School of Computer Science,
University of Birmingham

Final year project

Analysis and Implementation of Post-Quantum Cryptosystems

Andrew Bryer - 1523503
BSc Mathematics and Computer Science

supervised by
David Galindo

March 30, 2018

Contents

1 Abstract

In the light of recent advances in quantum computing, the National Institute of Standards and Technology (NIST) initiated a process to solicit, evaluate and standardise one or more quantum-resistant public-key cryptographic algorithms (*Post-Quantum Cryptography*, 2017). In this project, I implemented two such algorithms: Kyber, 'CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM' (Bos, J. *et al.*, 2017); and LIMA, 'Tightly Secure Ring-LWE Based Key Encapsulation with Short Ciphertexts' (Albrecht, M.R. *et al.*, 2017), whose securities are based on the Module Learning with Errors (Module-LWE) and Ring Learning with Errors (Ring-LWE) problems respectively. Using these implementations I compared the efficiency of the two schemes for a range of security levels using given parameters for Kyber and by finding suitable parameters for LIMA.

2 Acknowledgements

I would like to express my gratitude to my project supervisor, David Galindo, for help with suggesting the schemes to implement and for being available for questions throughout the implementation and analysis process.

3 Introduction

Since the theorisation and recent realisation of quantum computers, the limitations of current cryptosystems and the mathematical problems on which they are based have become clear and also that work needs to be done to find replacement problems on which to base quantum-resistant schemes. This has been emphasised by the launch of a Post-Quantum Cryptography project by the US standards body, NIST, in early 2017 (*Post-Quantum Cryptography*, 2017) for submissions of quantum-resistant cryptographic algorithms. A large proportion of these submissions have security based on the family of problems: Learning with Errors (LWE); Ring Learning with Errors (Ring-LWE); and Module Learning with Errors (Module-LWE) which are seen to be leading candidates for the problems on which to base quantum-resistant cryptosystems.

My project involved studying and implementing two CCA-secure Key Encapsulation Mechanisms detailed in the papers; 'CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM' (Bos, J. *et al.*, 2017) and 'Tightly Secure Ring-LWE Based Key Encapsulation with Short Ciphertexts' (Albrecht, M.R. *et al.*, 2017). This has involved me reading the two papers in detail, understanding the underlying problems and mechanisms used to convert the CPA-secure scheme to a CCA-secure scheme and then using this knowledge to implement them in SageMath, a Python library. I then used these implementations to make comparisons between the two schemes, in particular between their efficiencies and key sizes. To do this I required parameters for different security levels. Kyber provided me with parameters for 128-bit, 192-bit and 256-bit security however I had to find my own parameters for LIMA to match these security levels.

4 Background Material

Before implementing the schemes I needed to make sure I had a full knowledge of the underlying theories behind them. Although I already had some knowledge of the mathematics behind Ring-LWE from an internship in 2017, I decided to start from the beginning with my research to ensure I did not have any misconceptions.

4.1 Learning with Errors (LWE)

Learning with Errors (LWE) is a mathematical problem on which many proposed quantum-resistant cryptoschemes are based. It is conjectured to be hard to solve since an efficient solution implies a quantum solution to GapSVP and SIVP (Regev, O., 2009). The problem can be expressed either as a search problem or a decision problem, both begin with the same construction. Given dimension n, modulus q and an error distribution χ:

- Let $\mathbf{s} \in \mathbb{Z}_q^n$ be fixed
- Let the $\mathbf{a_i}$'s $\in \mathbb{Z}_q^n$ be sampled independently from a uniform distribution over $\mathbb{Z}_q^n$
- Let the e_i's $\in \mathbb{Z}_q$ be sampled independently from χ
- Let the b_i's $\in \mathbb{Z}_q$ be such that $b_i = \langle \mathbf{a_i}, \mathbf{s} \rangle + e_i$

The decision problem is to determine whether the b_i's were sampled uniformly at random from $\mathbb{Z}_q$ or as above. The search problem is to find $\mathbf{s}$ given the list of pairs $(\mathbf{a_i}, b_i)$.

4.1.1 Ring-LWE

Ring-LWE is an extension of the LWE problem and is alternatively referred to as Learning with Errors over Rings. The main difference is that the n-dimensional vectors are replaced by polynomials with degree smaller than n (Albrecht, M.R. and Deo, A., 2017). In order to do this we consider polynomials with integer coefficients taken module q that are members of the polynomial ring with quotient $\Phi(x)$. Usually this irreducible polynomial is the $(n+1)^{th}$ cyclotomic polynomial where $n+1$ is prime or a power of 2. Similar to above, the problem is constructed as follows. Given dimension, n, modulus q, an error distribution χ and an irreducible polynomial $\Phi(x)$:

- Let $s \in \mathbb{Z}_q[x]/\Phi(x)$ have coefficients sampled independently from χ
- Let the a_i's $\in \mathbb{Z}_q[x]/\Phi(x)$ be sampled independently from a uniform distribution over $\mathbb{Z}_q[x]/\Phi(x)$
- Let the e_i's $\in \mathbb{Z}_q[x]/\Phi(x)$ have coefficients sampled independently from χ

- Let the b_i's $\in \mathbb{Z}_q[x]/\Phi(x)$ be such that $b_i = a_i \cdot s + e_i$

The search and decision problems are the same as for LWE. It has been proven that Ring-LWE is at least as hard as LWE on ideal lattices (Lyubashevsky, V., Peikert, C. and Regev, O. (2013); Peikert, C., Regev, O. and Stephens-Davidowitz, N., 2017).

4.1.2 Module-LWE

The Module-LWE problem can be viewed as an extension of Ring-LWE where the ring elements, a_i's, s and e_i's are replaced with module elements over the same ring, that is, the a_i's are sampled from $(\mathbb{Z}_q[x]/\Phi(x))^{k\times k}$ and s and the e_i's are sampled from a distribution χ over $(\mathbb{Z}_q[x]/\Phi(x))^k$. In this regard, Ring-LWE can be viewed as Module-LWE with module rank 1 (Albrecht, M.R. and Deo, A., 2017).

4.2 CPA and CCA Security

One key aspect of a cryptographic algorithm that needs to be considered is its security, in particular whether it is CPA-secure (Chosen Plaintext Attack) or CCA-secure (Chosen Ciphertext Attack). CCA-security is considered the strongest model of security for a cryptosystem (Tomescu, A., 2011). Both of my chosen cryptosystems have a CCA-secure system, based on an initial CPA-secure system.

4.2.1 CPA Security

The IND-CPA game is as follows. Suppose you have a challenger with a cryptosystem and an adversary:

1. The challenger generates a key pair (pk, sk) based on some security parameter and publishes pk to the adversary but keeps sk secret.
2. The adversary may now carry out any number of encryptions.
3. Eventually the adversary submits two distinct chosen plaintexts m_0 and m_1 to the challenger.
4. The challenger selects a bit $b = \{0, 1\}$ uniformly at random and sends the challenge ciphertext $c = \text{Enc}(pk, m_b)$ back to the adversary.
5. The adversary can then perform any other operations with the aim to determine the value of b.

A cryptosystem is CPA secure if any adversary has only a negligible advantage over random guessing. It is clear from this that any CPA-secure scheme is non-deterministic since otherwise the adversary could already have the encryption of m_0 or m_1 from step 2.

4.2.2 CCA Security

The IND-CCA game is set out similar to the IND-CPA game. However, in steps 2 and 5, the adversary also has access to decryption but is not allowed to query the decryption of the challenge ciphertext. Again, a cryptosystem is CCA-secure if any adversary has only a negligible advantage over random guessing.

4.3 Key Encapsulation Mechanisms

One issue with asymmetric cryptosystems is that often the length of a message is limited. This is a problem should one want to transfer the key of a symmetric system that is longer than this limit. A Key Encapsulation Mechanism (KEM) is a system that allows for the transfer of such keys. Instead of encrypting a message, m, one generates a random list of bits, l, and encrypts l, producing a cipher, c, while also passing l into a predetermined Key Derivation Function (KDF), that extends l to a key, k, of the required length. The cipher, c, can be decrypted and passed into the same KDF to recover the same key at the other end. It is said that the cipher, c, encapsulates the key, k.

4.4 Hash Functions

Hash functions are functions that map data of an arbitrary size to data of a fixed size. In cryptography, cryptographic hash functions are particularly useful. These are hash functions designed to be one way, that is that they are infeasible to invert. This makes them particularly useful because an adversary cannot feasibly recover the original data from the hashed data. Both schemes make use of hash functions in multiple places.

5 Implementation

Once I had researched all the background information needed to understand the two schemes I was able to begin coding the implementations. I chose to implement both schemes in SageMath, a Python library, as it adds some useful mathematical structures, in particular polynomials, matrices and vectors. Although this implementation will not be as efficient as one in a language closer to the machine level, such as C, it will allow me to implement the two schemes more easily and make relative comparisons. During implementation I had to ensure I was correctly implementing the algorithms in each of the papers as any mistake would render any results useless due to it being a different scheme. For each algorithm I will include a copy of the pseudo code from the respective paper and any design choices I made during implementation.

5.1 Kyber

5.1.1 Helper Functions

Kyber made use of some helper functions to compress the coefficients of the polynomials in the public key and cipher, and also used a binomial distribution rather than a Gaussian distribution which required me to implement the binomial sampler detailed in the paper.

5.1.1.1 Compress$_q$

Compression needed to be an iterative process in which every polynomial element of the input matrix has its coefficients compressed. This caused some problems during implementation as my polynomials were members of the polynomial ring with coefficients taken modulo q, and quotient $\Phi(x)$. Therefore in order to multiply by $2^d/q \in \mathbb{R}$ I had to deal with each coefficient separately before recombining the coefficients to form the resulting polynomial.

Algorithm 1 Compress$_q(x \in R_q^{m \times n}, d)$

$x' \sim R_{2^d}^{m \times n}$
for $i \leftarrow 1, m$ **do**
    for $j \leftarrow 1, n$ **do**
        $x'_{i,j} = \lceil (2^d/q) \cdot x_{i,j} \rfloor \bmod 2^d$
    end for
end for
return x'

5.1.1.2 Decompress$_q$

Decompression is a very similar process to compression. It also needed to iterate over the matrix to ensure every polynomial element had its coefficients decom-

pressed which led to the same issues regarding the polynomial ring of which a polynomial was a member. I solved this in the same way as in compression.

Algorithm 2 Decompress$_q(x \in R_{2^d}^{m \times n}, d)$

1: $x' \sim R_q^{m \times n}$
2: **for** $i \leftarrow 1, m$ **do**
3: **for** $j \leftarrow 1, n$ **do**
4: $x'_{i,j} = \lceil (q/2^d) \cdot x_{i,j} \rfloor$
5: **end for**
6: **end for**
7: **return** x'

5.1.1.3 Generate Binomial

This function was used to generate the coefficients of particular polynomials. It was detailed in the paper as the way to generate a random sample from a centred binomial distribution. It was very simple to implement using the in-built random number generator to generate either 0 or 1 for each a_i and b_i and then sum appropriately.

Algorithm 3 GenerateBinomial(η)

1: $(a_1, \ldots, a_\eta, b_1, \ldots, b_\eta) \leftarrow \{0,1\}^{2\eta}$
2: **return** $\sum_{i=1}^{\eta}(a_i - b_i)$

5.1.2 Key Generation

The main issue I had during the implementation of the key generation function was with the extendable output function, Sam. I was initially not sure how to go about implementing this. However I soon realised that I could treat the input to Sam as the seed for a random generator used to generate the output.

Algorithm 4 KyberKeyGen()

1: $\rho, \sigma \leftarrow \{0,1\}^{256}$
2: $\mathbf{A} \sim R_q^{k \times k} := \text{Sam}(\rho)$
3: $\mathbf{(s,e)} \sim \beta_\eta^k \times \beta_\eta^k := \text{Sam}(\sigma)$
4: $\mathbf{t} := \text{Compress}_q(\mathbf{As} + \mathbf{e}, d_t)$
5: **return** $(pk := (\mathbf{t}, \rho), sk := \mathbf{s}$

5.1.3 CPA Scheme

5.1.3.1 CPA Encryption

As compression and decompression had already been implemented, these parts of the function were easy to implement. Furthermore, this function made use of

the same extendable output function, Sam, to generate certain random values and this could once again been implemented as described in Key Generation.

Algorithm 5 KyberEncCPA($pk = (\mathbf{t}, \rho), m \in \{0,1\}^{256}$)

$r \leftarrow \{0,1\}^{256}$
$\mathbf{t} := \text{Decompress}_q(\mathbf{t}, d_t)$
$\mathbf{A} \sim R_q^{k \times k} := \text{Sam}(\rho)$
$(\mathbf{r}, \mathbf{e_1}, e_2) \sim \beta_\eta^k \times \beta_\eta^k \times \beta_\eta := \text{Sam}(r)$
$\mathbf{u} := \text{Compress}_q(\mathbf{A}^T\mathbf{r} + \mathbf{e_1}, d_u)$
$v := \text{Compress}_q(\mathbf{t}^T\mathbf{r} + e_2 + \lceil q/2 \rfloor \cdot m, d_v)$
return $c := (\mathbf{u}, v)$

5.1.3.2 CPA Decryption

CPA decryption was easy to implement since it is a simple function.

Algorithm 6 KyberDecCPA($sk = \mathbf{s}, c = (\mathbf{u}, v)$)

$\mathbf{u} := \text{Decompress}_q(\mathbf{u}, d_u)$
$v := \text{Decompress}_q(v, d_v)$
return $\text{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1)$

5.1.4 CCA KEM

5.1.4.1 CCA Encapsulation

For the encapsulation mechanism, I had to make two design choices: the choice of the hash functions, G and H; and how to input the public key into G. By looking at the C reference implementation of Kyber I decided to use SHA-3-256 for H and SHAKE-256 with output size of 768 bits for G. In an ideal world I would convert the public key to a bit string and input this into G, however, for ease of implementation in Python I chose to use only the value of ρ since it is difficult to access the bit string in a high level language.

Algorithm 7 KyberEncap($pk = (\rho, \mathbf{t})$)

$m \leftarrow \{0,1\}^{256}$
$(\hat{K}, r, d) := \text{G}(pk, m)$
$(\mathbf{u}, v) := \text{KyberEncCPA}((\rho, \mathbf{t}), m; r)$
$c := (\mathbf{u}, v, d)$
$K := \text{H}(\hat{K}, c)$
return (c, K)

5.1.4.2 CCA Decapsulation

The design choices I had made for the encapsulation function were applicable to the decapsulation function. I was able to use the same choices for the functions G, H and inputting the public key into G.

Algorithm 8 KyberDecap($sk = \mathbf{s}, pk = (\rho, \mathbf{t}), c = (\mathbf{u}, v, d)$)

$m' :=$ KyberDecCPA$(s, (\mathbf{u}, v))$
$(\hat{K}', r', d') :=$ G(pk, m')
$(\mathbf{u'}, v') :=$ KyberEncCPA$((\rho, \mathbf{t}), m'; r')$
if $(\mathbf{u'}, v', d') = (\mathbf{u}, v, d)$ **then**
    return $K :=$ H$(\hat{K}', c)$
else
    $z \leftarrow \{0,1\}^{256}$
    return $K :=$ H(z, c)
end if

5.2 LIMA

5.2.1 Key Generation

Key generation in LIMA required the sampling of multiple variables from Gaussian and Uniform distributions. I was able to use in-built random samplers to sample each of the variables from each distribution and then combine these as required to produce the secret and public keys.

Algorithm 9 KeyGen()

$a \leftarrow R_q$
$s \leftarrow \chi_\sigma^N$
$e \leftarrow \chi_\sigma^N$
$b \leftarrow a \cdot s + e$
$sk \leftarrow s$
$pk \leftarrow (a, b)$
return (pk, sk)

5.2.2 CPA Scheme

5.2.2.1 CPA Encryption

The CPA encryption function required the function, BV-2-RE, which is already implemented in SageMath as you can create a new polynomial with the elements of a list as its coefficients. I also needed to decide how to sample with random coins r and this could be solved by setting r as the seed of a random generator used to sample v, e and d. It also requires use of the Trunc function which can be implemented by truncating the list of coefficients and recreating a polynomial with this new list of coefficients.

Algorithm 10 EncCPA($m \in \{0,1\}^l, pk = (a, b), r$)

$\mu \leftarrow$ BV-2-RE(m) ▷ BV-2-RE converts m to a polynomial in R_q
$v, e, d \leftarrow_r \chi_\sigma^N$ ▷ Random coins r used as sampling seed
$x \leftarrow d + \lfloor q/2 \rfloor \cdot \mu \pmod q$
$t \leftarrow b \cdot v + x$
$c_0 \leftarrow$ Trunc(t, l) ▷ Trunc removes powers of x greater than l
$c_1 \leftarrow a \cdot v + e$
return $c = (c_0, c_1)$

5.2.2.2 CPA Decryption

Decryption posed two challenges: converting f to centred representation; and the implementation of RE-2-BV. To convert f to centred representation I considered each coefficient independently and converted it as required. RE-2-BV was implemented by using the in-built function that lists the coefficients of a polynomial.

Algorithm 11 DecCPA($c = (c_0, c_1), sk = s$)

Define l to be the length of c_0
$v \leftarrow s \cdot c_1$
$t \leftarrow$ Trunc(v, l) ▷ Trunc removes powers of x greater than l
$f \leftarrow c_0 - t$
Convert f into centred-representation
$\mu \leftarrow ||\lfloor \frac{2}{q} f \rceil|$
$m \leftarrow$ RE-2-BV(μ) ▷ RE-2-BV converts μ into a binary vector in $\{0,1\}^l$
return m

5.2.3 CCA Scheme

Even though this scheme was not used in my tests. It is not a KEM but rather is a CCA secure encryption-decryption scheme. I decided to implement it for completeness as it is mentioned in the paper to show the advantages of the KEM.

5.2.3.1 CCA Encryption

I had to make one design decision during implementation of this function, that was, the choice of hash function, H. I chose to use SHA-3-256 for consistency with Kyber and as it is a suitably secure hash function.

Algorithm 12 EncCCA($m \in \{0,1\}^l, pk = (a, b)$)

$u \leftarrow \{0,1\}^\lambda$ ▷ λ is the security parameter
$\mu \leftarrow m||u$
$r \leftarrow \mathrm{H}(\mu)$
$(c_0, c_1) \leftarrow \mathrm{EncCPA}(\mu, pk, r)$
return $c = (c_0, c_1)$

5.2.4 CCA Decryption

As I had made a choice for H in encryption and this needed to be the same for decryption, I had no further choices to make.

Algorithm 13 DecCCA($c = (c_0, c_1), sk = s$)

$\mu \leftarrow \mathrm{DecCPA}(c, sk)$
$m||u \leftarrow \mu$, where u is λ bits long ▷ λ is the security parameter
$r \leftarrow \mathrm{H}(\mu)$
$c' \leftarrow \mathrm{EncCPA}(\mu, pk, r)$
if $c \neq c'$ **then**
    return $\perp$
else
    return m
end if

5.2.5 CCA KEM

5.2.5.1 CCA Encapsulation

The encapsulation function required me to make choices for the hash function, H and the key derivation function, K. I chose to use SHA-3-256 as H to keep it as similar to Kyber as possible and SHAKE-256 for K as it is a suitably secure hash function that allows for keys of any length to be outputted.

Algorithm 14 EncapCCA($l, l', pk = (a, b)$)

$x \leftarrow \{0,1\}^l$
$r \leftarrow \mathrm{H}(x)$
$(c_0, c_1) \leftarrow \mathrm{EncCPA}(x, pk, r)$
$k \leftarrow K^{(l')}(x)$ ▷ K is the key derivation function
return $(c = (c_0, c_1), k)$

5.2.5.2 CCA Decapsulation

Decapsulation needs to use the same hash functions as encapsulation which left me with no decisions to be made as to its implementation.

Algorithm 15 DecapCCA($l', c = (c_0, c_1), sk = s$)

1: $x \leftarrow \text{DecCPA}(c, sk)$
2: $r \leftarrow \text{H}(x)$
3: $c' \leftarrow \text{EncCPA}(x, pk, r)$
4: **if** $c \neq c'$ **then**
5: **return** $\perp$
6: **else**
7: $k \leftarrow K^{(l')}(x)$ ▷ K is the key derivation function
8: **return** k
9: **end if**

5.3 Hash Functions

Both algorithms make use of hash functions in their KEMs. As mentioned I made use the SHA-3 and SHAKE-256 functions detailed in FIPS 202 (*SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, 2015) as they are cryptographic hash functions and were suggested in Kyber. Initially I implemented these functions myself to ensure my understanding was correct. While my implementation was correct it was rather inefficient and thus made it difficult to determine the efficiencies of the two schemes. This meant that I needed to use the built-in python SHA-3 library in my final implementation.

6 Testing

I wrote tests to test my code. Due to the random data used to encrypt messages and thus its non-deterministic property it is not possible to calculate test cases and confirm that my implementation does indeed return the correct result. Instead I simply checked that, for the CPA and CCA schemes in LIMA and the CPA scheme in Kyber, encrypted messages were correctly decrypted to the original message. For the Key Encapsulation Mechanisms, I checked that encapsulation and decapsulation returned the same key. I ran this test for a number of trials and this allowed me to find some coding bugs that were then rectified.

7 Parameter Selection

In order to compare the two schemes I needed to find parameters for various security levels for each scheme. These levels were predetermined by the parameters provided in Kyber as 128-bit, 192-bit and 256-bit security.

7.1 Kyber

Kyber provided parameters for 128-bit, 192-bit and 256-bit security in their C implementation of Kyber so I used these parameters for my tests.

	q	n	k	η	d_t	d_u	d_v	Security
I	7681	256	2	5	11	11	3	128
II	7681	256	3	4	11	11	3	192
III	7681	256	4	3	11	11	3	256

7.2 LIMA

Exact parameters for this scheme were not provided in the paper, instead they state that 'suitable choices of the parameters can be selected for given values of λ'. Therefore, in order to compare efficiency for the same security level I needed to find some suitable parameters for each of 128-bit, 192-bit and 256-bit security. To do this I expanded on the work in 'Even More Practical Key Exchanges for the Internet using Lattice Cryptography' (Singh, V. and Chopra, A., 2015) and, by approximating the Ring-LWE problem to a LWE problem used an LWE estimator (Albrecht, M.R., Player, R. and Scott, S., 2015) to estimate the security over a range of suitable parameters. I later discovered that in the NIST submission of LIMA, parameters were provided for various security levels. However, by this time I had already completed all my tests and gathered my results using my chosen parameters. Although these parameters were different to my parameters, due to a change that had been made to the LWE-estimator when their parameters were generated, they were similar enough to lead me to suspect similar results.

In order to generate the following parameters I ensured that the values I found were consistent with the properties laid out in 'Even More Practical Key Exchanges for the Internet using Lattice Cryptography' (Singh, V. and Chopra, A., 2015). That is, m is a prime and lattice dimension $n = \phi(m) = m - 1$ and the modulus q satisfies the following conditions. The security analysis in 'Lattice Cryptography for the Internet' (Peikert, C., 2014) gives a practical bound on the modulus, that is $q \approx n^{(3/2)}$. I also ensured I chose $q \equiv 1 \pmod{m}$ to maintain consistency with the proof of security for Ring-LWE. Therefore I searched for the smallest $q > n^{(3/2)}$ that is congruent to 1 modulo m that achieved the target security levels of 128-bit, 192-bit and 256-bit.

	m	n	q	σ	Security
II*	433	432	35507	3.192	128
IV*	631	630	44171	3.192	192
VI*	821	820	49261	3.192	256
I	439	438	49169	3.192	128
II	719	718	51769	3.192	192
III	971	970	54377	3.192	256

In this table of parameters, **II***, **IV*** and **VI*** are results from 'Even More Practical Key Exchanges for the Internet using Lattice Cryptography' (Singh, V. and Chopra, A., 2015). The others are the new parameters I found.

		MitM		**uSVP**		**Dec**		**Dual**		**BKW**	
	Target	rop	mem	rop	red	rop	red	rop	red	rop	mem
II*	128	376	366	135	135	144	144	152	152	139	**125**
IV*	192	543	533	247	247	258	258	262	262	188	**174**
VI*	256	703	692	373	373	384	384	394	394	237	**222**
I	128	381	371	131	131	140	140	152	152	143	**129**
II	192	617	606	300	300	311	311	316	316	207	**192**
III	256	829	818	483	483	491	491	502	502	271	**256**

In this table of estimated security levels shows the results from the LWE estimator for each of the chosen parameters. The critical value is shown in bold. I have included updated results from 'Even More Practical Key Exchanges for the Internet using Lattice Cryptography' (Singh, V. and Chopra, A., 2015) to show why it was necessary to find new parameters.

8 Analysis

In order to compare the two cryptoschemes, I chose to consider efficiency, public key size and cipher size. Efficiency is important to a cryptoscheme as any computation that uses encrypted data should be able to encrypt or decrypt with little increase in computation time. Public key and cipher sizes are important since they will be the data that is sent from client to client. It would therefore be helpful if this could all be sent in one packet (∼12000 bits) to improve the speed of transmission of keys and ciphers.

8.1 Efficiency

In order to test the efficiencies of the two schemes, I ensured I implemented them as similarly as possible to ensure fairness and then ran a series of tests for different security parameters to determine the average time taken over 1000 encapsulations and decapsulations. The results are summarised below. Times are in seconds, given to 4 significant figures.

8.1.1 Kyber

	Security	KeyGen	Encapsulation	Decapsulation
I	128	0.1234	0.03066	0.03069
II	192	0.1323	0.04338	0.04342
III	256	0.1414	0.05504	0.05510

8.1.2 LIMA

	Security	KeyGen	Encapsulation	Decapsulation
I	128	0.005349	0.008026	0.008038
II	192	0.007520	0.01043	0.01045
III	256	0.009663	0.01332	0.01333

From these results it is clear that Kyber is a less efficient system as encapsulation and decapsulation take around 4 times as long. This is to be expected since Kyber requires the multiplication of matrices of polynomials whereas LIMA only requires the multiplication of single polynomials. It might be expected that Kyber would be k^2 times less efficient since it does k^2 many times more multiplications. However, n remains constant in Kyber but increases with the security in Ring-LWE which counteracts this.

8.2 Key Size

I did not use my implementation to analyse key size because it is more helpful to consider theoretical public key size. Therefore, I needed to derive equations that determine the size for each of the security levels of each scheme.

8.2.1 Kyber

In Kyber, $pk = (\mathbf{t}, \rho)$ where $\mathbf{t} \in R_{2^{d_t}}^k$ and $\rho \in \{0,1\}^{256}$. If we assume that we can convert this to a simple bit string we can conclude that each coefficient of $\mathbf{t}$ requires d_t bits and we will require n of these coefficients for each of the k elements of $\mathbf{t}$. Hence we arrive at the following equation for the size of pk:

$$|pk| = |\mathbf{t}| + |\rho| = (k \cdot (n \cdot d_t)) + 256$$

Evaluating this for our parameters we compute the following table:

	Security	k	n	d_t	$\lvert pk \rvert$
I	128	2	256	11	5888 bits
II	192	3	256	11	8704 bits
III	256	4	256	11	11520 bits

8.2.2 LIMA

In LIMA, $pk = (a, b)$ where a and b are polynomials of degree n with coefficients taken modulo q. If we assume that we can convert this to a simple bit string we can conclude that each of the n coefficients of a and b will require $\lceil \log_2 q \rceil$ bits. Hence we arrive at the following equation for the size of pk:

$$|pk| = |a| + |b| = 2 \cdot (n \cdot \lceil \log_2 q \rceil)$$

Evaluating this for our parameters we compute the following table:

	Security	n	q	$\lceil \log_2 q \rceil$	$\lvert pk \rvert$
I	128	438	49169	16	14016 bits
II	192	718	51769	16	22976 bits
III	256	970	54377	16	31040 bits

Kyber is clearly better when it comes to key size, the public key for each of the security levels can fit inside one ethernet packet whereas 128-bit and 192-bit LIMA require two packets and 256-bit requires three. This is because instead of sending the value of a, Kyber sends the 256 bit seed used to generate a. Furthermore, the dimension, n, is smaller and the coefficients are compressed.

8.3 Cipher Size

I used a similar approach as in key size for determining cipher size and thus needed to derive similar equations for the size of cipher for each of the security levels of each scheme.

8.3.1 Kyber

In Kyber, $c = (\mathbf{u}, v, d)$ where $\mathbf{u} \in R_{2^{d_u}}^k$, $v \in R_{2^{d_v}}$ and $d \in \{0,1\}^{256}$. If we assume we can convert this to a simple bit string we can conclude that each coefficient of $\mathbf{u}$ requires d_u bits and we will require n of these coefficients for each of the k elements of $\mathbf{u}$. Also, each coefficient of v will require d_v bits and there are n of these coefficients. Hence we arrive at the following equation for the size of c:

$$|c| = |\mathbf{u}| + |v| + |u| = (k \cdot (n \cdot d_u)) + (n \cdot d_v) + 256$$

Evaluating this for our parameters we compute the following table:

	Security	k	n	d_u	d_v	$\lvert c\rvert$
I	128	2	256	11	3	6656 bits
II	192	3	256	11	3	9472 bits
III	256	4	256	11	3	12288 bits

8.3.2 LIMA

In LIMA, $c = (c_0, c_1)$ where c_0 is a polynomial with l coefficients each taken modulo q and c_1 is a polynomial with n coefficients each taken modulo q. If we assume we can convert this to a simple bit string we can conclude that each coefficient of c_0 and c_1 will require $\lceil \log_2 q \rceil$ bits and c_0 will require l coefficients whereas c_1 will require n coefficients. Hence we arrive at the following equation for the size of c:

$$|c| = |c_0| + |c_1| = (l \cdot \lceil \log_2 q \rceil) + (n \cdot \lceil \log 2q \rceil) = (l + n) \cdot \lceil \log_2 q \rceil$$

Evaluating this for our parameters we compute the following table:

	Security	n	l	q	$\lceil \log_2 q \rceil$	$\lvert c\rvert$
I	128	438	256	49169	16	11584 bits
II	192	718	256	51769	16	15584 bits
III	256	970	256	54372	16	19616 bits

Again, Kyber is better when it comes to cipher size, however by less of a margin if you consider the number of packets required to transfer a cipher. 128-bit and 192-bit only require one packet however 256-bit requires two packets in Kyber

whereas 128-bit requires one packet and 192-bit and 256-bit require two packets in LIMA. Therefore Kyber only has an advantage for 192-bit security in terms of packets required for transmission even though far fewer bits are required for all three security parameters.

9 Evaluation

My results show that while Kyber has smaller public key and cipher sizes, it is less efficient than the LIMA scheme. Therefore the suitability of either scheme will depend on the application and a judgement cannot be made on which is most suitable in every situation. However, considering two applications, one where time is the most valuable resource and the other where network load is the most valuable resource one can see that LIMA would be the most suitable choice for the first situation and Kyber for the second. One factor to consider is that Kyber will always be less efficient than the LIMA scheme since it requires the multiplication of more polynomials but it may be possible to use some of the techniques used in Kyber to improve the sizes of keys and ciphers in the LIMA scheme. That is, instead of passing the pair (a, b), one could use the same tactic as Kyber and instead pass the pair (ρ, b) where ρ is the seed used to generate a, or one could compress b and the cipher in a similar way to Kyber. If these improvements could be made, then LIMA would be the most suitable of the two schemes in either situation. In the following section I propose a scheme with these improvements. This, however, has no proof of security or correctness.

10 Improved Hybrid Scheme

As previously mentioned it is easy to imagine how the compression mechanisms and passing of seeds rather than polynomials in Kyber can be applied to LIMA. In this section I detail a hybrid scheme that includes these changes.

10.1 Helper Functions

Like Kyber, this scheme requires some helper functions used to compress and decompress polynomial coefficients.

10.1.1 Compress$_q$

As in Kyber, Compress$_q$ takes an input, x, and compresses all the coefficients of x, that are at most q to integers taken modulo 2^d. However since this will only ever be called with polynomials, not with matrices or vectors of polynomials, we do not need to worry about checking the dimension of such a matrix and can just convert x directly. It is defined as follows, where $x \in R_q$ and $d < \lceil \log_2 q \rceil$.

Algorithm 16 Compress$_q(x \in R_q, d)$

1: $x' \leftarrow \lceil (2^d/q) \cdot x \rfloor \bmod 2^d$
2: **return** x'

10.1.2 Decompress$_q$

Decompress$_q$ is also defined similar to that in Kyber. It takes an input, x, and decompresses all the coefficients of x, that are at most 2^d to integers taken modulo q. Again, as this will only ever be called with polynomials, we can convert x directly. It is defined as follows, where $x \in R_{2^d}$ and $d < \lceil \log_2 q \rceil$.

Algorithm 17 Decompress$_q(x \in R_{2^d}, d)$

1: $x' \leftarrow \lceil (q/2^d) \cdot x \rfloor$
2: **return** x'

10.2 Key Generation

Key generation is almost identical to that in LIMA with the exception that a is generated with random coins ρ and then the public key consists of this seed and the compressed polynomial b. These changes work together to reduce the size of the public key drastically.

Algorithm 18 HybridKeyGen()

1: $\rho \leftarrow \{0,1\}^{256}$
2: $a \leftarrow_{\rho} R_q$ ▷ Random coins ρ used as sampling seed
3: $s \leftarrow \chi_{\sigma}^{N}$
4: $e \leftarrow \chi_{\sigma}^{N}$
5: $b \leftarrow \text{Compress}_q(a \cdot s + e, d_b)$
6: $sk \leftarrow s$
7: $pk \leftarrow (\rho, b)$
8: **return** (pk, sk)

10.3 CPA Scheme

The CPA scheme is very similar to LIMA, with only some minor changes to allow for the compression of polynomials and the fact that the first element of the public key is not the polynomial a but rather the seed used to generate a.

10.3.1 CPA Encryption

As mentioned, CPA encryption is very similar to that in LIMA. The first difference is that the polynomial a had to be generated using ρ at the start of the function. The second difference is that b needs to be decompressed before it can be used. And the final difference is that both elements of the cipher are compressed. This compression is what reduces the size of the cipher when compared to LIMA.

Algorithm 19 HybridEncCPA($m \in \{0,1\}^l, pk = (\rho, b), r$)

1: $a \leftarrow_{\rho} R_q$ ▷ Random coins ρ used as sampling seed
2: $b' \leftarrow \text{Decompress}_q(b, d_b)$
3: $\mu \leftarrow$ BV-2-RE(m) ▷ BV-2-RE converts m to a polynomial in R_q
4: $v, e, d \leftarrow_r \chi_{\sigma}^{N}$ ▷ Random coins r used as sampling seed
5: $x \leftarrow d + \lfloor q/2 \rfloor \cdot \mu \pmod q$
6: $t \leftarrow b' \cdot v + x$
7: $t' \leftarrow \text{Trunc}(t, l)$ ▷ Trunc removes powers of x greater than l
8: $c_0 \leftarrow \text{Compress}_q(t', d_{c_0})$
9: $c_1 \leftarrow \text{Compress}_q(a \cdot v + e, d_{c_1})$
10: **return** $c = (c_0, c_1)$

10.3.2 CPA Decryption

CPA decryption is also very similar to LIMA. There is only one change required, that is, that the elements of the cipher need to be decompressed using the Decompress_q function at the start of the function but after that the function is identical to LIMA.

Algorithm 20 HybridDecCPA($c = (c_0, c_1), sk = s$)

$c_0' \leftarrow \text{Decompress}_q(c_0, d_{c_0})$
$c_1' \leftarrow \text{Decompress}_q(c_1, d_{c_1})$
Define l to be the length of c_0'
$v \leftarrow s \cdot c_1'$
$t \leftarrow \text{Trunc}(v, l)$ ▷ Trunc removes powers of x greater than l
$f \leftarrow c_0' - t$
Convert f into centered-representation
$\mu \leftarrow ||\lfloor \frac{2}{q} f \rceil||$
$m \leftarrow \text{RE-2-BV}(\mu)$ ▷ RE-2-BV converts μ into a binary vector in $\{0,1\}^l$
return m

10.4 CCA KEM

The CCA Key Encapsulation Mechanism for this scheme uses the same algorithms as LIMA but with the new CPA encryption and decryption algorithms detailed above. This works since the KEM does not make use of the public key or cipher directly but simply inputs them into the relevant CPA algorithm.

10.5 Public Key and Cipher Sizes

These improvements will slightly decrease the efficiency of the scheme in comparison to LIMA but it should still be more efficient than Kyber. It will however improve the size of public key and cipher size to be more similar to those in Kyber. Again, I calculated equations for both the public key size and cipher size.

In this scheme $pk = (\rho, b)$ where $\rho \in \{0,1\}^{256}$ and b is a polynomial with n coefficients each taken modulo 2^{d_b}. If we assume we can convert this to a simple bit string we can conclude that each coefficient of b will require d_b bits and we will have n of these coefficients. Hence we arrive at the following equation for the size of pk:

$$|pk| = |\rho| + |b| = 256 + n \cdot d_b$$

Furthermore, $c = (c_0, c_1)$ where c_0 is a polynomial with l coefficients taken modulo $2^{d_{c_0}}$ and c_1 is a polynomial with n coefficients each taken modulo $2^{d_{c_1}}$. If we assume we can convert this to a simple bit string we can conclude that each coefficient of c_0 will require d_{c_0} bits and there will be l such coefficients, whereas each coefficient of c_1 will require d_{c_1} bits and there will be n such coefficients. Hence we arrive at the following equation for the size of c:

$$|c| = |c_0| + |c_1| = l \cdot d_{c_0} + n \cdot d_{c_1}$$

I ran various tests with different values and have found the following estimated parameters: $d_b = 11$, $d_{c_0} = 14$ and $d_{c_1} = 15$. However, I provide no proof of security or correctness so this would be required before making such changes to LIMA. Evaluating the above equations for these parameters we compute the

following tables:

	Security	n	d_b	d_{c_0}	d_{c_1}	$\|pk\|$	$\|c\|$
I	128	438	11	14	15	5074 bits	10154 bits
II	192	718	11	14	15	8154 bits	14354 bits
III	256	970	11	14	15	10926 bits	18134 bits

This has caused a massive improvement in the size of the public key, mostly due to replacing a with the seed used to generate a which points to this being a useful improvement. Unfortunately, during tests, I found that compression of the cipher often led to encryption errors and therefore the parameters d_{c_0} and d_{c_1} had to be large. This suggests that this improvement may not be worthwhile. Further research into the correctness of the scheme with such compression would be required to determine whether this indeed is the case.

11 Discussion

In this section I will discuss the achievements and deficiencies of my project and how I would have completed the project differently should I have had more time or if things had gone differently.

This project has taught me a great deal about a lot of new areas of computer science and deepened my knowledge of others. In particular, my knowledge of the LWE family of problems has improved and a few misconceptions I had have been rectified. Furthermore I have learnt about the different security levels of cryptosystems, namely, CPA and CCA security and how Key Encapsulation Mechanisms can make use of Hash Functions to convert a CPA secure scheme to a CCA secure one. Using this knowledge I was able to successfully implement two CCA-secure post-quantum cryptosystems in SageMath and use these implementations, along with my knowledge of the schemes, to compare and analyse the following features: efficiency; public key size; and cipher size. Following this I was able to use this analysis to propose a new scheme that is a hybrid of the two, making use of the efficiency of LIMA and the improvements made to the public key size and cipher size of Kyber.

One deficiency in my project is the parameter selection for LIMA. I should have realised sooner that parameters for LIMA were provided in the submission to NIST even though they were not in the paper I was reading as this meant I spent a large amount of time finding my own parameters. Secondly, I did not manage to complete my project quick enough to allow me to implement the schemes in C, which would have been more efficient than in SageMath. If I had had more time I would have liked to be able to do this, in order to compare the two schemes in a lower level language. Finally, if I had had more time I would have liked to be able to provide some proof of security and correctness for my hybrid scheme.

Despite these deficiencies, I feel my project has been a success. I have completed everything I set out to do with the exception of implementing in C. Furthermore I have proposed the hybrid scheme, something I did not expect to be able to do but that was the natural progression given my results.

12 References

Albrecht, M.R. and Deo, A. (2017) *Large Modulus Ring-LWE ≥ Module-LWE.* Available at: `https://eprint.iacr.org/2017/612.pdf` (Accessed: 12th March 2018)

Albrecht, M.R., Orsini, E., Paterson, K.G., Peer, G. and Smart N.P. (2017) *Tightly Secure Ring-LWE Based Key Encapsulation with Short Ciphertexts.* Available at: `https://eprint.iacr.org/2017/354.pdf` (Accessed: 29th November 2017).

Albrecht, M.R., Player, R. and Scott, S. (2015) *On the concrete hardness of Learning with Errors.* Available at: `https://eprint.iacr.org/2015/046.pdf` (Accessed: 20th Feburary 2018)

Albrecht, M.R. (2015) *Security Estimates for the Learning with Errors Problem.* Available at: `https://bitbucket.org/malb/lwe-estimator` (Accessed: 20th February 2018)

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P. and Stehlé, D. (2017) *CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM.* Available at: `https://eprint.iacr.org/2017/634.pdf` (Accessed: 10th December 2017).

Lyubashevsky, V., Peikert, C. and Regev, O. (2013) *On Ideal Lattices and Learning with Errors Over Rings.* Available at: `https://eprint.iacr.org/2012/230.pdf` (Accessed: 28th October 2017)

NIST (2017) *Post-Quantum Cryptography.* Available at: `https://csrc.nist.gov/projects/post-quantum-cryptography` (Accessed: 9th November 2017).

NIST (2015) *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* Available at: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf` (Accessed: 15th December 2017)

Peikert, C., Regev, O. and Stephens-Davidowitz, N. (2017) *Pseudorandomness of Ring-LWE for Any Ring and Modulus.* Available at: `https://eprint.iacr.org/2017/258.pdf` (Accessed: 15th March 2018)

Peikert, C. (2014) *Lattice Cryptography for the Internet.* Available at: `https://eprint.iacr.org/2014/070.pdf` (Accessed: 22nd February 2018)

Regev, O. (2009) *On Lattices, Learning with Errors, Random Linear Codes, and Cryptography.* Available at: `https://cims.nyu.edu/~regev/papers/qcrypto.pdf` (Accessed: 10th November 2017)

Singh, A. and Chopra, A. (2015) *Even More Practical Key Exchanges for the Internet using Lattice Cryptography.* Available at: `https://eprint.iacr.org/2015/1120.pdf` (Accessed: 15th February 2018)

Tomescu, A. (2011) *CPA and CCA Security.* Available at: `http://people.csail.mit.edu/alinush/cse508-spring-2011/03-03-cpa-and-cca.pdf` (Accessed: 5th Dec. 2017)

13 Appendix

13.1 File Structure

The following directories are contained within my .zip file.

13.1.1 Hybrid Scheme

```
Hybrid Scheme
├── Results ..... Results for Hybrid Scheme
│   └── Tests_results.txt ..... Tests.py Results
├── Sage Files ..... SageMath Source Code
│   ├── fips202.sage ..... FIPS 202 - Only use h2b Function
│   ├── Hybrid_CPA.sage ..... CPA Scheme Functions
│   ├── Hybrid_KEM.sage ..... KEM Scheme Functions
│   └── Tests.sage ..... Tests for CPA and KEM Schemes
├── fips202.py ..... Sage Preparse of fips202.sage
├── Hybrid_CPA.py ..... Sage Preparse of Hybrid_CPA.sage
├── Hybrid_KEM.py ..... Sage Preparse of Hybrid_KEM.sage
└── Tests.py ..... Sage Preparse of Tests.sage
```

13.1.2 Kyber

```
Kyber
├── Results ..... Results for Kyber
│   ├── Kyber_128_results.txt ..... Kyber_128_Test.py Results
│   ├── Kyber_192_results.txt ..... Kyber_192_Test.py Results
│   ├── Kyber_256_results.txt ..... Kyber_256_Test.py Results
│   └── Tests_results.txt ..... Tests.py Results
├── Sage Files ..... SageMath Source Code
│   ├── fips202.sage ..... FIPS 202 - Only use h2b Function
│   ├── Kyber_128_Test.sage ..... 128-bit Parameters Test
│   ├── Kyber_192_Test.sage ..... 192-bit Parameters Test
│   ├── Kyber_256_Test.sage ..... 256-bit Parameters Test
│   ├── Kyber_CPA.sage ..... CPA Scheme Functions
│   ├── Kyber_Helper.sage ..... Helper Functions
│   ├── Kyber_KEM.sage ..... KEM Scheme Functions
│   └── Tests.sage ..... Tests for CPA and KEM Schemes
├── fips202.py ..... Sage Preparse of fips202.sage
├── Kyber_128_Test.py ..... Sage Preparse of Kyber_128_Test.sage
├── Kyber_192_Test.py ..... Sage Preparse of Kyber_192_Test.sage
├── Kyber_256_Test.py ..... Sage Preparse of Kyber_256_Test.sage
├── Kyber_CPA.py ..... Sage Preparse of Kyber_CPA.sage
├── Kyber_Helper.py ..... Sage Preparse of Kyber_Helper.sage
├── Kyber_KEM.py ..... Sage Preparse of Kyber_KEM.sage
└── Tests.py ..... Sage Preparse of Tests.sage
```

13.1.3 LIMA

- LIMA
  - Results ..... **Results for LIMA**
    - LIMA_128_Results.txt ..... LIMA_128_Test.py Results
    - LIMA_192_Results.txt ..... LIMA_192_Test.py Results
    - LIMA_256_Results.txt ..... LIMA_256_Test.py Results
    - Test_results.txt ..... Tests.py Results
  - Sage Files ..... **SageMath Source Code**
    - fips202.sage ..... FIPS 202 - Only use h2b Function
    - LIMA_128_Test.sage ..... 128-bit Parameters Test
    - LIMA_192_Test.sage ..... 192-bit Parameters Test
    - LIMA_256_Test.sage ..... 256-bit Parameters Test
    - LIMA_CCA.sage ..... CCA Scheme Functions
    - LIMA_CPA.sage ..... CPA Scheme Functions
    - LIMA_KEM.sage ..... KEM Scheme Functions
    - LIMA_Parameter_Search.sage ..... Code used to find Parameters
    - Tests.sage ..... Tests for CPA, CCA and KEM Schemes
  - fips202.py ..... Sage Preparse of fips202.sage
  - LIMA_128_Test.py ..... Sage Preparse of LIMA_128_Test.sage
  - LIMA_192_Test.py ..... Sage Preparse of LIMA_192_Test.sage
  - LIMA_256_Test.py ..... Sage Preparse of LIMA_256_Test.sage
  - LIMA_CCA.py ..... Sage Preparse of LIMA_CCA.sage
  - LIMA_CPA.py ..... Sage Preparse of LIMA_CPA.sage
  - LIMA_KEM.py ..... Sage Preparse of LIMA_KEM.sage
  - LIMA_Parameter_Search.py ..... Sage Preparse of LIMA_Parameter_Search.sage
  - Tests.py ..... Sage Preparse of Tests.sage

13.2 Running the Code

Details of how to run the code is provided in README.md in the .zip file. Here is how to run the code:

- Navigate to your installation of SageMath
- Run ./sage path_to_test_file

For example, my installation of SageMath is in a directory called SageMath that is in the same directory as the parent (called bryer1523503) of 'Hybrid Scheme', 'Kyber' and 'LIMA'. To run the 128-bit test of LIMA I would run the following command from the SageMath directory:

- ./sage ../bryer1523503/LIMA/LIMA_128_Test.py

13.3 Project Proposal

Introduction

Since quantum computers were theorised and lately realised it has become clear that the security of the majority of current encryption technologies will be severely compromised as a result of quantum algorithms that reduce the difficulty of the problems on which they are based. An example of this is RSA, a cryptosystem that relies on the factoring of large numbers not being polynomial time. Shor's algorithm, a quantum algorithm, however can factorise an integer N in polynomial time in logN, which would result in RSA not longer being a secure way to encrypt.

In 2016 the US standards body NIST launched a Post Quantum Crypto (PQC) project and published a call for submissions of quantum-resistant public-key cryptographic algorithms. The leading candidates for the problem on which to base such algorithms are the Learning with Errors (LWE) problem and it's ring counterpart, the Ring Learning with Errors (RLWE) problem. There have been many submissions for algorithms based on these problems however often the emphasis has been on designing algorithms that are CPA-secure (Chosen Plaintext Attack) and not necessarily CCA-secure (Chosen Ciphertext Attack).

For my project I will be studying and implementing two algorithms that have proven to be CCA-secure, detailed in the papers, 'Tightly Secure Ring-LWE Based Key Encapsulation with Short Ciphertexts' and 'CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM'. Using these implementations I will gather data in the interest of studying efficiency and use this along with the key-sizes to determine which I believe to be the most suitable for real-world application.

Solution

A large proportion of the project will be spent studying and understanding the mathematics behind each algorithm. I will need to ensure I have an in depth understanding of the LWE problem, RLWE problem and the key encapsulation method of converting a CPA-secure scheme to a CCA-secure scheme. I will also need to ensure I understand the two attack methods and how the two schemes prevent against them. Furthermore I will need to study the algorithms so that I know exactly how they work to make sure there are no errors in my implementation so that I am indeed studying the efficiency of the algorithms in the papers.

I will initially aim to create a naive implementation of each scheme in SageMath, an extension of the python programming language, and then do some research into suitable parameters for either scheme to ensure certain security levels, something that requires careful consideration of many factors.

Once I have a naive implementation for both schemes that correctly generates

keys, encrypts messages and decrypts ciphers I can begin to start optimising the functions in order to improve efficiency as much as possible in the language. I expect to find this quite difficult as my initial implementation will hopefully already be quite efficient so finding areas in which to improve may require a lot of work.

After optimising each implementation I will use the previous research in which I found suitable parameters for different security levels and write tests in order to measure the efficiency of each scheme at different security levels. I will then analyse this data in order to compare the efficiency of the two algorithms and to draw conclusions as to which would be more suitable for real-world application.

If I implemented the algorithms in C I would expect to get much faster implementations due to the C being a low-level language where Python and SageMath are high-level languages. Because of this, towards the end of my project, if time permits, I will aim to implement both algorithms in C to get a comparison of efficiency in a faster language. I would expect to get the similar results from either language scaled by the difference in the languages respective speeds.

Requirements

Software

Due to the theoretical basis of my problem I will not need any software other than the SageMath library, python programming language, C programming language and relevant compilers provided by the corresponding languages.

Hardware

All my implementation and testing will be done on my personal laptop. When I am collecting data I may need to run tests overnight without interruption so I would need a university machine that I can leave running and come back to at a later date.

Timeline