-1 represents 0 as $0 = 1 + (-1) = (-1) + 1$ and REP-2 uses the number 2 to rewrite some numbers, such as in the following examples:

$$\begin{aligned}
&\text{REP-}0 \rightarrow (1, 0, -1, 1, -1) = (1, 0, -1, 0, 0) + (0, 0, 0, 1, -1) \\
&\text{REP-}1 \rightarrow (1, 0, -1, 1, -1) = (1, 1, -1, 0, 0) + (0, -1, 0, 1, -1) \\
&\text{REP-}2 \rightarrow (1, 0, -1, 1, -1) = (2, 1, -1, 0, 0) + (-1, -1, 0, 1, -1).
\end{aligned}$$

Using these representations, Algorithm 22 illustrates how the Meet-LWE algorithm works. The idea is similar to the Meet-in-the-Middle algorithm, however instead of approximating the whole equation, it tries to find an exact solution by guessing r coordinates of $\boldsymbol{e}$ and an approximation for the remaining $n - r$ coordinates.

Algorithm 22: Meet-LWE

Input: $\boldsymbol{A} \in \mathbb{Z}_q^{n \times n}, \boldsymbol{b} \in \mathbb{Z}_q^n$
Choose representation REP-$0, 1, 2$
Guess r coordinates of $\boldsymbol{e}$, denoted $\boldsymbol{e}_r$.
    For all $\boldsymbol{s}_1$: Construct L_1 with entries $(\boldsymbol{s}_1, h(\boldsymbol{A}\boldsymbol{s}_1))$
    For all $\boldsymbol{s}_2$: Construct L_2 with entries $(\boldsymbol{s}_2, h(-\boldsymbol{A}\boldsymbol{s}_2 + \boldsymbol{b}))$
    Return: $\boldsymbol{s}_1 + \boldsymbol{s}_2$ if:
        $\boldsymbol{A}\boldsymbol{s}_1 = -\boldsymbol{A}\boldsymbol{s}_2 + \boldsymbol{b} + \boldsymbol{e}_r$ on r coordinates
        $h(\boldsymbol{A}\boldsymbol{s}_1) = h(-\boldsymbol{A}\boldsymbol{s}_2 + \boldsymbol{b})$ on $n - r$ coordinates

As the number of representations of the secret vector $\boldsymbol{s}$ increases, the trade-off between the number of guessed coordinates r and the size of the complement set $\mathcal{S}^c$ becomes more pronounced. Opting for a larger value of r in this trade-off proves beneficial, as key guessing exhibits slightly sub-exponential complexity, making it more feasible compared to the fully exponential growth of the set $\mathcal{S}$ in terms of the vector dimension n [20]. May's paper [20] determines that this brings down an attack complexity from $\mathcal{S}^{\frac{1}{2}}$ to $\mathcal{S}^{\frac{1}{4}}$.

In SMAUG-128, the sparse secret key s has two polynomials where 140 coefficients out of 512 are nonzero, and the nonzero coefficients can take either 1 or -1 as a value. Therefore, the amount of different ways those values can happen in 140 coefficients are 2^{140}. Moreover, the amount of different ways nonzero coefficients can be arranged in 512 coefficients is $\binom{512}{140}$. Therefore, the total amount of different possibilities for s are

$$\mathcal{S} = 2^{140} \cdot \binom{512}{140} \approx 2^{568}.$$

Even if $\mathcal{S}^{0.25}$ is assumed, that means that $\mathcal{S}^{0.25} \approx 2^{142}$. Fugaku, which is one of the most potent supercomputers can do 442 petaFLOPS, which if each FLOP corresponds to 1 secret key try then it tries roughly 2^{58} per second, this means that the time spent to break this is 4 million times longer that the current lifespan of the universe. Therefore, it can be assumed that, with the developments at the time of writing of this report, this is not feasible, and therefore using a sparse secret is secure as SMAUG claims.

7 Conclusion

In conclusion, this report has provided a detailed exploration of the current state of post-quantum cryptography - analysing both the winner of the NIST competition, as well as SMAUG, a challenger to the throne. The report has covered a wide range of topics, including the basics of encryption and decryption, the differences between symmetric key encryption and public key encryption, and the advantages of the Learning with Errors/Rounding technique in cryptography. Additionally, the paper has provided a comprehensive comparison of two cryptographic systems, SMAUG and Kyber, highlighting their strengths and weaknesses. The report has also explored the security of each scheme and given some of the most common attacks. The main findings of this report are summarized in the following paragraphs.

Firstly, instead of relying on the security assumption MLWE for the encryption system like Kyber, SMAUG relies on MLWR. SMAUG claims that using MLWR for encryption allows for faster operations than MLWE. For both MLWE and MLWR, a vector $\boldsymbol{r}$ is created to be multiplied by $\boldsymbol{A}$ and used in the inner product with $\boldsymbol{b}$. However, in MLWE an additional vector $\boldsymbol{e}_1$ and polynomial e_2 have to be created to use in the encryption and then added. On the other hand, MLWR only needs to do one scalar multiplication and rounding operation. Therefore, SMAUG should be faster in encryption, however this is not the case because of the different approaches in polynomial multiplication, rather than because of the nature of MLWE and MLWR themselves.

Secondly, SMAUG chooses to use sparse secrets in both its key generation and encryption systems, while Kyber has a more conventional key. This means that SMAUG's secret keys take less storage than Kyber's, at the cost of reducing the key space for the secret key. The sparse structure also allows SMAUG to perform polynomial multiplication in a faster manner than regular multiplication by leveraging the amount of nonzero coefficients present in each sparse polynomial. Specific attacks on the sparse secret, such as MEET-LWE, are still not efficient enough to be a concern practically, but further study is needed to ensure the long-term security of it.

Lastly, the modulo value with which most operations happened was q, and both schemes chose distinct values with different rationale. Kyber's $q = 3329$ is larger than SMAUG's $q = 1024$, which means Kyber needs two additional bits to represent numbers. However, Kyber uses a compress function that discards lower order bits, meaning that both schemes store the same amount of bits. However, because of how the q's are chosen, Kyber can benefit from NTT while SMAUG cannot. SMAUG implements their own polynomial multiplication algorithm, nonetheless NTT is considerably faster than it. Therefore, Kyber has an advantage in running time when it comes to multiplication.

Overall, the report accomplished the goal set in the introduction. A comprehensive understanding of post-quantum cryptography, with a focus on lattice-based problems was effectively provided. However, more work can still be done which is outlined in the next section.

8 Future Work

Even though the report successfully gave an overview of MLWE/R-based post-quantum cryptography, more work can be done in regards to comparing SMAUG and Kyber. As it was demonstrated in Section 5.5, the polynomial multiplication in Kyber is much faster than in SMAUG. However, SMAUG claims that it runs 100% faster than Kyber, so this claim requires further analysis. The code implementation should be studied, since perhaps the efficiency comes from there, given how the modular arithmetic was done in powers of 2 and its relationship to bit shifting. The speed increase should also be noticed in the MLWR operations compared to the MLWE ones in the encryption system. Other areas where SMAUG is faster is in their KEM, where it gains some efficiency by not computing the shared key with the ciphertext.

More attacks could be studied, and more in-depth explanations of how those attacks work. Currently, only Meet-LWE has an in-depth explanation, but the BKZ algorithm could be further explained. It was not done in this report due to its complexity and the time constraint on this project. The hybrid dual attacks could be explored further too, since they gave the lowest attack costs for some variants of SMAUG. Additionally, the sampling algorithms could be studied more to ensure that they do not introduce any security issues.

Finally, more research can be done into the errors affecting the decryption failure probability. In Section 6.2.1 and 6.2.2, the general error term was given for each scheme, however each one was defined differently and had different characteristics. Therefore, calculations could be run to estimate how much the general error terms perturb the original message.

References

[1] Martin R Albrecht and Amit Deo. *Large Modulus Ring-LWE ≥ Module-LWE*. 2017. URL: https://ia.cr/2017/612.

[2] Martin R Albrecht and Nadia Heninger. *On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem*. 2021, p. 5. URL: https://ia.cr/2020/1540.

[3] Erdem Alkim et al. *FrodoKEM Learning With Errors Key Encapsulation Algorithm Specifications And Supporting Documentation*. June 2021, p. 44. URL: https://frodokem.org/.

[4] Erdem Alkim et al. *Post-quantum key exchange-a new hope*. 2019, p. 8. URL: https://ia.cr/2015/1092.

[5] Roberto Avanzi et al. *CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 3.0)*. 2020. URL: https://pq-crystals.org/kyber/resources.shtml.

[6] Andrea Basso et al. *SABER: Mod-LWR based KEM (Round 3 Submission)*. 2020. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[7] Daniel Bernstein. *Combinatorial lattice security?* 2022. URL: https://groups.google.com/g/kpqc-bulletin/c/GejJ_lp3GLI.

[8] Andrej Bogdanov et al. *On the Hardness of Learning with Rounding over Small Modulus*. Cryptology ePrint Archive, Paper 2015/769. 2015. URL: https://eprint.iacr.org/2015/769.

[9] Andrej Bogdanov et al. *On the Hardness of Learning with Rounding over Small Modulus*. Cryptology ePrint Archive, Paper 2015/769. 2015. URL: https://eprint.iacr.org/2015/769.

[10] Jung Hee Cheon et al. *SMAUG: Pushing Lattice-based Key Encapsulation Mechanisms to the Limits*. May 2023. URL: https://ia.cr/2023/739.

[11] Whitfield Diffie and Martin E Hellman. *New Directions in Cryptography*. 1976. URL: https://www-ee.stanford.edu/~hellman/publications/24.pdf.

[12] Léo Ducas et al. *CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme*. 2018. URL: https://ia.cr/2017/633.

[13] Morris J. Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. National Institute of Standards and Technology, July 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[14] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34. URL: https://link.springer.com/content/pdf/10.1007%2F3-540-48405-1_34.pdf.

[15] Steve Fyffe and Tom Abate. *Stanford cryptography pioneers win 2015 Turing Award*. 2016. URL: https://news.stanford.edu/2016/03/01/turing-hellman-diffie-030116/.

[16] Ruben Gonzalez. *Kyber - How does it work?* 2021. URL: https://cryptopedia.dev/posts/kyber/.

[17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. *A Modular Analysis of the Fujisaki-Okamoto Transformation*. 2021. URL: https://ia.cr/2017/604.

[18] Adeline Langlois and Damien Stehle. *Worst-Case to Average-Case Reductions for Module Lattices*. Cryptology ePrint Archive, Paper 2012/090. 2012. URL: https://eprint.iacr.org/2012/090.

[19] Zhichuang Liang and Yunlei Zhao. "Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey". In: (Nov. 2022), pp. 8–11. URL: http://arxiv.org/abs/2211.13546.

[20] Alexander May. *How to Meet Ternary LWE Keys*. 2021. URL: https://ia.cr/2021/216.

[21] Krzysztof Pietrzak. "Subspace LWE". In: *Theory of Cryptography*. Vol. 7194. Lecture Notes in Computer Science. Springer, 2012, pp. 548–563. DOI: 10.1007/978-3-642-28914-9_31. URL: https://www.iacr.org/archive/tcc2012/71940166/71940166.pdf.

[22] Center for Quantum Resistant Cryptography. *Home Page*. 2022. URL: https://www.kpqc.or.kr/.

[23] Oded Regev. *On Lattices, Learning with Errors, Random Linear Codes, and Cryptography*. 2009. URL: https://cims.nyu.edu/~regev/papers/qcrypto.pdf.

[24] National Institute of Standards and Technology. *Post-Quantum Cryptography*. 2022. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/.

Own Contributions

The original report had quite a few typos that made understanding some sections challenging. Therefore, the SMAUG authors were emailed a couple of times for clarifications on those matters. In this section, the typos that were corrected in the report and the communication with the SMAUG authors are provided.

Firstly, in line 2 of algorithm 6, XOF is used to initialise the seeds in the algorithm. However, in the actual paper H is used which is defined as SHA3-256 in page 20 Section 6.1 in [10]. This function outputs a fixed length 256-bit output, and since $\text{seed}_{\boldsymbol{A}}$ is 256-bits it means that either seed_{sk} is not populated properly, or they are both populated with the same value which would be incredibly dangerous. Since $\text{seed}_{\boldsymbol{A}}$ is made public, it would effectively make seed_{sk} public, allowing anyone to reconstruct the secret key.

Secondly, the polynomial multiplication algorithm 1 has a few typos in the original paper too. The most important one is that the variables x and y were also named a, therefore it had only one variable name when 3 were being used. Algorithm 23 below shows the original algorithm, with the changed sections highlighted in red.

Algorithm 23: poly_mult_add: Original

Input: $a \in \mathcal{R}_q, b \in \mathcal{S}_\eta$
For i from 0 to neg_start-1 **do**
    degree $= b[i]$
    For j from 0 to n **do**
        $a[\text{degree} + j]$ += $a[j]$
    end for
end for
For i from neg_start to len(b) **do**
    degree $= b[i]$
    For j from 0 to n **do**
        $a[\text{degree} + j]$ −= $a[j]$
    end for
end for
For i from 0 to n **do**
    $a[j] = a[j] - a[n + j]$
end for
Return: a

Below the original messages exchanged with the SMAUG authors can be seen.

Jorge Correa Merlino <jorgecorreamerlino@gmail.com>

Re: Mensaje privado sobre: [SMAUG update!] SMAUG version update v1.0

4 messages

Hyeongmin.Choe <sixtail528@snu.ac.kr> 11 July 2023 at 14:45
To: Jorge Correa Merlino <jorgecorreamerlino@gmail.com>

Dear Jorge Corres Merlino,

Thank you for your interest in SMAUG.

> In Algorithm 1 (keygen) Line 2, you instantiate seed_A and seed_sk with the hash function H. Then in section 4.2, you instantiate H as SHA3-256 which generates a fixed output of 256 bits. However, in Algorithm 7 (expandA), the seed that is fed into it is a 256 bit seed. In Algorithm 1 Line 3, you call expandA with seed_A therefore seed_A should be of 256 bits. But then how is seed_sk instantiated? Since it would look like all the output bits from H are going to seed_A.

We are sorry for the confusion. It should be XOF as in the previous version. Specifically, we used SHAKE128.

> Additionally, by Algorithm 1 Line 4, where HWT is called with seed_sk, and Algorithm 8 (HWT) we see that seed_sk is also a 256 bit seed. But then in Algorithm 1 Line 5, you pass seed_sk as a parameter, when in Algorithms 9 and 10 (dGaussian) the input looks like it is a 10 bit number. Do you take just the first 10 bits as input? Moreover, Algorithms 9 and 10 return s which looks like it is a 2 or 3 bit number, but in Algorithm 1 line 5, it says that e is an element of the polynomial ring. Does this mean that the output is always just an integer?

dGaussian is an *integer sampler* and also implies a *polynomial sampler* (with a natural extension).
The latter uses the integer sampler to sample each coefficient.

Specifically, in Algorithm 1, seed_sk is fed into the polynomial sampler dGaussian. Then the seed is extended to have 10*n*k bits using XOF (11*n*k bits in the case of the level-3 parameter set).
Then we slice the extended seed into n*k number of *small seeds* (each having 10 bits or 11 bits for level 3).
Each small seed will be fed into the integer sampler dGaussian and the integer sampler will output one coefficient from each small seed. Then the polynomial sampler finally outputs a polynomial having those coefficients.

Best,
Hyeongmin Choe
--

Hyeongmin Choe
Department of Mathematical Sciences,

Seoul National University, Republic of Korea

Office: Bldg 27, Rm 441
Phone: +82-2-880-6272

E-mail: sixtail528@snu.ac.kr

Website: hmchoe0528.github.io

Jorge Correa Merlino <jorgecorreamerlino@gmail.com> 15 July 2023 at 19:57
To: "Hyeongmin.Choe" <sixtail528@snu.ac.kr>

Dear Hyeongmin Choe,

Thank you very much for your explanation, everything is now clear to me. I would also like to ask you about the poly_mult_add function if that would be alright.

After inspecting your code, I believe that in lines 3, 7, and 9 instead of n they should be n-1. Additionally, I think line 5 should be len(b)-1 since for all these operations you use a less than symbol.
Also the variable "a" is used through the algorithm while it takes the form of three different variables. If we name these "x", "y" and "a" then the following lines would be like this:

Line 4: x[degree + j] = x[degree + j] + a[j]
Line 8: x[degree + j] = x[degree + j] - a[j]
Line 10: y[j] = x[j] - x[n+j]
Line 11: return y

Is this interpretation correct? If it is not, I do not understand how the algorithm works and would kindly request if you could explain it to me. Thank you for your time.

Kind regards,
Jorge Correa Merlino
[Quoted text hidden]

Hyeongmin.Choe <sixtail528@snu.ac.kr> 19 July 2023 at 14:23
To: Jorge Correa Merlino <jorgecorreamerlino@gmail.com>

Hi Jorge,

Sorry for the late reply. Yes, you are correct.
Thanks again for pointing out the typos we made.
FYI: within a few weeks, the updated paper will be uploaded to eprint.

Best,
Hyeongmin

2023년 7월 16일 (일) 오전 3:58, Jorge Correa Merlino <jorgecorreamerlino@gmail.com>님이 작성:
[Quoted text hidden]
[Quoted text hidden]

Jorge Correa Merlino <jorgecorreamerlino@gmail.com> 19 July 2023 at 18:52
To: "Hyeongmin.Choe" <sixtail528@snu.ac.kr>

Great! I look forward to reading it

Kind regards,
Jorge Correa Merlino
[Quoted text hidden]