Imperial College London

Department of Electrical and Electronic Engineering

Final Project Report 2018

| | |
|---|---|
| Project Title: | **Post-Quantum Key Exchange over the Internet** |
| Student: | **Afraz Arif Khan** |
| CID: | **00963429** |
| Course: | **EE4T** |
| Project Supervisor: | **Dr Cong Ling** |
| Second Marker: | **Prof Pier Luigi Dragotti** |

# Abstract

Quantum computing can open many doors for researchers, mathematicians, engineers and computer scientists. With new research projects such as D-Wave showing great potential behind quantum computers, cryptographers are becoming increasingly alarmed as earlier unbreakable mathematical principles like the discrete logarithm problem and the integer factorization problem can now be solved in polynomial time. These mathematical properties form the basis of some of the worlds most widely used public-key cryptography algorithms. As a result alternative quantum-resistant cryptographic protocols are now being explored. One popular variant is Lattice Cryptography which is largely based on the famous Learning with Errors Problem. Lattice Cryptography offers a very promising response to post quantum cryptography due to its uncomplicated structure and its (provable) security towards quantum computers. In fact most researchers have shown promising post-quantum key exchanges like Frodo, NewHope and the famous BCNS protocol. This document implements a Diffie Hellman like key exchange based on the commutativity of matrices which is secure against quantum computers based on the Hermite-Normal Form Learning With Errors Assumption. It extends the implementation of this key exchange from single bits to multiple bits of shared secrecy to repel brute force attacks while retaining its resistance to quantum computers. It also explores the integration of discrete gaussian distributions which can fundamentally be used for further post-quantum research or in other projects. Most post-quantum key exchanges based on the simple learning with errors problem are usually slow due to the large computation time required to generate matrices of higher dimensions. This paper explores different lattice dimensions and offers a significant trade off between quantum security and speed and yields a positive response on the optimal performance of the key exchange for 38ms (on average) in Python and 30ms (on average) in C of computation time on a Lattice Dimension of 512.

# Acknowledgements

I would like to thank my supervisor Dr Cong Ling for his help in the project and Mr Jiabo Wang for all his guidance and help. I would like to thank my parents for supporting me throughout these years at university and my sister for her unconditional advice and guidance. I would also like to thank all my friends at Imperial and back home for their unwavering advice both directly on this project and indirectly during my stay at Imperial College London. Finally, I would like to thank the teachers at this university who have made it possible for me to acquire the correct skills needed for this project.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Abbreviations

PQC - Post-Quantum Cryptography/Cryptographic
PQKE - Post-Quantum Key Exchange
NIST - National Institute of Science & Technology
DH - Diffie Hellman (Key Exchange)
ECDH - Elliptic Curve Diffie Hellman
RSA - Rivest, Shamir and Adelman's Encryption Algorithm
TLS - Transport Layer Security
SSL - Secure Sockets Layer
API - Application Programming Interface
LWE - Learning With Errors
RLWE - Ring-Learning With Errors
DGS - Discrete Gaussian Statistical Library
AWS - Amazon Web Services (Cloud Computing Platform)

# Preliminaries

**Notations** In this document bold capital faced letters are used to represent matrices such as $\mathbf{A}$ and bold lower case letters are used to represent vectors such as $\mathbf{a}$. The notation $\mathbf{A}^T$ represents the transpose of the matrix $\mathbf{A}$. A matrix $\mathbf{A}$ will use $m \times n$ to denote the number of rows and the number of columns respectively. The notation $\mathbf{A}[i][j]$ denotes the $i$-th row and the $j$-th column of the matrix $\mathbf{A}$ where $i \in 0, 1, 2, ..., m$ and $j \in 0, 1, 2, ..., n$. A matrix can be indexed in an equation by a decomposed vector format as $\mathbf{s}_A^1$ to $\mathbf{s}_A^n$ which corresponds to $n$ different vectors placed in each row of the matrix $\mathbf{S}_A$ sampled from the same distribution. The notation $\mathrm{s}_{A(1)}^1$ corresponds to the 1st entry of vector $\mathbf{s}_A^1$. The following Mathematical notations are summarized in the table below:

| | |
|---|---|
| $\mathbb{R}$ | The set of real numbers |
| $\mathbb{Z}$ | The set of all integers |
| $\mathbb{Z}^n$ | An n-dimensional vector of the set of integers |
| $\mathbb{Z}_q$ | The set of all integers modulo q |
| $\mathcal{D}_{\Lambda,\sigma,\mathbf{c}}$ | The discrete Gaussian Distribution over $\Lambda$ with center $\mathbf{c}$ and parameter $\sigma$ |

Table 1: Mathematical Notations used in this document

Unless stated otherwise the letter 'q' will formally denote a very large prime number throughout this document. This document will also make use of Big-O ($\mathcal{O}$) notation to describe computational complexity. This `text` will be used to describe functions in programming languages and libraries for programming languages interchangeably. The notation $\mathrm{q} \xleftarrow{\$} [0, 1, 2, ..., N]$ refers to sampling a random number q from a uniform distribution comprised of the set of numbers from 0 to $N$. Unless stated otherwise, the logarithmic function will always have a base of 2 (that is $log_2(a)$ for some $a \neq 0$ and $a \in \mathbb{R}$). The parameter $\tau$ which is used for Discrete Gaussians is used to denote the range of values acceptable for rejection. This corresponds to a range of values from $c - \sigma\tau$ to $c + \sigma\tau$ (where $c$ denotes the center, $\sigma$ denotes the standard deviation and $\tau \in \mathbb{Z}^+$) that are accepted as values of the new Discrete distribution and all values outside this range are rejected (as they have negligible probability).

**Mathematical Operators** This document heavily relies on modular arithmetic where the notation $\mathrm{r} = a \bmod p$ (where $a \in \mathbb{Z}$ and $p \in \mathbb{Z}$) refers to the remainder when $a$ is divided by $p$. The function $\lfloor\rfloor$ denotes the `floor` function and rounds a real number down to its nearest integer value (for example $\lfloor 2.7 \rfloor = 2$). Conversely, the function $\lceil\rceil$ denotes the `ceil` function and rounds a real number up to the nearest integer value (for example $\lceil 2.3 \rceil = 3$). The function $\lfloor\rceil$ denotes the `round` function which rounds a real number to its nearest integer value (for example $\lfloor 2.8 \rceil = 3$ or $\lfloor 2.2 \rceil = 2$). The % operator in each programming language denotes the modulo operator but does not exactly correspond to modular arithmetic. It actually outputs the remainder and in the case of negative values it will output a negative remainder which is not possible in abstract algebra. This % operator will be used in algorithms and will be adjusted to correspond to modular arithmetic wherever necessary.

# 1 Introduction

This report will firstly highlight the motivations and the background behind different Post Quantum Cryptography implementations. It will then discuss the requirements of this project followed by an analysis and design review on the PQKE implemented in this paper. Then it will highlight the implementation behind all the software developed for this PQKE. By discussing important metrics in the testing section, this report will then comment on numerous parameters essential to the key exchange followed by a critical analysis on the test results. Finally, it will suggest some future work (including further optimisations) for the PQKE.

## 1.1 Motivation

Outside the initial plight of quantum computing as a work of science fiction, a new initiative by NASA and Google has shown that quantum computing is indeed possible [6] through their joint venture with D-Wave. A quantum computer does not work like a regular computer, in the sense that regular computers rely on only 2 known states in binary, simply "on" or "off." Quantum computers rely on many states formed through the working of qbits. Qbits can also co-exist in 2 states like a regular computer but can form even more states through the superposition of the initial 2 [7]. This enables a quantum computer to perform calculations that cannot be achieved with a pen and paper and consequently allow calculations that were initially thought to be impossible, hence opening new doors for mathematicians, chemists and engineers but posing a serious threat to cryptographers.

With the advent of quantum computing expected to transcend current computational limits, cryptographers are becoming increasingly alarmed by the newfound computation power available as previous unbreakable mathematical principles which prove the hardness of current cryptographic implementations like the Diffie-Hellman key exchange [8] or RSA encryption [9] would now be vulnerable to quantum computer attacks. Shor's Algorithm makes use of the quantum fourier transform and modular exponentiation by repeated squarings to factorise N prime integers in polynomial time [10] which can solve the integer factorization problem.

By developing post quantum cryptographic implementations, cryptographers can safeguard current cryptographic applications which are all heavily reliant on the D-H key exchange or on RSA encryption. Organizations like NIST (National Institute of Standards and Technology) [11] consider post quantum cryptography to be of paramount importance and have dedicated research into different quantum resistant techniques.

## 1.2 Project Specification

It is imperative that new cryptographic algorithms are unbreakable by quantum computers and are readily available to use in TLS (Transport Layer security). This project will deal with implementing a Post-Quantum Key Exchange into a programming language that could later guarantee its integration into TLS through OpenSSL. These aims can be accomplished by the following objectives:

- Perform a literature study already available PQKEs and select a PQKE that is not implemented in any programming language.

- Implement the PQKE in a programming language that would allow direct integration into OpenSSL.
- Analyse the performance of the PQKE based on select cryptographic metrics (such as algorithmic and communicational complexity).
- Performing a comparative assessment on already available implementations and the PQKE implemented in this project.

## 1.3 Report Structure

This report begins by introducing the required background material on cryptography and the mathematical foundations that build post-quantum key exchanges. It then discusses currently available PQKEs and also briefly discusses their security against quantum computer attacks. The next section of this report details the design requirements for generating a post-quantum key exchange. The proposed system design discusses these requirements and leads into the implementation process which details the underlying structure behind the PQKE, discrete gaussian distributions and selective stages of optimised levels of the PQKE. The report then explains the testing process and discusses the results followed by a conclusion on the evaluation of project and suggested future work.

# 2 Background

This section introduces the background understanding required to satisfy the aims of the project. It will define cryptographic terms needed to understand this paper, touch on current cryptographic implementations and their vulnerability to quantum computer attacks, introduce lattice based cryptography (and the required theory) as an emerging field for post quantum cryptography and then introduce and explain current implementations that form a part of the technical implementation of this project. It will also explain the TLS protocol followed by a discussion on an open-source library that hosts PQKEs implemented in TLS.

## 2.1 Definitions

**Cryptography** is a sub branch of engineering and computer science that focuses on writing codes to keep messages between people (namely the user and the client) private [12]. To grasp cryptographic fundamentals properly it is necessary to understand the nomenclature:

1. **Plaintext**: This is the "hidden" or "secret" message that is being exchanged between two parties.

2. **Ciphertext**: This is an encrypted message that is sent over a channel or any kind of medium.

3. **Key**: This is a secret mechanism used to encrypt the plaintext or decrypt the ciphertext (see more on symmetric cryptography). Keys are usually numbers or matrices that have significant bearings on the encryption algorithm and vary in both size and applicability subject to the current encryption algorithm.

4. **Encryption Algorithm**: A computer algorithm that takes in input the plaintext and a key and outputs the ciphertext.

5. **Decryption Algorithm**: A computer algorithm that takes in input the ciphertext and a key (not necessarily the same key!) and outputs the plaintext.

Cryptography is categorized as:

**Symmetric Cryptography**[1] Symmetric Cryptography is defined as the same key or a transformed version of it used in both the encryption algorithm and decryption algorithm [13]. Due to the fact that symmetric cryptography is very fast it is used for bulk messages on hardware like stream ciphers and block ciphers. It was widely used in Data Encryption Standard (block ciphers) which were developed by IBM in the 1970s [14]. Nowadays cryptographers favour Advanced Encryption Standard [15] for block ciphers. Symmetric Cryptography is not vulnerable to quantum computer attacks, Grover's algorithm [16] halves the key size used in Symmetric Cryptography and therefore seriously weakens it but according to NIST [17] using larger key sizes will maintain security.

[1]Commonly known as private-key cryptography.

**Asymmetric Cryptography** Asymmetric Cryptography involves different keys used in encryption and decryption algorithms. More specifically a public key is used to encrypt the plaintext and a private key is used to decrypt the ciphertext [18]. The public key is known to all parties and the private key must only be known to the owner of the key. It is widely used in digital signatures and key agreements over the Internet where digital signatures are used to confirm the authenticity of the sender of a message. An example of a public-key algorithm used for digital signatures is the Digital Signature Algorithm developed by NIST [19]. Asymmetric cryptography is vulnerable to quantum computer attacks through Shor's Algorithm [10] and as a result NIST has placed high priority [17] in finding secure post quantum cryptographic solutions which form the core of this project.

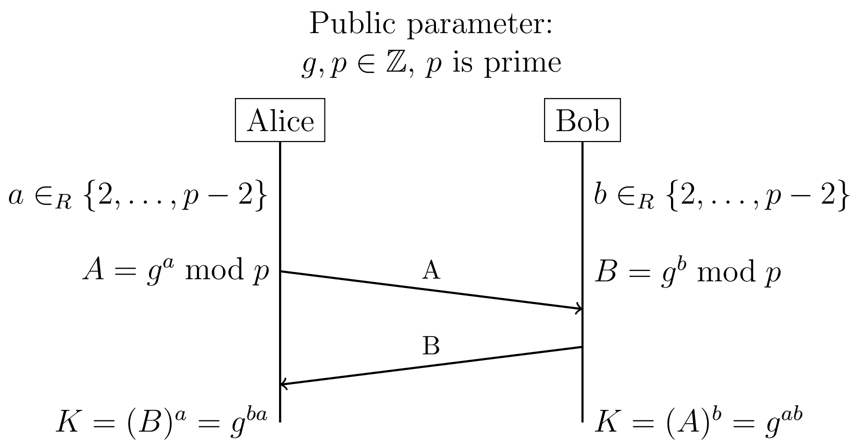
## 2.2 Existing Public Key Cryptography Algorithms

To fully grasp the aims and objectives of this project it is important to identify the need for post quantum cryptography which entails a short discussion on the vulnerabilities of current cryptographic implementations. This section presents a very short explanation on two very important cryptographic protocols widely used in everyday life and highlights their vulnerabilities to quantum computer attacks.

### 2.2.1 Diffie-Hellman Key Exchange

One of the most famous and widely applied key exchange algorithms is the Diffie-Hellman Key Exchange [8] where two parties, Alice and Bob, who have never met before are able to agree on the same value for a key.

**Key Exchange** Suppose Alice wishes to send a secret message $\mathbf{M}$ (where $\mathbf{M} \in \{0,1\}^n$) to Bob and wishes to make use of numerous encryption algorithms to encrypt her message. In order to make use of an encryption algorithm Alice must first choose a shared secret key bounded by the only condition that whatever key she chooses, Bob must have the same shared secret key. So in order to select the same key as Bob she will run the Diffie-Hellman Key exchange:



Protocol 1: Diffie-Hellman Key Exchange

Due to the property of the commutativity[2] of the multiplication of indices that arise from raising powers to powers Alice and Bob share the same key as $(B)^a = (A)^b$.

[2] $(g^a)^b \bmod p = (g^b)^a modp = g^{ab} \bmod p = g^{ba} \bmod p$

**Security and vulnerability to quantum computer attacks** There are many variants to the D-H Key Exchange as the original Diffie-Hellman Problem was considered insecure due to logjam attacks [20]. A relevant and practical example of those variants would be the utilization of the Elliptic Curve Diffie-Hellman (ECDH) [21] which makes use of Elliptic Curve Cryptography. This still makes use of the main property of hardness in the D-H Key Exchange known as the discrete logarithm problem [22]. Secure parameter selection for Internet applications require $p, a$ and $b$ to be very large (around 1024 bits for $p$) and $g$ is bounded by the cyclic group $G$ where $g = h^{(p-1)/q} \bmod p$ and $h$ is any integer with $1 < h < p - 1$ [23]. In fact finding the two most significant bits in the D-H Problem [24] is considered hard further cementing the D-H Problems hardness in modern cryptography. By relying on two modular exponentiations and two quantum fourier transforms Shor's algorithm [10] is efficiently able to solve the discrete logarithm problem in polynomial time. This seriously threatens all Internet applications that are heavily dependent on the D-H key exchange (and any variants of it) like establishing session keys in Internet protocols.

### 2.2.2 RSA Encryption

Envisioned by Rivest, Shamir and Adelman (dubbed as RSA from the authors) as a public key cryptographic encryption scheme, RSA encryption [9] deals with key construction, distribution, encryption and decryption. This section will only focus on key exchange in the key construction and distribution protocol and its inherent vulnerability to quantum computer attacks.

**Key Exchange** Like the D-H Key Exchange, RSA makes use of public keys and private keys. The public key is used to encrypt messages and the private key is used to decrypt them. The following points summarize the required steps for the key construction [25].

- Choose two distinct prime numbers $p$ and $q$.
- Find $n = pq$ (known as the modulus).
- Find the euler totient as $\phi(n) = (p - 1)(q - 1)$.
- Choose an integer $e$ such that $1 < e < \phi(n)$ and $e$ is co-prime[3] with $\phi(n)$.
- Choose $d$ such that $d = e^{-1}(\text{mod } \phi(n))$.

Here $e$ is the public key and $d$ is the private key. Both $p$ and $q$ must also be kept secret as they directly lead to $d$ and $e$. By choosing very large prime numbers the hardness of RSA encryption is preserved.

For key distribution:

- Alice chooses a secret message $\mathbf{M}$ where $\mathbf{M} \in \{0, 1\}^n$
- Alice needs Bob's public key to encrypt messages. Bob's public key includes $n$ and $e$ and consequently is sent as $(n, e)$ over a medium (the channel doesn't have to be secure).
- Bob will then decrypt the messages using his private key $d$.

[3]Integers $a$ and $b$ are said to be co-prime if $gcd(a, b) = 1$

**Security and vulnerability to quantum computer attacks** The hardness of RSA encryption is based on the secrecy of the integers $p$ and $q$. If these integers are very large then they cannot be computed using the computational power available in the current era of computing which is better known as the integer factorization problem. The integer factorization problem can formally be represented as:

$$\textit{Given an integer N, find its prime factors as } N = \prod p_i^{ei}$$

The best known algorithm which simplified the complexity of the integer factorization problem to $O(exp\sqrt[3]{\frac{64}{9}b(logb)^2})$ is called the general number sieve algorithm [26]. For arbitrarily large distinct integers it is hard to solve the integer factorization problem but Shor's algorithm [10] which makes use of periodic determinations using integers[4] and euclids greatest common divisor algorithm is able to solve it on a quantum computer in polynomial time. By determining the period of a particular integer Shor's algorithm is able to approximate a common period between $p$ and $q$ with overwhelming probability and therefore determine their relational assumption to compute $n$. This methodology would render RSA encryption insecure as the hardness of the integer factorization problem is no longer preserved.

## 2.3 Lattice Based Cryptography

This section will provide a short discussion on the background theory of lattices (groups) and the relevant branches of lattice based cryptography which are required for the Post-Quantum Key Exchanges present in this paper.

### 2.3.1 What are Lattices?

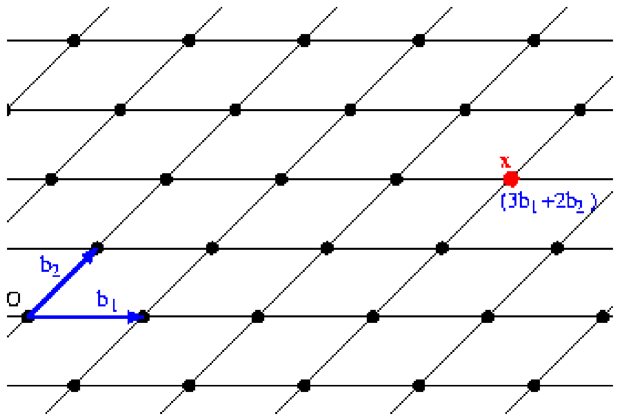


Figure 1: A Lattice in $\mathbb{R}^2$ [1]

A Lattice [27] is a discrete subgroup of Euclidean Space:

$$L = \text{discrete subgroup of } \mathbb{R}^n$$

Given that they form a subgroup in a group, they can be represented as linear combinations of independent basis vectors [28]:

$$L = \sum_i^n x_i \, \mathbf{b}_i \, : \, x_i \in \mathbb{Z}$$

[4]Suppose $n = pq$ from the previous section, then a chosen integer $a$ would be $1 < a < n$

$$= \mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n$$

Here **B** denotes a Matrix full of the basis vectors and if the matrix is full rank (or all the basis vectors are linearly independent) then **B** is a basis.

The **Minimum Distance** is the *"shortest distance between two points in a lattice* [29]*."* The **0** vector is also a point in the lattice so the definition can be further reduced to the norm[5] of each basis vector:

$$\boxed{\lambda(L) = min(||\mathbf{b}|| : \mathbf{b} \in L(\mathbf{b} \neq \mathbf{0}))}$$

### 2.3.2 Hardness of Lattices

Due to the mathematical fundamentals discussed in 2.3.1, Lattices exhibit many properties that are conjectured as hard to solve [30] even by **quantum computers**. There are three such properties [28] presented in this section, namely:

- The Shortest Vector Problem (SVP).
- The Approximate Shortest Vector Problem.
- The Shortest Independent Vector Problem (SIVP$_\gamma$).

**The Shortest Vector Problem** $\lambda(L)$ denotes the **Minimum Distance** function defined in 2.3.1. The Shortest Vector Problem is:

If $L$ has a basis, find $\mathbf{b} \in L$ (where $\mathbf{b} \neq \mathbf{0}$) such that:

$$\boxed{||\mathbf{b}|| = \lambda(L)}$$

**Approximate Shortest Vector Problem** The parameter $\gamma$ represents an approximation factor (where $\gamma \geq 1$) that further bounds the Shortest Vector Problem to help obtain an approximation on it as:

If $L$ has a basis, find $\mathbf{b} \in L$ (where $\mathbf{b} \neq \mathbf{0}$) such that:

$$\boxed{||\mathbf{b}|| \leq \gamma.\lambda(L)}$$

**The Shortest Independent Vectors Problem** The **Minimum Distance** function can further incorporate successive minima ($1 \leq i \leq n$) as $\lambda_i(L) = \min(\max_{1 \leq k \leq i}||\mathbf{v}_k||)$ where $\mathbf{v}_1 ... \mathbf{v}_i$ are linearly independent vectors in $L$. The Shortest Independent Vectors problem is:

If $L$ has a basis, find $\mathbf{b}_1, ..., \mathbf{b}_n \in L$ linearly independent vectors such that:

$$\boxed{max_{1 \leq k \leq i}||\mathbf{b}_k|| = \gamma.\lambda_i(L)}$$

With $\gamma = O(1)$ the SIVP$_\gamma$ is NP-hard[6].

[5]This is the euclidean norm of a vector.

[6]Non-deterministic polynomial hardness represents a *"complexity class of decision problems that are harder than problems that can be solved in polynomial time by a non-deterministic Turing Machine"* [31]

### 2.3.3 The Learning with Errors Problem

One of the hardest machine learning problems presented in modern times is the Learning with Errors problem proposed by Oded Regev in 2005 [32]. The mathematical principles of Regev's learning with errors problem are difficult to solve for worst case *generic* lattices. There are numerous interpretations of the learning with errors problem but the LWE definition in the *"Simple Provably Secure Key Exchange based on the Learning with Errors Problem"* by Jintai et al [4] provides a simplified version to help grasp its application.

**Public Parameters** The Public Parameters are:

- A uniformly random matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$
- A vector $\mathbf{s} \in \mathbb{Z}_q^n$
- An error vector $\mathbf{e} \in \mathbb{Z}^m$

By using these public parameters the LWE sample vector $\mathbf{u}$ (where $\mathbf{u} \in \mathbb{Z}_q^m$) can be computed as $\mathbf{u} = \mathbf{A}^T\mathbf{s} + \mathbf{e}$.

**Encryption & Decryption using LWE** Standard cryptographic messages are in the form of a binary string where $\mathbf{x} \in \{0,1\}^m$.

To **encrypt** the user will compute $\mathbf{c}_1 = \mathbf{A}\mathbf{x} \bmod q$, $c_2 = <\mathbf{u}{,}\mathbf{x}>$[7] $+ m.[\frac{q}{2}] \bmod q$.

To **decrypt** the user will compute $c_2 - \mathbf{s}^T\mathbf{c}_1 \bmod q$ which removes $\mathbf{s}^T\mathbf{A}\mathbf{x}$ and leaves $\mathbf{e}.\mathbf{x}$. The message can then be recovered from the "error."

### 2.3.4 Ring Learning with Errors

The Ring Learning with Errors problem is a specialized version of the original LWE problem [33]. It makes use of polynomial rings over finite fields to help make the original LWE problem more efficient. The R-LWE problem is hard to solve for worst-case problems in *ideal* lattices but not *generic* lattices [33].

## 2.4 Post-Quantum Key Exchanges

To appropriately cater to the demands of the post quantum era of computing, cryptographers, researchers and academics alike have proposed new Post Quantum Key Exchanges. This section will provide a short discussion on each of these implementations, detailing each implementation with a short presentation on their theory, security and (in some cases) the key exchange protocol.

### 2.4.1 Lattice Cryptography for the Internet

Lattice Cryptography for the Internet [34] presents lattice-based protocols which can be used for tasks like encryption, key encapsulation and authenticated key exchange (AKE). The structural layout of the implementation includes a reconciliation mechanism

[7] $<\mathbf{u}{,}\mathbf{x}>$ denotes the concatenation of vectors $\mathbf{u}$ and $\mathbf{v}$

which is quite important for the efficiency of this scheme, followed by a passively[8] secure KEM (Key Encapsulation Mechanism) and then an actively secure KEM using the Fujisaki-Okamoto Transformation [35]. A real world application of this KEM is also presented where an AKE protocol is instantiated with the passively secure KEM established in the paper.

This implementation is mostly focused on providing lattice-cryptography methods for *real world networks.* The sections below will provide a short overview on the performance and security that directly result from the outcomes of the KEM schemes proposed:

**Passively Secure KEM** The Passively Secure KEM (referred to as KEM1) is an important component used for AKE and also for the Actively Secure KEM. This subsection will examine the performance and security aspects of KEM1.

**Performance** KEM1 is closely related to [36] but is different in the following aspects:

- Avoids the use of the "codifferent" ideal $R^\vee$
- Makes use of the *"Reconciliation Mechanism"* which (almost) **halves the ciphertext length**. More specifically the ciphertext length reduces from $2n \log q$ to $n(1+\log q)$

The KEM1 scheme suffers a **performance downgrade** by introducing some **security loss** because the RLWE error rate is half as large as prior schemes but the improvement in ciphertext length outweighs this security loss.

**Security** KEM1 is considered *IND-CPA* [37] secure based on the hardness of the R-DLWE$_{q,\mathcal{X}}$ (Ring-Decision Learning with Errors) given two samples. The protocol enjoys provable security by providing computationally indistinguishable "games" based on the R-DLWE$_{q,\mathcal{X}}$ hardness assumption.

**Actively Secure KEM** KEM1 is transformed into an Actively Secure KEM (referred to as PKC2) reliant on an asymmetric encryption scheme (PKC), a cryptographic PRG[9] (Pseudo-random number generator) and hash functions which are modeled as independent random-oracles [38].

**Performance** The best methodology to achieve an efficient KEM results from the Fujisaki-Okamoto transformation on KEM1 which is motivated by the minimally small plaintext space of KEM1. This results in a *hybrid* encryption scheme in which PKC2 symmetrically encrypts a plaintext of arbitrary length. The performance is compromised by the utilization of the random-oracle models output as the *randomness* for asymmetric encryption and the decryption algorithm runs the encryption algorithm again because it needs to validate the ciphertext produced. By using the random oracles output as a seed for a PRG the efficiency compromise made is rectified in PKC2.

[8]Passively secure KEMs are safe from passive adversaries who see public keys and ciphertexts but do not create any of their own

[9]A cryptographic PRG: $\{0,1\}^l \rightarrow \{0,1\}^l$ for some $l$ (the seed length).

**Security** PKC2 is *IND-CCA* [37] secure assuming that PKC is *IND-CCA* secure, the PRG is a secure random number generator and the hash functions are modeled as random-oracles.

### 2.4.2 A Simple Provably Secure Key Exchange Based on the Learning with Errors Problem

By making use of the LWE/RLWE problem this PQC implementation [4] generalizes the Diffie Hellman problem to a "Diffie Hellman Problem with errors." There are two implementations given in this paper with a further proposed implementation. By using the commutativity of computing a bilinear form in two ways that comes from the associativity of matrix multiplications, the key exchange mimics properties from the Diffie Hellman key exchange:

$$\textbf{The D-H Key Exchange}: g^{ab} = (g^a)^b = (g^b)^a$$

$$\textbf{Matrix associativity}: \mathbf{x}^T\mathbf{M}\mathbf{y} = (\mathbf{x}^T\mathbf{M})\mathbf{y} = \mathbf{x}^T(\mathbf{M}\mathbf{y})$$

The section below will discuss **Robust Extractors** that form an important part of generalizing the values computed during the Key Exchange to agreeable values between Alice and Bob. Then the remaining sections will focus on the Key Exchange using LWE and the security and performance of the LWE and RLWE key exchange protocols. There is also a section entitled Interactive Multiparity Key Exchange Protocol which shows Multiparity Key exchange based on the RLWE problem but it is an open problem and will not be discussed in the following sections.

**Robust Extractors** By making use of a hint algorithm two parties that have never met are able to extract identical information. In the scope of this paper Robust Extractors are defined as:

- A deterministic algorithm that takes as input an $x \in \mathbb{Z}_q$ and a signal $\sigma \in \{0, 1\}$ and outputs $k = E(x, \sigma) \in \{0, 1\}$ where $E$ denotes the deterministic algorithm.
- A hint algorithm denoted by $S$ which takes as input $y \in \mathbb{Z}_q$ and outputs a signal $\sigma \leftarrow S(y) \in \{0, 1\}$

There is a condition placed on the robust extractor that must hold true for it to be used:

*For any $x, y \in \mathbb{Z}_q$ such that $x - y$ is even and $|x - y| \leq \delta$ it holds that $E(x, \sigma) = E(y, \sigma)$ where $\sigma \leftarrow S(y)$.*

The term $\delta$ denotes an error tolerance which is given as $\frac{q}{4} - 2$ where $q > 2$ is prime. The robust extractor used for the key exchange scheme is:

$$E(x, \sigma) = (x + \sigma.\tfrac{q-1}{2} \text{ mod q}) \text{ mod } 2$$

## Learning With Errors

### Key Exchange

- First the system generates the public parameters $q, n, \alpha$ where $q > 2$ is prime. The public matrix $\mathbf{M} \leftarrow \mathbb{Z}_q^{n\times n}$ is uniformly sampled.
- Alice selects a secret vector $\mathbf{s}_A \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$ [10] and finds $\mathbf{p}_A = \mathbf{M}\mathbf{s}_A + 2\ \mathbf{e}_A \bmod q$ where $\mathbf{e}_A \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$. She then sends $\mathbf{p}_A$ to Bob.
- After receiving $\mathbf{p}_A$ Bob chooses a secret vector $\mathbf{s}_B \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$ and an error (scalar) $\text{e'}_B \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$. Computes $K_B = \mathbf{p}_A^T.\ \mathbf{s}_B + 2\text{e'}_B \bmod q$. Now Bob will send his LWE sample to Alice by first sampling $\mathbf{e}_B \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$ and then sending $\mathbf{p}_B = \mathbf{M}^T\mathbf{s}_B + 2\ \mathbf{e}_B \bmod q$. By using the definition of the hint algorithm in the Robust Extractor section, he finds $\sigma \leftarrow S(K_B)$ and obtains his shared key as $SK_B = E(K_B, \sigma)$ through the robust extractor given in the previous section. Bob now sends $(\mathbf{p}_B, \sigma)$ to Alice.
- Alice samples an error (scalar) $\text{e'}_A \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$ and finds $K_A = \mathbf{s}_A^T.\ \mathbf{p}_B + 2\text{e'}_A \bmod q$ and obtains $SK_A = E(K_A, \sigma)$ using the same signal $(\sigma)$ as Bob.

If Alice & Bob ran the protocol honestly then they will share an identical key.

**Performance** The performance given is estimated for 1-bit secret keys:

| | Public Param. | Commun. Complexity | Comput. Comp. | Assumption |
|---|---|---|---|---|
| Reg'05 | $4(n+1)n\ \log^2 q$ | $4(n^2+6n)\ \log^2 q + 2\ \log q$ | $4n^2\ \log q$ | $\text{SIVP}_{\mathcal{O}_{(n^3)}}$ |
| LP'11 | $4n^2\ \log q$ | $4(n^2+n)\ \log q$ | $6n^2$ | $\text{SIVP}_{\mathcal{O}_{(n^3)}}$ |
| Jintai'17 | $n^2\ \log q$ | $2n\ \log q + 1$ | $2n^2$ | $\text{SIVP}_{\mathcal{O}_{(n^4)}}$ |

Table 2: Public Parameter corresponds to the size of the public parameter, Computational complexity is estimated through the number of operations in $\mathbb{Z}_q$

The communicational complexity is greatly reduced in this scheme due to the one-time generation of the public parameter $\mathbf{M}$. Other Public Key Encryption schemes which rely on key transport schemes often rely on the shared key determined by one party but this protocol allows the key to be determined by both parties. This protocol also fares better in Key Encapsulation Mechanisms as the communicational and computational complexity are quite efficient.

**Security** The protocol is considered provably secure on the HNF-LWE[11] assumption . The assumption is based on a theorem that states *The LWE problem is no easier if the secret vectors are sampled from an error distribution* [39]. Here the error distribution is the Discrete Gaussian Distribution in the Key Exchange section.

[10] $\mathcal{D}_{\mathbb{Z}^n,\alpha q}$ represents a function which generates a vector of $n$ integers sampled from the Discrete Gaussian Distribution with mean 0 and standard deviation $\alpha q$.

[11] Hermite normal form of Learning with Errors

**Ring Learning with Errors**

**Performance**

| | Public Param. | Commun. Complexity | Comput. Comp. | Assumption |
|---|---|---|---|---|
| LPR'10 | $4n \log q$ | $8n \log q$ | 6 | Ideal-SIVP$_{\mathcal{O}_{(n^3)}}$ |
| SS'11 | $2n \log q$ | $4n \log q$ | 4 | Ideal-SIVP$_{\mathcal{O}_{(n^3)}}$ |
| Jintati'17 | $n \log q$ | $2n \log q + n$ | 4 | Ideal-SIVP$_{\mathcal{O}_{(n^4)}}$ |

Table 3: Public Parameter corresponds to the size of the public parameter, Computational complexity is estimated through the number of operations in the ring $R_q$

Compared to Public Key Encryption with RLWE this scheme is more efficient as the public parameter $m$ (in the key exchange) is generated once which reduces the communication cost. A direct comparison with SS'11 showcases a worse computational cost but the SS'11 scheme is more inefficient than this protocol as a high security parameter is needed for practical applications.

**Security** The RLWE Key Exchange is considered secure based on the same assumptions presented in the LWE Key Exchange.

### 2.4.3 Frodo: Take off the ring! Practical, Quantum-Secure Key Exchange from LWE

Many PQC implementations make use of the RLWE problem based on the efficiency that is attained due to the additional ring structure but Frodo [2] makes use of the Learning with Errors Problem to generate a pragmatic and efficient key exchange scheme. Frodo proves its security by using fundamentals from the original LWE problem. Frodo incorporates several innovation that set it apart from other PQC implementations, namely:

- Efficiently sampleable noise distribution:

  The Discrete Gaussian distribution is replaced by a custom discrete noise distribution due to the analysis of the Rényi Divergence [40] on rounding continuous Gaussian Distributions.

- Efficient and dynamic generation of public parameters:

  Instead of transmitting the large public parameter **A** the protocol transmits the seed. It is still challenging to efficiently generate **A** but the protocol dynamically generates it to allow usage at run time and discards it if it is unnecessary.

Frodo is available as an open source implementation[12] in the C language and has already been integrated in OpenSSL to be used with Apache.

The TLS message flow can be summarized in the following figure:

[12]The link can be found here: https://github.com/lwe-frodo/lwe-frodo

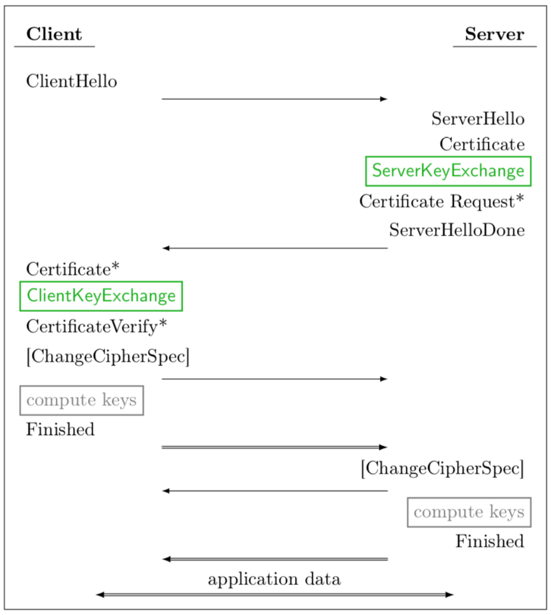


Figure 2: Transport Layer Security Message Flow [2]

Furthermore, Frodo is deployed with elliptic curve Diffie-Hellman as part of a *hybrid ciphersuite* as the message flow of the two protocols is identical.

**Performance** Frodo is evaluated on the basis of the following metrics:

- Speed of standalone cryptographic operations.
- Speed of HTTPS connections.
- Communication costs.

The hardware used for standalone cryptographic operations and acting as a server for TLS connections was an **n1-standard-4** Google Cloud VM instance with 15 GB of memory, which has 4 virtual CPUs. Clients for measuring the throughput of TLS connections were run on an **n1-standard-32** Google Cloud VM instance with 120 GB of memory and 32 virtual CPUs, which ensured that the server could be fully loaded.

The following sections summarize the performance of Frodo for Standalone Cryptographic Operations and HTTPS connections:

**Standalone Cryptographic Operations** The following table showcases Frodos performance in terms of standalone cryptographic operations using the **ideal** and **paranoid** parameters with **quantum** security (in bits), messages sent from Alice to Bob (**A** $\rightarrow$ **B**) in communication and **Alice0** denotes Alice's initial ephemeral key and message generation (e.g Alice's operations up to and including her message transmission); **Bob** denotes Bob's full execution (including ephemeral key and message generation and key derivation) and **Alice1** denotes Alice's final key derivation.

| **Scheme** | **Alice0** (ms) | **Bob** (ms) | **Alice1** (ms) | **Communication (in bytes)** | **Claimed Security (in bits)** |
|---|---|---|---|---|---|
| Recomm. | **1.13**±0.09 | **1.34**±0.02 | **0.13**±0.01 | **11296** | **130** |
| Paranoid | 1.25 ± 0.02 | 1.64±0.03 | 0.15 ± 0.01 | 12976 | 161 |

Table 4: Performance of Standalone Cryptographic Operations in Frodo with Standard Deviations. [2]

The runtime of Frodo is orders of magnitude faster than SIDH (Supersingular isogeny Diffie-Hellman) [41], faster than NTRU [42] (on the server side), about 1.8× slower than ECDH nistp256, and about 9× slower than R-LWE (NewHope).

**HTTPS connections** The HTTPS connections measurements include the standalone Frodo implementation with the recommended parameters and a *hybrid ciphersuite*. The following tables show the number of connections per second and connection time of the two implementations with performance measured using RSA signatures (3072-bit keys) and ECDSA signatures (with **nistp256** keys).

| **Sig** | **Connections/sec** | | | | **Connection time (ms)** | |
|---|---|---|---|---|---|---|
| | **1B** | **1 KiB** | **10 KiB** | **100 KiB** | **w/o load** | **w/load** |
| **ECDSA** | **923**±49 | **892**±59 | **878**±70 | **843**±68 | **18.3**±0.5 | **31.5** |
| **RSA** | **703**±4.2 | **700** ±6.2 | **698** ±1.8 | **635** ±16 | **20.7** ±0.6 | **32.7** |

Table 5: Standalone Frodo with the recommended parameters on an Apache HTTP server. [2]

The Handshake size for the ECDSA signature is **23.725** KB and for the RSA signature it is **24.228** KB. Wireshark was used to measure the time for the client to establish a TLS connection on an otherwise unloaded server, measured from when the client opens the TCP connection to when the client starts to receive the first packet of application data, and the size in bytes of the TLS handshake as observed. Connections were initiated using the `openssl s_client` command.

| **Sig** | **Connections/sec** | | | | **Connection time (ms)** | |
|---|---|---|---|---|---|---|
| | **1B** | **1 KiB** | **10 KiB** | **100 KiB** | **w/o load** | **w/load** |
| **ECDSA** | **735**±12 | **716**±1.5 | **701**±20 | **667**±7 | **22.9**±0.5 | **36.4** |
| **RSA** | **552**±0.5 | **551** ±1.9 | **544** ±1.6 | **516** ±1.8 | **24.5** ±0.3 | **39.9** |

Table 6: Frodo with ECDH in a hybridised ciphersuite on an Apache HTTP server. [2]

The Handshake size for the ECDSA signature is **23.859** KB and for the RSA signature it is **24.439** KB.

**Security** The proof of security in Frodo is provably based on the *decisional* LWE-problem [32] where if an adversary was able to distinguish matrices (instead of vectors due to Frodos nature) from uniformly random samples then the adversary could also solve the *decisional* LWE-problem. As the *decisional* LWE-problem cannot currently be solved by any 'known' quantum algorithm, it is implied as provably secure. TLS security is of paramount importance, a model for TLS which is acceptable would be the *authenticated and confidential channel establishment* (ACCE) model [43]. In the context of Frodo Bos et. al [44] determine that Frodo exhibits **ddh-like** security which automatically allows Frodo to inherit ACCE security. It can also be shown that Frodo is secure under the IND-CCCA [37] model though that would require extra bits and could undermine the efficiency of the protocol.

## 2.5 Transport Layer Security

Transport Layer Security is a cryptographic protocol designed to protect network connections. It was first introduced in 1999 as a significant upgrade to its predecessor SSL [45]. It is used for Web browsers and other applications that require data to be securely exchanged over a network such as file transfers, Virtual Private Network connections and Instant Messaging.

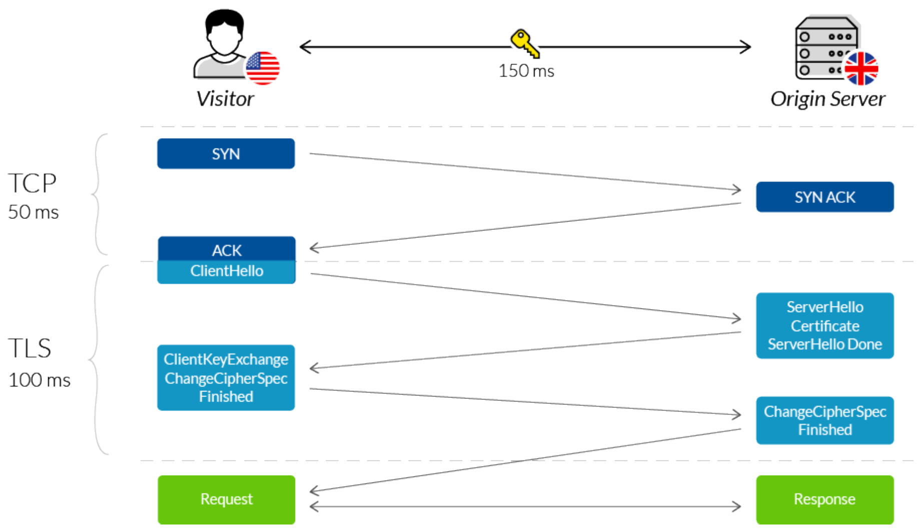


Figure 3: TLS Handshake [3]

The figure above summarises the connection that is experienced by a client (a typical user browsing the internet in a private or even commercial setting) and a server (a website hosted by another computer or large scale network system that has access to domains). The first half of the diagram shows a TCP (Transmission Control Protocol) packet exchange which is a network protocol designed to acknowledge the presence of a server to a client. Then the TLS communication suite starts after the client has received a guarantee that there is indeed a server available to listen to the clients connection but before the TLS protocol is launched this type of communication is insecure as the Client still does not know that they are indeed communicating with the server they asked for. To launch the TLS protocol the client sends the server a `ClientHello` request which sends information about the TLS version and the cryptographic suite used by the client. The server responds with a `ServerHello` which sends a potential cipher-suite chosen from the original `ClientHello`, a session ID, a random byte string and usually a digital certificate.

The client then validates the server's digital certificate. The client then sends a random byte string which is encrypted with the server's public key. The server then validates the client's public key by using a cryptographic algorithm to retrieve the public key. Then both the client and the server respond with a "finished" message indicating the protocol ran successfully [46].

Most web browsers like Google Chrome, Safari and Microsoft Edge make use of TLS 1.3 which is the current cryptographic suite used for securing communication against adversaries over a network. The most pressing area of the TLS protocol involves the public key sent to the server by the client. This area is now vulnerable to quantum computer attacks but by incorporating the PQKEs discussed in section 2.4 into TLS this severe breach of security can be avoided. In order for most secure communications to eventually be interfaced with TLS, it must first be designed in OpenSSL [47]. OpenSSL is an open source software which can simulate TLS connections used in TLS 1.3. It also provides access to source code files that can be modified to include more public-key cryptography algorithms for eventual use in their cipher-suite. Unfortunately, the documentation for adding new algorithms to OpenSSL is amiss and the internal file structure requires large modifications in many files. Fortunately, the Open Quantum Safe project [5] has developed an API which allows developers to wrap their public-key algorithms into OpenSSL to prototype its use in TLS.

## 2.6 Liboqs: The Open Quantum Safe Project

The Open Quantum Safe project [5] has recently created a large cryptographic library of PQKEs. Liboqs [48] currently provides the following algorithms for PQC:

| Name | Frodo'16 | BCNS'15 | NewHope'16 | MSRLN'16 | SIDH'16 |
|---|---|---|---|---|---|
| Type | LWE | RLWE | RLWE | RLWE | Supersingular Isogeny DH |

Table 7: PQKEs in Liboqs

Liboqs comes equipped with a large test-suite that allows a developer to test numerous PQKEs for computation time, communicational complexity and even OpenSSL connections. The following benchmark results from the liboqs library (of standalone cryptographic operations) were tested on a 2.6GHz Intel Xeon E5 (Sandy Bridge) [2]:

| **Scheme** | **Alice0** | **Bob** | **Alice1** | **Communication** | | **Claimed Security** | |
|---|---|---|---|---|---|---|---|
| | (ms) | (ms) | (ms) | **A→B** | **B→A** | **classical** | **quantum** |
| RSA 3072-bit | - | 0.09 | 4.49 | 387/0* | 384 | 128 | - |
| ECDH `nistp256` | 0.37 | 0.70 | 0.33 | 32 | 32 | 128 | - |
| BCNS | 1.01 | 1.59 | 0.17 | 4,096 | 4,224 | 86 | 78 |
| NewHope | 0.11 | 0.16 | 0.03 | 1,824 | 2,048 | 229 | 206 |
| NTRU `EES743EP1` | 2.00 | 0.28 | 0.15 | 1,027 | 1,022 | 256 | 128 |
| Frodo Recomm. | 1.13 | 1.34 | 0.13 | 11,377 | 11,296 | 144 | 130 |
| Frodo Paranoid | 1.25 | 1.64 | 0.15 | 13,057 | 12,976 | 177 | 161 |
| SIDH | 135 | 464 | 301 | 564 | 564 | 192 | 128 |

Table 8: Performance of Standalone Cryptographic Operations: Mean runtime of standalone cryptographic operations. Communication is given in bytes.

The table also includes traditional cryptographic key exchanges like RSA or ECDH which are not quantum resistant as reference. SIDH (Supersingular Isogeny Diffie Hellman) [41] is another PQKE but is not based on Lattice Cryptography primitives, instead it makes use of Elliptic Curve Cryptography. Some of the PQKEs given in the table rely on the AVX2 instruction set (SIDH, to an extent NewHope). but the computer that tested these PQKEs did not have the AVX2 instruction set so the results given in this table are not indicative of its optimal performance. In the table **Alice0** represents all of Alice's operations before she sends public data to **Bob**, **Bob** represents all of his operations up till the generation of the shared key and **Alice1** represents all of Alice's operations after receiving the public data from **Bob** till her shared key.

It is clear from this table that NewHope has the best performance of all of the PQKEs listed. This is partly due to the fact that it is an RLWE key exchange which is orders of magnitude more optimal than an LWE key exchange.

The Liboqs library also provides a common API that allows developers to integrate their PQKEs into OpenSSL to test connection time and handshake size for cryptographic operations on TLS. There are numerous prerequisites for a PQKE that would follow its integration into liboqs:

- The PQKE must be in the C language.
- The PQKE must wrap its core functions showcasing **Alice0**, **Bob** and **Alice1** in wrapper functions in liboqs with the prefix `__OQS`.
- All PQKEs must make use of OpenSSL's random seed.
- Over 6 different files need to be modified to allow a new PQKE to reflect in the build and testing process.

As a side note, this project will aim towards crafting a PQKE which is ready to integrate into Liboqs but will not explore its integration into Liboqs.

# 3 Requirements Capture

The project required the development of a new lattice cryptography based quantum safe key exchange. Based on the relevant material presented in the Background section there are only 2 PQKEs that have not been implemented in programming languages:

1. Lattice Cryptography for the Internet [34].
2. A Simple Provable Secure Key Exchange based on the Learning With Errors Problem [4].

The following points explore the main requirements of this project in further detail:

- The PQKE generates the correct shared keys between the server and the client in a Key Exchange Simulation. This is achieved by optimal parameter selection ensuring a high probability of success (as discussed in section 2.4.2).
- The protocol must implement large key sizes to guarantee security against brute-force attacks.
- The Probability of failure of the key exchange is negligible ($\Pr[\text{Failure}] \leq 2^{-k}$ where $k \geq 30$ in most cases [2]) to guarantee the correctness of the protocol in real world settings. Testing the probability of failure may be impractical with hardware limitations so a suitable threshold value can determine a qualitative successful measurement.
- The implemented PQKE must be designed in such a way that it can be readily interfaced with OpenSSL.
- A direct comparison of the implemented PQKE in this project with already existing PQKEs. This is achieved through the liboqs library [5] as it provides access to a rich API that uses automated shell scripts to test correctness, statistical distance, memory consumption and latency.

While both "Lattice cryptography for the Internet" and "A Simple Provably secure key exchange based on the Learning with Errors Problem" satisfy all of these requirements listed above, the former is mostly focused on RLWE key exchanges while the latter provides an algorithm for a Learning with Errors protocol. Since LWE is secure against *worst case/generic* lattices it is much safer than RLWE protocols but incurs communication cost. It is also important to note that apart from Frodo [2] there aren't any other LWE protocols.

In compliance with these requirements the chosen protocol from section 2 is the "Simple Provably Secure Key Exchange based on the Learning With Errors Problem" (as discussed in section 2.4.2).

As a supplementary milestone, this project explored the integration of a discrete gaussian distribution library [49] and details a step by step guide allowing developers to use discrete gaussians with 5 distinct algorithms in their development and research.

# 4 Analysis and Design

The design for a successful Post-Quantum Key Exchange can be accomplished through the implementation of the PQKE selected in section 3. After this algorithm has been successfully implemented in a programming language it needs to be modified into a multiple-bits algorithm to guarantee security against *brute-force* attacks. It must also be designed in such a way that the PQKE's runtime is fast to accommodate its eventual integration into Internet applications (which would require fast protocols due to the number of clients available connecting to a server).

## 4.1 System Requirements

The following points present a short overview of the requirements needed for a Post-Quantum Key Exchange that could be used over the Internet. Due to time constraints some requirements were unable to be fulfilled and are therefore discussed in this section with some suggestions for implementation and are recommended as future work in the latter half of this report.

- **Ensuring the session keys match:** By designing an extractor, this would guarantee that the session keys are bounded within a certain range and the extracted value is a uniformly random bit.
- **Optimising parameter generation for large sizes:** This is explored through a large variety of test results which include significant trade-offs between time and security.
- **Simplicity and Adaptability:** The source code for the PQKE must be simple and elegant so that anyone can potentially optimise the PQKE further or can integrate it into OpenSSL.

#### Security

The most recognisable issue in the design of a PQKE focuses on its security towards Quantum Computers. Lattice Cryptography offers an interesting solution to PQC as it is provably secure towards Quantum Computers. There exists no quantum algorithm that could solve the *decisional-LWE* problem in polynomial time and the *decisional-LWE* problem is secure against *generic* lattices guaranteeing its security as a worst case possible measure. By utilising the Key Exchange discussed in section 2.4.2 [4] the system designed retains provable security based on the *decisional-LWE* problem ensuring that an adversary is unable to distinguish from samples that *look random.* This provable security is of paramount importance and can only be retained if proper server-client side simulations are generated in a source code file to ensure that further integration of it in network applications can retain secure and specific design features.

## 4.2 Design

Designing the PQKE involved designing separate subcomponents which eventually fed into each other to create a fully optimal multiple-bits protocol which utilises discrete gaussians. This was initially achieved through algorithmic prototyping in Python as the syntax of key exchanges is better understood in pseudo-code. Then working implementations of a modified key exchange (since Python does not support discrete gaussian libraries) were transferred into C where memory consumption and other iterative algorithms (like matrix multiplications) supplemented with discrete gaussians allowed a more efficient protocol.

The system would begin the protocol by generating public parameters which are preprocessed in a header (`.h`) file. Then the system would generate the public matrix **M** once and would use it for subsequent key exchanges. This matrix could be efficiently generated by a trusted third party organisation like NIST but in the current design could be uniformly generated within the source code. The client would construct their public matrix and send it to the server. The server would use that public matrix to generate a 'hint' signal and a session key and transmit their public matrix and hint signal to the client. Both the client and the server would then continue to generate shared session keys which could be reconciled through an extractor. This design guarantees a secure system as an adversary would only have access to the public matrices and the 'hint.' The public matrices are all secure under the *decisional-LWE* problem which relies on the Shortest Independent Vectors Product Problem in Lattice Cryptography. The 'hint' is secure as it is a uniformly random $n$-bit sequence. The current design also unburdens communicational complexity by generating the parameters once before the key exchange is launched.

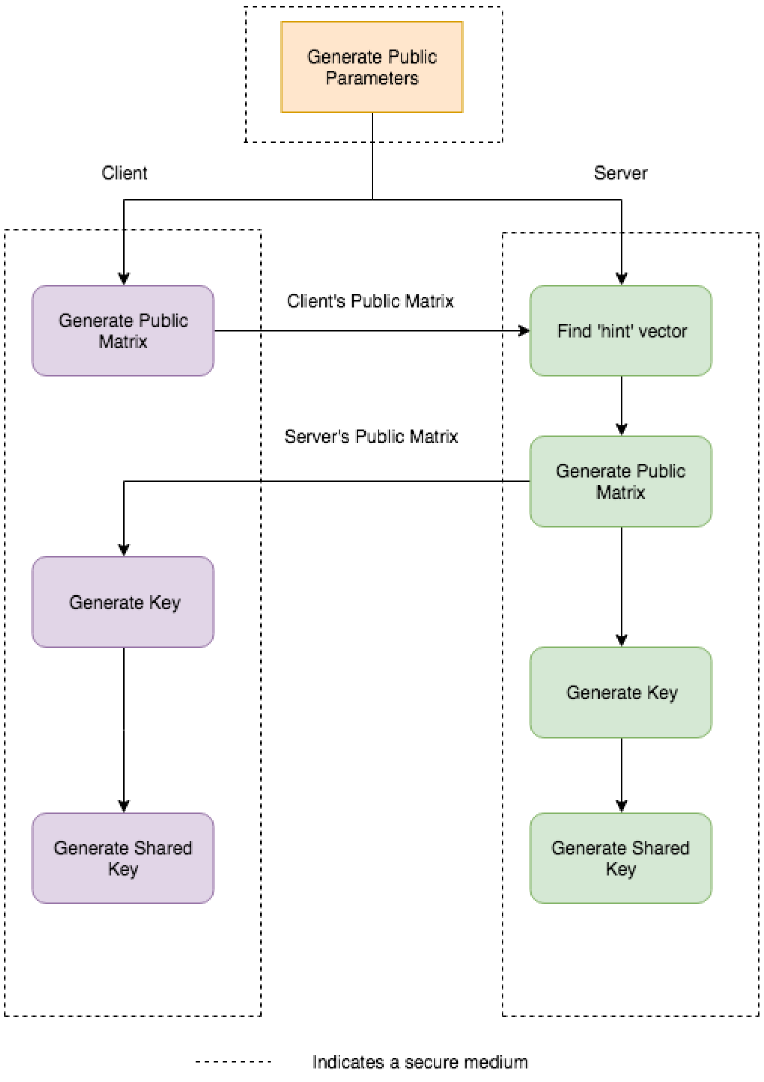


Figure 4: System Design for the PQKE [4].

# 5 Implementation

As detailed in the design section this project was implemented in incremental parts. The PQKE was prototyped in Python initially due to its easy abstraction with functions, ability to generate large integer values and ease in programming. The key exchanges in Python made use of NumPy [50], which is a library specifically designed for matrix operations (allowing explicit programming for Lattice Cryptography). After successful stages of the protocol were working in Python, the implementation was ported to C. Since C allows explicit data type setting and more freedom with memory usage a robust implementation with optimal memory usage and time complexity was realized. Python does not have libraries for Discrete Gaussians which resulted in correctness checks through Continuous distributions rounded off to the nearest integer. This section involves a discussion on the numerous steps taken to implement a Lattice Based Cryptography protocol based on the Learning With Errors problem discussed in section 2.4.2 and focuses on the algorithms implemented in C because the algorithms in Python are more explicit and convey less meaning behind the back-end structure of their operations.

## 5.1 Discrete Gaussian Distributions

A fundamental part of the key exchange involves sampling vectors from Discrete Gaussian Distributions. Currently most programming languages provide no support for Discrete Gaussian distributions. NumPy mostly heralds continuous distributions and C involves using the Box-Mueller method [51] to generate continuous Gaussian distributions. Python does not currently have many libraries or open source programs to generate Discrete Gaussian Distributions so a Continuous Gaussian Distribution which rounded off to the nearest integer value was used. Fortunately, there is an open source implementation available in C [49] which implements 5 algorithms for Discrete Gaussian Distributions.

### 5.1.1 Algorithms in the DGS Library

The DGS library implements the following algorithms:

1. **Uniform Table lookup**: After sampling from a uniform distribution, values are extracted from a table by using the uniform distribution value to perform a table lookup. These values are proportional to $e^{\frac{-(x-c)^2}{2\sigma^2}}$. Since these values are integer values (or scalar ones) it is necessary to iterate over vectors and perform an individual table lookup to generate discretely random vectors.

2. **Uniform Logtable**: After sampling from a uniform distribution, values are accepted with a probability proportional to $e^{\frac{-(x-c)^2}{2\sigma^2}}$. Here $e^{\frac{-(x-c)^2}{2\sigma^2}}$ is computed using logarithmically many calls to Bernoulli distributions.

3. **Uniform Online**: Same as algorithm 1 but instead of pre-computing samples and performing a table lookup, $e^{\frac{-(x-c)^2}{2\sigma^2}}$ is computed in each invocation. This makes this algorithm very slow.

4. **Sigma2 Logtable:** Samples are drawn from $\sigma = k.\sigma_2$ where $\sigma_2 = \sqrt{\frac{1}{2\log 2}}$ and are accepted with a probability proportional to $e^{\frac{-(x-c)^2}{2\sigma^2}}$ where $e^{\frac{-(x-c)^2}{2\sigma^2}}$ is computed

by using logarithmically many calls to Bernoulli Distributions (but no calls to the exponential).

5. **Alias Method**: This method relies on the Alias Method (An efficient table lookup method on arbitrary probability distributions). Table lookup is now just a randomized lookup where set up costs are roughly $\sigma^2$ and table sizes are linear in $\sigma$.

Algorithm 1 is based on classical rejection sampling [52]. Algorithms 2-4 are described in a paper on Lattice Signatures and Biomodial Gaussians [53]. The PQKE makes use of Algorithm 1 for default test settings but other algorithms are evaluated in section 6.3.

### 5.1.2 DGS Library Integration

The DGS library comes equipped with 5 files that are imperative for generating Discrete Gaussian Distributions namely `dgs_bern.c`, `dgs_gauss_dp.c`, `dgs_gauss_mp.c`, `dgs_rround_dp.c` and `dgs_rround_mp.c`. Here the terms 'dp' and 'mp' denote double precision (using machine doubles) and multi-precision using MPFR [54] respectively.

A common way of using other C files and functions in one C file to generate a single executable involves generating Object files with the `.o` extension. The header files (files with the `.h` extension) for these corresponding `.o` files can serve as access points to the functions encapsulated in the `.o` files. This is quite important as the DGS library invokes multiple functions and all of them are accessible through a single initialiser function namely `dgs_disc_gauss_dp_init(`$\sigma$`, `$\mu$`, `$\tau$`, algorithm)`. Here the 'algorithm' parameter corresponds to the algorithms discussed in section 5.1.1. Therefore, in order to generate the corresponding object files the following command can be used:

```
User $> gcc -c dgs_bern.c -I/ -o dgs_bern.o
User $> gcc -c dgs_gauss_dp.c -I/ -o dgs_gauss_dp.o
User $> gcc -c dgs_gauss_mp.c -I/ -o dgs_gauss_mp.o
User $> gcc -c dgs_rround_dp.c -I/ -o dgs_rround_dp.o
User $> gcc -c dgs_rround_mp.c -I/ -o dgs_rround_mp.o
```

Listing 1: Shell commands for Linux and Unix systems to generate the `.o` files from DGS.

The shell commands above assume that all the corresponding header files are in the same directory to invoke the "`/I`" (include) parameter. Once these Object files are generated it is important to link the correct libraries[13] in a Makefile or at compile time to ensure that there are no linker errors. DGS uses 2 external libraries which must be explicitly mentioned at compile time:

- MPFR - The GNU Multi-precision library [54] which deals with generating multiple precision values instead of machine doubles. These are invoked in files which have the "`_mp`" name in their pre-extensions but the library itself is available to all files in the DGS Library.

[13]To add these libraries to your environment a short tutorial is present at https://github.com/Afrazinator/fypafraz/blob/master/README.md

- GMP - Another GNU Multi-precision library [55] which is more focused on fast algorithms for achieving the same results as MPFR. It is mostly used for cryptographic applications and can also be used for generating Discrete Gaussian Distributions. GMP is available to all the files in the DGS Library.

To generate an executable and to link all these Object files, header files (assuming they are all in the same directory) and linker libraries the following command can be used:

```
User $> gcc -o filename filename.o dgs_bern.o dgs_gauss_dp
    .o dgs_gauss_mp.o dgs_rround_dp.o dgs_rround_mp.o -I/
    -lmpfr -lgmp -lm
```

Listing 2: Linux/Unix command to generate an executable "filename" that invokes all the functions from the DGS library.

This command generates an executable `filename` which came from the source file `filename.c` that makes use of all the functions and algorithms from the DGS Library.

### 5.1.3 Generating Discrete Gaussian arrays in C

The C language relies on allocating memory to arrays before utilizing them in arithmetic operations. This is achieved by using the in-built `malloc` function, where each vector is constrained by the size of the Lattice Dimension and each matrix is an $n \times n$ multi-dimensional array where $n$ represents the Lattice Dimension. To make use of the DGS library the `dgs_disc_gauss_dp_init(`$\sigma$`, `$\mu$`, `$\tau$`, algorithm)` function must be invoked and set to a pointer `D`. Then that pointer can return a value through the method "`D` $\rightarrow$ `call(D)`."

**Algorithm 1:** Using DGS in C

```
Result: discrete_val
1 Public Parameters: $\mu = 0$, $\sigma$ = Lattice Dimension, $\tau = 6$, algorithm = Uniform
   Table;
2 dgs_disc_gauss_dp_t *D =
   dgs_disc_gauss_dp_init($\sigma, \mu, \tau$,DGS_GAUSS_UNIFORM_TABLE);
3 */ Some code /*;
4 Function DiscGauss():
5     long discrete_val = D $\rightarrow$ call(D) ;        // The call function returns a long
       integer value
6     return discrete_val
```

Algorithm 1 is encapsulated as a function which can now be invoked as many times as necessary to generate the Discrete Gaussian vectors and matrices required by the PQKE. This can now be demonstrated in Algorithm 2 which also showcases use of the `malloc` function to allocate memory.

**Algorithm 2:** Generating a Discrete Gaussian Vector in C

```
Result: gauss_vec
Public Parameters: $n$ = Lattice Dimension, DiscGauss();
int gauss_vec = (int*)malloc(N*sizeof(int)) ;   // Allocate memory to gauss_vec
*/ Some Code */;
for $i \leftarrow 0$ to $n$ by 1 do
    gauss_vec[i] = DiscGauss() ;   // Set each entry to a discrete random value
end
```

These algorithms can now be used for generating all of the secret, public and error vectors required by the PQKE.

## 5.2 Single-Bit Post-Quantum Key Exchange

Section 2.4.2 explained the construction and the utilization of the key exchange on paper but this section will explore the algorithmic implementation of the key exchange.

### 5.2.1 Public Parameters

There are 6 public parameters that the scheme requires before generation:

1. $n$: The Lattice Dimension. This is also used for the key size.
2. q: A very large prime number for modulo operations and finite field construction.
3. **M**: A public matrix important for secret sharing between Alice and Bob. Here **M** $\in \mathbb{Z}_q^{n \times n}$.
4. $\mu$: The centre for the Discrete Gaussian Distribution.
5. $\alpha q$: The standard deviation for the Discrete Gaussian Distribution[14]. This is equivalent to the Lattice Dimension.
6. $\tau$: The rejection value constricting the Discrete Gaussian to the range $\mu - \sigma\tau$ to $\mu + \sigma\tau$.

The most important public parameter in the entire scheme is the matrix **M**. **M** represents a uniformly random matrix comprised of the integers mod $q$. This implies that:

$$\mathbf{M}[i][j] \xleftarrow{\$} [0, 1, 2, ..., q-1] \text{ where } i, j \in 0, 1, ..., n$$

Python explicitly allows use of the `random.randint()` method which was used to generate **M**. It is also important to generate $\mathbf{M}^T$ which needs to be used for generating Bob's public vector $p_B$.

C requires a nested loop with `rand()` through modular operations to generate the Matrix. The `rand()` function in C returns a pseudo-random number which is uniformly distributed and by using the modulo operator it is possible to generate a finite field. The `rand()` function requires seeding to ensure pseudo-random behaviour. Implementations like Frodo make use of a string of 8 characters but the `srand()` seed used for this Matrix

[14]Here the value of $q$ is not the same as the large prime number q

generation was the system clock allowing a pseudo-random value generated every time. The algorithm which was used to generate the Matrix[15] **M** is given below:

```
Algorithm 3: Generating the matrix M
  Result: The matrix M
1 Public Parameters: n = Lattice Dimension, rand()%q = [0, 1, ... , q-1];
2 for $i \leftarrow 0$ to $n$ by 1 do
3     for $j \leftarrow 0$ to $n$ by 1 do
4         M[i][j] = rand()%q
5     end
6 end
```

The current test-suite makes use of the parameters given in the table below.

| $n$ | q | $\mu$ | $\alpha q$ | $\tau$ |
|---|---|---|---|---|
| 512 | $2^{31}$ - 1 | 0 | 512 | 6 |

Table 9: Public Parameters for Key Construction

The parameter 'q' is usually a very large prime number to prevent brute-force attacks but most PQKEs available like Frodo [2] make use of the $2^{31}$ - 1 value due to the risk of overflow in most programming languages. A trade-off value of $n$ is quite necessary for computation time.

### 5.2.2 Robust Extractor & Signal Functions

The PQKE discussed in section 2.4.2 makes use of 2 special functions. A robust extractor which allows reconciliation between Alice & Bob allowing them to agree upon the same key and a signal function generated from a hint algorithm that forms a part of the robust extractor. This section provides a short summary on these 2 functions and showcases functions that can be used in the key exchange.

**Signal Functions** There are 2 signal functions [4] used for the robust extractor. For a prime number q > 2, $\sigma_0(x)$, $\sigma_1(x)$ from $\mathbb{Z}_q$ to $\{0, 1\}$ the signal functions are defined as:

$$\sigma_0(x) = \begin{cases} 0, & x \in [-\lfloor \frac{q}{4} \rfloor, \lfloor \frac{q}{4} \rfloor] \\ 1, & \text{otherwise.} \end{cases} \quad \sigma_1(x) = \begin{cases} 0, & x \in [-\lfloor \frac{q}{4} \rfloor + 1, \lfloor \frac{q}{4} \rfloor + 1] \\ 1, & \text{otherwise.} \end{cases}$$

$\sigma_b(x)$ is formally denoted as $S(x)$ (the hint algorithm). C makes use of an inline optimisation avoiding conditional `if statements` which saves instruction cycles by using the following algorithm:

```
Algorithm 4: Signal Function
1 Public Parameters: q = $2^{31} - 1$, $b \xleftarrow{\$} \{0, 1\}$ ;
  Result: $\sigma \in \{0, 1\}$
2 Function SignalFunc($x$, $b$):
3     return not (x $\geq$ floor($-\frac{q}{4}$)+$b$ and x $\leq$ floor($\frac{q}{4}$)+$b$) ;    // A statement
      evaluates to 1 if it is true and 0 (false) otherwise
```

[15]Note that $\mathbf{M}^T$ is required for Bob's public vector but swapping the indexes during generation can save memory complexity.

**Robust Extractor** A robust extractor [4] is formally denoted as an algorithm $E$ on $\mathbb{Z}_q$ with an error tolerance of $\delta$ with respect to a hint function $S$ (defined in the signal functions), if the following conditions hold true:

- The deterministic algorithm $E$ takes $x \in \mathbb{Z}_q$ and a signal $\sigma \in \{0, 1\}$ and outputs $k = E(x, \sigma) \in \{0, 1\}$.
- The hint algorithm $S$ takes as input an integer $y \in \mathbb{Z}_q$ and outputs a signal $\sigma \leftarrow S(y) \in \{0, 1\}$. This refers to the signal functions defined in the previous subsection.
- For any $x, y \in \mathbb{Z}_q$ the extractor requires $x - y$ to be even and $|x - y| \leq \delta$ so that $E(x, \sigma) = E(y, \sigma)$ where $\sigma \leftarrow S(y)$ and $\delta = \frac{q}{4} - 2$.

From these conditions the robust extractor used in this scheme is:

$$E(x, \sigma) = (x + \sigma.\frac{q-1}{2} \bmod q) \bmod 2 \tag{1}$$

**Algorithm 5:** Robust Extractor on $K_A$ and $K_B$

```
1 Public Parameters: q = $2^{31} - 1$, x $\in \mathbb{Z}_q$;
  Result: $SK \in \{0, 1\}$
2 Function RobustExtractor($x$, $\sigma$):
3     return ((x + (int64_t)$\sigma * \frac{q-1}{2}$)% q)% 2 ;    // it is necessary to cast to 64
        bits to avoid overflow
```

A common problem with the key exchange is based on condition 3 which requires the robust extractor to be even with respect to 2 different Keys and those values to be absolutely bounded within $\delta$. The sheer randomness of generating these keys does not allow this condition to always hold true which is why it is necessary to have a function in place to determine special case handling to ensure the protocol runs correctly. That is why it is important to have a function that flags the comparison of Alice and Bob's keys:

**Algorithm 6:** Check Robust Extractor on K$_A$ and K$_B$

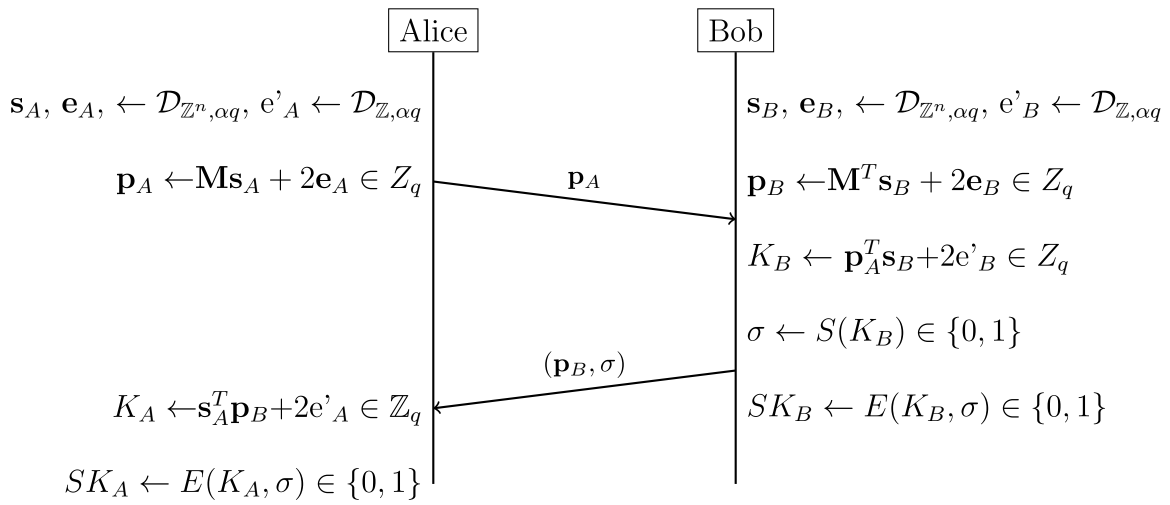
```
1 Public Parameters: q = $2^{31} - 1$;
  Result: True if condition 3 is satisfied, False otherwise
2 Function CheckRobustExtractor($x$, $y$):
3     $\delta = \frac{q}{4}$ - 2;
4     return ((x-y)%2 == 0 and abs(x-y) $\leq \delta$) ;  // This statement returns True if
        it is correct or False otherwise
```

The abs function returns the absolute value of its input parameter. Since the keys generated are discretely random despite their shared secrecy through the original protocol there will be instances when the protocol will not obey this extractor condition. Section 5.2.3 includes a subsection on generating proper keys bounded by these conditions. These algorithms will now be used in the single-bit key exchange and will be modified for the multiple-bits key exchange.

### 5.2.3 Key Exchange

The Key Exchange for this PQKE is based on the same algorithm discussed in section 2.4.2. .

Public Parameters:
q, $n$, $\alpha q$, $\tau$ and $\mu$ from Decision-LWE, $\mathbf{M} \overset{\$}{\leftarrow} [0, 1, 2, ..., \mathrm{q}-1]$


Alice
Bob
$\mathbf{s}_A$, $\mathbf{e}_A$, $\leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$, e'$_A \leftarrow \mathcal{D}_{\mathbb{Z},\alpha q}$
$\mathbf{s}_B$, $\mathbf{e}_B$, $\leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$, e'$_B \leftarrow \mathcal{D}_{\mathbb{Z},\alpha q}$
$\mathbf{p}_A \leftarrow \mathbf{M}\mathbf{s}_A + 2\mathbf{e}_A \in Z_q$
$\mathbf{p}_A$
$\mathbf{p}_B \leftarrow \mathbf{M}^T\mathbf{s}_B + 2\mathbf{e}_B \in Z_q$
$K_B \leftarrow \mathbf{p}_A^T\mathbf{s}_B$+2e'$_B \in Z_q$
$\sigma \leftarrow S(K_B) \in \{0,1\}$
$(\mathbf{p}_B, \sigma)$
$K_A \leftarrow \mathbf{s}_A^T\mathbf{p}_B$+2e'$_A \in \mathbb{Z}_q$
$SK_B \leftarrow E(K_B, \sigma) \in \{0,1\}$
$SK_A \leftarrow E(K_A, \sigma) \in \{0,1\}$

Protocol 2: Single-Bit Post-Quantum Key Exchange

Generating the public vector $\mathbf{p}_A$ is achieved by generating $\mathbf{s}_A$ and $\mathbf{e}_A$ from algorithm 2 and by using $\mathbf{M}$ generated in algorithm 3.

**Algorithm 7:** Generating the public vector $\mathbf{p}_A$

```
Result: $\mathbf{p}_A \in \mathbb{Z}_q^n$
Public Parameters: $n$ = 512, q = $2^{31} - 1$ $\mathbf{s}_A$, $\mathbf{e}_A$ and $\mathbf{M}$;
for $i \leftarrow 0$ to $N$ by 1 do
    for $j \leftarrow 0$ to $N$ by 1 do
        $\mathbf{p}_A$[i] = $\mathbf{p}_A$[i] + ($\mathbf{M}$[i][j]*$\mathbf{s}_A$[j] + 2*$\mathbf{e}_A$[j]) ;    // This value could be negative
    end
    if $\mathbf{p}_A$[i] is negative then
        $\mathbf{p}_A$[i] = $\mathbf{p}_A$[i]%q + q ; // This is the modulo operation for negative numbers
    end
    else
        $\mathbf{p}_A$[i] = $\mathbf{p}_A$[i]%q;
    end
end
```

Algorithm 7 can be reused for generating $\mathbf{p}_B$ by replacing the relevant vectors and by swapping the indexes of the matrix $\mathbf{M}$ so that $\mathbf{M}^T$ may be used.

After Bob receives Alice's public vector $\mathbf{p}_A$ he computes $K_B$ which is needed for the value of $\sigma$.

**Algorithm 8:** Finding Bob's Key $K_B$

```
Result: $K_B \in \mathbb{Z}_q$
Public Parameters: $n$ = 512, q = $2^{31} - 1$ , $\mathbf{p}_A$ and e'$_B$;
temp = 0 ;                                              // temp = $p_A^T.s_B$
for $i \leftarrow 0$ to $N$ by 1 do
    temp = temp + $\mathbf{p}_A$[i]*$\mathbf{s}_B$[i] ;      // $p_A^T$ is not explicit in C, achieved through
                                                           element operation
end
K$_B$ = (temp + 2*e'$_B$) ;
if $K_B$ is negative then
    $K_B$ = $K_B$%q + q ;          // This is the modulo operation for negative numbers
end
else
    $K_B$ = $K_B$%q;
end
```

Now that Bob has $K_B$ he can use the `SignalFunc()` function to obtain a value for $\sigma$.

**Algorithm 9:** Obtaining $\sigma$ from $K_B$

```
Result: $\sigma \in \{0, 1\}$
Public Parameters: $n$ = 512, q = $2^{31} - 1$ ;
Functions: SignalFunc($x, b$) from algorithm 4 ;
$\sigma$ = SignalFunc($K_B$,rand()%2) ;      // Here we reuse the rand() function to
                                           uniformly generate a 0 or a 1
```

After Bob generates $\sigma$ he sends ($\mathbf{p}_B$, $\sigma$) to Alice. Alice takes similar steps indicated in Protocol 2 and by reusing algorithm 8 (with the necessary changes) to generate $K_A$. Then both of them send their keys to an intermediary to run algorithm 6 on $K_A$ and $K_B$ to determine whether they are ready for the extractor. If the value of $K_A$ and $K_B$ with respect to each other is not even or if they are not bounded by the error tolerance $\delta = \frac{q}{4} - 2$ then the following algorithm is utilised to prepare them for the extractor:

**Algorithm 10:** Modifying $K_A$ and $K_B$ for the robust extractor

```
Result: Modified $K_A$ and $K_B$ values
Public Parameters: q = $2^{31} - 1$, $\delta = \frac{q}{4} - 2$;
Functions: DiscGauss() from algorithm 1 ;
if $K_A - K_B$ is not even then
    $K_A = K_A - 1$ ;          // This is permuted between $K_A$ and $K_B$ in the program
end
if abs($K_A - K_B$) is greater than $\delta$ then
    if $K_A > K_B$ then
        $K_B = K_A$ + DiscGauss() ;                        // Add Noise
    end
    else
        $K_A = K_B$ + DiscGauss() ;                        // Add Noise
    end
end
```

Now that $K_A$ and $K_B$ are suitable for the extractor they can both be used in the Robust Extractor function from Algorithm 5.

**Algorithm 11:** Retrieving and comparing Shared Keys

```
Result: $SK_A$ and $SK_B \in 0, 1$
1  Parameters: $K_A$, $K_B \in \mathbb{Z}_q$ and $\sigma \in \{0, 1\}$;
2  Functions: RobustExtractor($K$, $\sigma$) from algorithm 5;
3  $S_{KA}$ = RobustExtractor($K_A$, $\sigma$);
4  $S_{KB}$ = RobustExtractor($K_B$, $\sigma$);
5  if $S_{KA}$ equals $S_{KB}$ then
6      Success, stop the program ; // Stop the program, keys are similar, you can use
         them for encryption
7  end
8  else
9      Fail, run the key exchange again ;                    // Exception handling
10 end
```

If Alice & Bob ran the protocol honestly they share the same key. This Shared Key can now be used between Alice & Bob during ongoing internet connections and for encryption functions.

## 5.3 Multiple-Bits Post-Quantum Key Exchange

The inherent problem with the single-bit key exchange is its lack of security against an attacker. A single-bit can either be a 0 or a 1 and allows a trivial guessing game for the adversary to find the public key and to use it to decrypt ongoing connections. From section 5.2.3 it is clear to see that each secret bit is quantum resistant. As a result it is important to convert this protocol into an implementation with multiple-bits to ensure its hardness as every quantum resistant single-bit generated of an arbitrary length can lead to a randomly large key every time thereby rendering a brute force search obsolete. Naturally, the lattice dimension is used to dictate the key size. All parameters and algorithms mentioned in section 5.2.1 (also including the `DiscGauss()` function from algorithm 1) will be retained for this section.

### 5.3.1 Modifying single-bit parameters & functions

Contrary to Alice's parameters set in section 5.2.3 the new multiple-bits protocol requires the generation of $\mathbf{S}_A \leftarrow \mathcal{D}_{\mathbb{Z}^{n\times n},\alpha q}$ and $\mathbf{E}_A \leftarrow \mathcal{D}_{\mathbb{Z}^{n\times n},\alpha q}$. The other consequent adjustments are now on the scalar values of e'$_A \leftarrow \mathcal{D}_{\mathbb{Z},\alpha q}$ and e'$_{B \leftarrow \mathcal{D}_{\mathbb{Z},\alpha q}}$ which are now transformed into vectors $\mathbf{e'}_A \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$ and $\mathbf{e'}_B \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$. These newly modified parameters allow the generation of $\boldsymbol{K}_A$ and $\boldsymbol{K}_B$[16] $\in \mathbb{Z}_q$ and consequently $\boldsymbol{SK}_A$ and $\boldsymbol{SK}_B \in \{0, 1\}^n$.

We can quantify these changes as:

- $\mathbf{s}_A$ changed from an $n \times 1$ to $\mathbf{S}_A$ an $n \times n$ matrix.
- $\mathbf{e}_A$ changed from an $n \times 1$ to $\mathbf{E}_A$ an $n \times n$ matrix.

[16]Here a bold capital faced letter denotes a vector ( coupled with $\boldsymbol{SK}_A$ and $\boldsymbol{SK}_B$). This will be the only exception of notation in this document.

- The small errors added for $K_A$ and $K_B$ went from scalars to $n \times 1$ vectors.

This modification is intuitively decomposing the original single-bit exchange into a row-wise matrix operation amongst the secret vector $\mathbf{s}_A$ and the error vector $\mathbf{e}_A$. It is better understood visually through the following matrix operations:

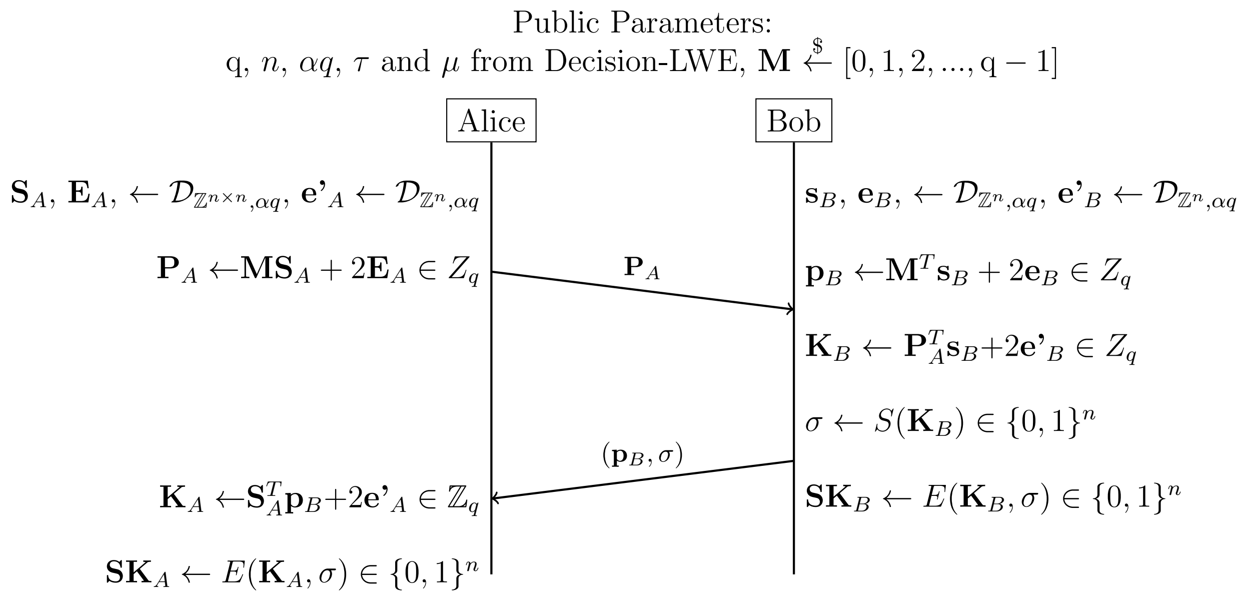
$$\begin{pmatrix} p^1_{A(1)} & \cdot & \cdot & p^1_{A(n)} \\ \cdot & \cdot & & \cdot \\ \cdot & & \cdot & \cdot \\ p^n_{A(1)} & \cdot & \cdot & p^n_{A(n)} \end{pmatrix} = (\mathbf{M} \begin{pmatrix} s^1_{A(1)} & \cdot & \cdot & s^1_{A(n)} \\ \cdot & \cdot & & \cdot \\ \cdot & & \cdot & \cdot \\ s^n_{A(1)} & \cdot & \cdot & s^n_{A(n)} \end{pmatrix} + 2 \begin{pmatrix} e^1_{A(1)} & \cdot & \cdot & e^1_{A(n)} \\ \cdot & \cdot & & \cdot \\ \cdot & & \cdot & \cdot \\ e^n_{A(1)} & \cdot & \cdot & e^n_{A(n)} \end{pmatrix}) \text{mod q} \qquad (2)$$

Equation 2: Decomposing Matrices for $\mathbf{P}_A$ in Multiple-bits exchange.

From the equation above each row of $\mathbf{S}_A$ is a different $\mathbf{s}_A$ vector which is the same case with $\mathbf{e}_A$. This implies that each row of the public matrix $\mathbf{P}_A$ is the same as the public vector $\mathbf{p}_A$ from the single-bit exchange but unique to its corresponding secret vector and error vector. This directly means that the same single-bit key exchange is now being applied $n$ times which is what is required for a multiple-bits protocol. Furthermore, since each single-bit that is generated is quantum-resistant and is uniformly random the entire Shared Key vector would correspond to a large random number which cannot be brute-force searched and is also quantum-resistant.

The functions defined in section 5.2.2 only work on single integer values and on a single $\sigma \in \{0, 1\}$ instead of vectors or the newly defined $\sigma \in \{0, 1\}^n$. Instead of modifying those functions they can be reused in an iterative loop $n$-times to generate the required values of the key exchange.

### 5.3.2 Key Exchange


Public Parameters:
q, $n$, $\alpha q$, $\tau$ and $\mu$ from Decision-LWE, $\mathbf{M} \overset{\$}{\leftarrow} [0, 1, 2, ..., \text{q} - 1]$
Alice
Bob
$\mathbf{S}_A$, $\mathbf{E}_A$, $\leftarrow \mathcal{D}_{\mathbb{Z}^{n\times n},\alpha q}$, $\mathbf{e'}_A \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$
$\mathbf{s}_B$, $\mathbf{e}_B$, $\leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$, $\mathbf{e'}_B \leftarrow \mathcal{D}_{\mathbb{Z}^n,\alpha q}$
$\mathbf{P}_A \leftarrow \mathbf{M}\mathbf{S}_A + 2\mathbf{E}_A \in Z_q$
$\mathbf{P}_A$
$\mathbf{p}_B \leftarrow \mathbf{M}^T\mathbf{s}_B + 2\mathbf{e}_B \in Z_q$
$\mathbf{K}_B \leftarrow \mathbf{P}^T_A\mathbf{s}_B + 2\mathbf{e'}_B \in Z_q$
$\sigma \leftarrow S(\mathbf{K}_B) \in \{0, 1\}^n$
$(\mathbf{p}_B, \sigma)$
$\mathbf{K}_A \leftarrow \mathbf{S}^T_A\mathbf{p}_B + 2\mathbf{e'}_A \in \mathbb{Z}_q$
$\mathbf{SK}_B \leftarrow E(\mathbf{K}_B, \sigma) \in \{0, 1\}^n$
$\mathbf{SK}_A \leftarrow E(\mathbf{K}_A, \sigma) \in \{0, 1\}^n$

Protocol 3: Multiple-Bits Post-Quantum Key Exchange

The multiple-bits key exchange generates shared keys according to the same principles and similar algorithms as its single-bit counterpart. Noticeable algorithmic changes include:

- Generating Alice's secret matrix $\mathbf{S}_A$ and her error matrix $\mathbf{E}_A$ (requires modifying the algorithm 2).
- Generating Alice's public matrix $\mathbf{P}_A$ (similar to algorithm 7).
- Generating $\boldsymbol{K}_A$ and $\boldsymbol{K}_B$ (similar to algorithm 8).
- Finding the signal $\sigma$ (which is now $\sigma \in \{0,1\}^n$).
- Finding the shared keys $\boldsymbol{SK}_A$ and $\boldsymbol{SK}_B$ by using the Robust Extractor.

Generating the secret matrix $\mathbf{S}_A$ can be achieved by a modification on algorithm 2 where instead of generating a discrete gaussian vector, a nested loop can generate a discrete gaussian matrix. There will also need to be memory allocation done in 2 dimensions. This can be achieved by declaring $\mathbf{S}_A$ as a 2-dimensional pointer and allocating memory to the first dimension by using a traditional `malloc` line followed by a loop to allocate memory to the inner-dimension (This approach can allocate memory to any degree of multi-dimensional arrays but C is limited by the amount of memory allocated to each variable so memory cleanup or limited memory consumption is important). It is also important to note that the memory allocated to all of the variables declared in both these key exchanges is necessary and no extraneous memory is consumed because this type of allocation is not dynamic.

**Algorithm 12:** Generating a Discrete Gaussian Matrix in C

```
Result: gauss_matrix
Public Parameters: $n = 512$, DiscGauss();
int gauss_matrix = (int**)malloc(N*sizeof(int*)) ;   // Allocate memory to gauss_matrix in 1 dimension
for $i \leftarrow 0$ to $n$ by 1 do
    int gauss_matrix[i] = (int*)malloc(N*sizeof(int)) ;   // Allocate memory to the 2nd dimension
end
*/ Some Code */;
Function GaussMatrix():
    for $i \leftarrow 0$ to $n$ by 1 do
        for $j \leftarrow 0$ to $n$ by 1 do
            gauss_matrix[i][j] = DiscGauss() ;   // Set each entry to a discrete random value
        end
    end
    return gauss_matrix
```

This will worsen the time complexity of the algorithm from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$. This time complexity issue only applies to the generation of the secret matrix and the error matrix, whereas the generation of the public matrix remains unhinged. Alice's public matrix will instead be worse in terms of space complexity as it was originally a public vector.

Generating the public matrix can be encapsulated in a nested loop with the correct index manipulations:

**Algorithm 13:** Generating the public matrix $\mathbf{P}_A$

```
Result: $\mathbf{P}_A \in \mathbb{Z}_q^{n\times n}$
Public Parameters: $n = 512$, q $= 2^{31} - 1$ $\mathbf{S}_A$, $\mathbf{E}_A$ and $\mathbf{M}$;
for $i \leftarrow 0$ to $N$ by 1 do
    for $j \leftarrow 0$ to $N$ by 1 do
        $\mathbf{P}_A$[i][j] = $\mathbf{P}_A$[i][j] + ($\mathbf{M}$[i][j]*$\mathbf{S}_A$[i][j] + 2*$\mathbf{E}_A$[i][j]) ;   // This value could be negative
        if $\boldsymbol{P}_A$[i][j] is negative then
            $\mathbf{P}_A$[i][j] = $\mathbf{P}_A$[i][j]%q + q ; // This is the modulo operation for negative numbers
        end
        else
            $\mathbf{P}_A$[i][j] = $\mathbf{P}_A$[i][j]%q;
        end
    end
end
```

Algorithm 13 is a good example of the necessary modifications required for the remaining algorithms to generate an $n$-bit key. Similar modifications can be made to algorithm 8 to generate $\boldsymbol{K}_A$ and $\boldsymbol{K}_B$ and algorithm 11 to generate $\boldsymbol{SK}_A$ and $\boldsymbol{SK}_B$:

**Algorithm 14:** Generating $\boldsymbol{K}_A$ and $\boldsymbol{SK}_A$

```
Result: $\boldsymbol{K}_A \in \mathbb{Z}_q^n$, $\boldsymbol{SK}_A \in \{0,1\}^n$
Public Parameters: $n = 512$, q $= 2^{31} - 1$ and $\sigma \in \{0,1\}^n$;
for $i \leftarrow 0$ to $N$ by 1 do
    for $j \leftarrow 0$ to $N$ by 1 do
        $\mathbf{K}_A$[i] = $\mathbf{K}_A$[i] + ($\mathbf{S}_A$[j][i]*$\mathbf{p}_B$[j] + 2e'$_A$[j])%q;   // This value could be negative
    end
    if $\boldsymbol{K}_A$[i] is negative then
        $\mathbf{K}_A$[i] = $\mathbf{K}_A$[i]%q + q ;   // This is the modulo operation for negative numbers
    end
    else
        $\mathbf{K}_A$[i] = $\mathbf{K}_A$[i]%q;
    end
    SK$_A$[i] = RobustExtractor($\mathbf{K}_A$[i],$\sigma$[i]) ;   // RobustExtractor(x,$\sigma$) from Algorithm 5
end
```

Here the CheckRobustExtractor(K$_A$,K$_B$) function is omitted for clarity but is part of the actual implementation. In fact the function is necessary to produce the correct Keys before the extractor can guarantee the correct value. If Alice & Bob run this protocol they will share an identical $n$-bit key which can now be used for encryption or authentication in Internet Communication applications.

# 6 Testing and Results

This section provides a detailed analysis on the tests performed on the PQKE. The section begins by exploring important cryptographic metrics that are essential to assess the performance of the PQKE. It then provides a short description of the testing environment to ensure impartial results. Then it details the numerous tests conducted on the test environment for each language used during the implementation of this PQKE. Finally, it provides a comparison of this PQKE with other already available Cryptographic Protocols available under the Open Quantum Safe project [5].

## 6.1 Important Cryptographic Metrics

The Institute for Defense Analyses Science and Technology Division published a paper in 1997 [56] detailing the importance and relevance of cryptographic metrics that could be used to assess the performance of a key exchange. Select cryptographic metrics that could quantify the optimality of the key exchange are:

1. **Runtime**: This is indicative of the speed of the protocol (time taken to run a successful key exchange). This is limited by Algorithmic Complexity.
2. **Communicational Complexity**: The traditional definition of communicational complexity based on matrices (where if an $n \times m$ matrix multiplies an $m \times p$ matrix it incurs $nmp$ multiplications and $\mathcal{O}(nmp)$ additions [2]) will be used. C allows manipulation of the `malloc` function and online tool `Valgrind` to measure memory consumption.
3. **Throughput**: Number of successful key exchanges run in a set amount of time.

Other cryptographic metrics like type/function are trivial as there is only one PQKE implemented. Strength and Security have already been discussed in sections 2 and 5 so they will be discarded for this section.

## 6.2 Test Environment

There are many available Cloud Computing resources that can be used to provide an unprejudiced result on the testing required for the PQKE. AWS was chosen as the cloud computing back-end service with an Intel Xeon E5 (Sandy Bridge). This is available on AWS as an **m4.xlarge** that has 4 virtual CPUs and 16GB of Memory. This represents a general purpose computer which makes it ideal for Cryptographic Applications. The instance also comes loaded with 4 cores which could serve as a reference point for parallel optimisations (though these were not explored in this report).

The **m4.xlarge** instance comes loaded with a customised AMI (Amazon Machine Image) on Ubuntu 16.04 which runs `gcc` compilers, allows package installation and the ability to run Python scripts with NumPy. It also comes with automake allowing build processing through Makefiles.

## 6.3 Results

This section details all of the results for the PQKE with different varying sub factors in each subsection. Consequently, tests will be carried out on the basis of:

- Programming Language (C or Python), this will incorporate all the metrics from section 6.1
- Lattice Dimension vs Runtime.
- DGS algorithms[17] vs Runtime.

For the following results the values for parameters discussed in section 5.2.1 were used unless stated otherwise. The Multiple-Bits Protocol will be the only one discussed in this test-suite as the single-bit protocol is impractical in real-world applications.

### 6.3.1 Programming Language

This section will discuss all of the cryptographic metrics in the previous section for both C and Python.

**Runtime** Since this Key Exchange was implemented in 2 programming languages it is useful to understand the performance of individual parameters in both and to try and analyse why certain operations perform faster in C or Python. C is a strictly typed mid-high level programming language which relies on a compiler to build and execute programs. Python is a high level interpreted language which relies on an interpreter to translate source code for computational operations. C is traditionally faster than Python in many instances due to the ability to declare types in variables, allocate memory statically and even with loops requires less instruction cycles. Most cryptographers tend to favour C for development of Cryptographic software due to the ability to manipulate memory usage, bitwise operations (heavily exploited in Frodo [2]) and the ready integration into OpenSSL. C also can then make use of the AVX2 instruction set [57] (currently used by SIDH [41] and NewHope [58]) to improve vector operations. The table below summarises the results obtained for each parameter in both C and Python alongside the total execution time.

| Parameter | Time(ms) | |
|---|---|---|
| | **C** | **Python** |
| **M** | $7.25 \pm 0.02$ | $2.07 \pm 0.00$ |
| **Alice0** | $22.36 \pm 0.13$ | $28.531 \pm 0.01$ |
| **Bob** | $1.34 \pm 0.01$ | $1.17 \pm 0.05$ |
| **Alice1** | $1.01 \pm 0.01$ | $0.84 \pm 0.03$ |
| **Total** | $30.5 \pm 1.4$ | $38.9 \pm 2.1$ |

Table 10: Runtime performance of the PQKE.

Perhaps the most notable value in the table is the performance of the parameter **M**. Python utilises the NumPy [50] library which uses locality of reference [59] at its back-end making matrix construction with random integers faster. In C to generate a uniformly random matrix, using the '%' (remainder) operator generally requires more instruction cycles at low level assembly also explaining its slower runtime. The other parameters follow their programming language counterparts (see Appendix A.2.1) closely, while Python shows faster results in this table other Lattice Dimensions convincingly show that C is also as fast with each parameter. Finally, C is always faster for the total key

[17]These are only available in C

exchange time. This is due to the subtle optimisations experienced in the differences between a compiler and an interpreted language.

**Communicational Complexity** The communicational complexity for both Programming Languages is the same. In order to estimate values for communicational complexity it is important to note which parameters are being transmitted between Alice & Bob. Alice sends $\mathbf{P}_A$ which is a $512 \times 512$ matrix for $n = 512$. Bob sends $(\mathbf{p}_B, \sigma)$ which are $512 \times 1$ vectors. Using Integer types in either programming language sends 4 bytes of data and while this could be optimised for lower dimensions it will be counteracted by integer overflow so 4 bytes of storage are essential. $\mathbf{A} \rightarrow \mathbf{B}$ denotes Alice's message to Bob.

| Message | Communication(in bytes) |
|---|---|
| $\mathbf{A} \rightarrow \mathbf{B}$ | 1048576 |
| $\mathbf{B} \rightarrow \mathbf{A}$ | 4096 |

Table 11: Communicational Complexity

Note that the total communicational complexity is $\mathcal{O}(n^2+2n)$. The Communicational complexity in an LWE protocol is generally expected to be quite high. Alice's message to Bob is heavy on memory usage due to the generation of all elements of the matrix without any operations to free memory or permute rows to avoid severe memory consumption (see section 6.4 for more information).

**Throughput** Major throughput measurements are generally a good indicator of runtime benchmarks for multiple key exchanges in a real world setting. Throughput generally serves as a better metric when Key Exchanges are used in TLS. Nonetheless, the throughput metric for raw cryptographic operations can still be worthwhile as they can inform decent analysis of the key exchanges runtime uncertainty. Throughput measurements were achieved by using the Linux/Unix command `timeout` which permits execution of `gcc` or `python` scripts until a time limit expires.

| Seconds | Throughput | |
|---|---|---|
| | **C** | **Python** |
| 0.1 | 2 | 1 |
| 1 | 30 | 10 |
| 10 | 299 | 103 |

Table 12: Throughput for standalone cryptographic operations of the PQKE

From the results presented in throughput and runtime it is clear to see that the C implementation far surpasses its python equivalent. It is also more adaptable for the system design of the PQKE and allows direct integration into OpenSSL. Furthermore, it bears closer resemblance to the actual PQKE because of the availability of a Discrete Gaussian Library.

### 6.3.2 Lattice Dimension vs Runtime

Another important test result includes the performance of the Key Exchange for various lattice dimensions. There are 2 tables in this section which discuss the performance of individual parameters with varying lattice dimensions (coupled with graphs in Appendix A.2.1).

| Lattice Dimension | Time(ms) | | | | |
|---|---|---|---|---|---|
| | **M** | **Alice0** | **Bob** | **Alice1** | **Total** |
| 312 | 1.3 | 8.18 | 0.46 | 0.32 | 10.38 |
| 468 | 2.94 | 18.53 | 1.09 | 0.79 | 23.46 |
| 512 | 7.25 | 22.36 | 1.34 | 1.01 | 30.50 |
| 684 | 6.83 | 40.60 | 2.36 | 1.93 | 51.95 |
| 752 | 8.48 | 47.73 | 3.38 | 3.65 | 63.40 |
| 819 | 9.91 | 57.40 | 3.44 | 3.21 | 74.12 |
| 945 | 14.09 | 76.97 | 6.37 | 5.78 | 103.41 |
| 1005 | 16.34 | 85.87 | 5.39 | 5.02 | 112.81 |
| 1128 | 20.06 | 107.87 | 7.79 | 7.98 | 143.72 |

Table 13: Lattice Dimension vs Runtime in C.

The parameter that scales poorly with increasing Lattice dimensions is **Alice0**. This is due to the higher level of operations required for increasing matrix dimensions. Parameter **M** also scales poorly due to the greater number of instruction cycles faced from modulo operations. Other vectorial operations scale linearly which is as expected. The Key Exchange's total runtime performs poorly with increasing Lattice Dimensions. It is important to choose an optimal dimension which provides significant balance. For $n = 512$ the throughput value from the previous section tends to favour more internet communications and retains provable security quite efficiently.

| Lattice Dimension | Time(ms) | | | | |
|---|---|---|---|---|---|
| | **M** | **Alice0** | **Bob** | **Alice1** | **Total** |
| 312 | 0.82 | 10.71 | 0.36 | 0.42 | 14.27 |
| 468 | 1.79 | 23.71 | 0.74 | 0.76 | 31.26 |
| 512 | 2.07 | 28.53 | 1.17 | 0.84 | 38.90 |
| 684 | 3.72 | 50.31 | 1.53 | 1.98 | 76.01 |
| 752 | 4.47 | 61.48 | 1.69 | 3.01 | 101.60 |
| 819 | 5.29 | 72.82 | 3.60 | 3.26 | 132.00 |
| 945 | 7.02 | 96.96 | 4.71 | 5.69 | 188.694 |
| 1005 | 7.94 | 108.61 | 5.34 | 5.09 | 256.25 |
| 1128 | 10.04 | 137.84 | 8.73 | 8.37 | 331.21 |

Table 14: Lattice Dimension vs Runtime in Python.

Python shows much better performance for parameter **M** due to locality of referencing [59] but worse performance for **Alice0** due to significant overhead in NumPy matrices. Other parameters have similar performances in C but the more notable measurement is centred on the total key exchange. Python has a worse rate of performance due to the lack of subtle optimisations that could be made in C (like pre-processor declarations). Profiling generally showed that this worse performance was also caused by the lack of a discrete gaussian library. The performance dip was precipitated by the greater number of values `CheckRobustExtractor()` was forced to correct due to the usage of a continuous gaussian distribution.

### 6.3.3 DGS vs Runtime

This section includes runtime test results for all the algorithms present in the discrete gaussian library.

| Algorithm | Time (ms) | | | | |
|---|---|---|---|---|---|
| | **M** | **Alice0** | **Bob** | **Alice1** | **Total** |
| Uniform Table | 7.25 | 22.36 | 1.34 | 1.01 | 30.5 |
| Uniform Logtable | 7.13 | 115.86 | 1.62 | 1.01 | 125.87 |
| Uniform Online | 7.12 | 33.34 | 1.35 | 1.00 | 42.94 |
| Sigma2 | 7.14 | 115.18 | 1.61 | 1.01 | 125.16 |
| Alias | 7.12 | 21.16 | 1.31 | 0.94 | 30.71 |

Table 15: DGS vs Runtime

A notable problem arises from the Logtable and Sigma2 algorithm. Simple profiling showed that more queries were forced in the table to generate values that would not be rejected by the probability distribution specified in the algorithms showcasing worse performance for larger structures like matrices (notable in **Alice0**). The Alias algorithm outperformed the Uniform Table algorithm but due to its unclear setting as a discrete gaussian distribution (as the Alias algorithm uses arbitrary distributions) it was not incorporated in the final implementation.

## 6.4 Performance against Liboqs

The PQKEs chosen for direct comparison were selected from liboqs [48] discussed in section 2.6. The following table only lists results for the C implementation of the PQKE.

| **Scheme** | **Alice0** | **Bob** | **Alice1** | **Communication** | | **Claimed Security** | |
|---|---|---|---|---|---|---|---|
| | (ms) | (ms) | (ms) | **A→B** | **B→A** | **classical** | **quantum** |
| RSA 3072-bit | - | 0.09 | 4.49 | 387/0* | 384 | 128 | - |
| ECDH `nistp256` | 0.37 | 0.70 | 0.33 | 32 | 32 | 128 | - |
| BCNS | 1.01 | 1.59 | 0.17 | 4,096 | 4,224 | 86 | 78 |
| NewHope | 0.11 | 0.16 | 0.03 | 1,824 | 2,048 | 229 | 206 |
| NTRU `EES743EP1` | 2.00 | 0.28 | 0.15 | 1,027 | 1,022 | 256 | 128 |
| **Jintailwe** | **22.36** | **1.34** | **1.01** | **1,048,576** | **4,096** | - | - |
| Frodo Recomm. | 1.13 | 1.34 | 0.13 | 11,377 | 11,296 | 144 | 130 |
| Frodo Paranoid | 1.25 | 1.64 | 0.15 | 13,057 | 12,976 | 177 | 161 |
| SIDH | 135 | 464 | 301 | 564 | 564 | 192 | 128 |

Table 16: Performance of Standalone Cryptographic Operations: Mean runtime of standalone cryptographic operations. Communication is given in bytes. Amended table includes the PQKE implemented in this paper.

This PQKE has worse runtime for **Alice0** in particular because of the lack of optimisation done on matrix generation. In Frodo [2] elements stored in matrices which should be in $\mathbb{Z}_q$ are actually in $\mathbb{Z}_{2^{16}} = \mathbb{Z}/(2^{16-x}q)\mathbb{Z}$ which allows lesser communicational complexity as all arithmetic operations of *modulo* $2^{16}$ are performed for free. This also affects the communicational complexity of the PQKE making it a slightly worse PQKE than Frodo. Once these optimisations are performed the performance of the PQKE will improve greatly.

# 7 Evaluation

This project has successfully achieved a Post-Quantum Key Exchange for Multiple-Bits of security. The open source C implementation of this key exchange can be modified to accommodate different parameters with an easily adjustable header file and the functions of this Key Exchange can be modified to integrate it into Liboqs for an OpenSSL implementation or can even be used for statistical applications.

The main objectives of this project were met with to a great extent. A proper background on Cryptography, current cryptographic algorithms and their vulnerabilities to Quantum Computers, Lattice Cryptography as a Quantum Safe primitive and Post-Quantum Key Exchanges were properly documented in this report. The Background also touched on TLS and the Liboqs library which can allow any developer to add PQKEs to prototype in TLS. The protocol was successfully implemented in 2 different programming languages with suboptimal (but pragmatic) versions in both. Optimisations in the C language were significantly achieved through Compiler (`-O3`) optimisations. Memory Usage was optimised in C to ensure significant reduction in Communication but other optimisations can still reduce this further. A full test-suite for the PQKE with command line arguments to test all the results required for cryptographic metrics has also been documented in this report and implemented in the source code. Additionally, this report and project detail a step-by-step guide in the integration of discrete gaussians with programs in C.

Even though this PQKE was implemented up to a great degree, future internet applications will rely on its integration in TLS so that it can be used in browsers. In hindsight, implementing all the necessary files and prerequisite software that could allow such an implementation to be explored in TLS was unrealistic because of the shear number of bottlenecks that originated in the Implementation and because of the lack of alternative software or proper documentation to integrate a PQKE into TLS (even in Liboqs, though this is discussed further in Future Work). While this system does efficiently construct a PQKE it does not showcase further use of it in Encryption functions or in Authentication. Integration of the PQKE into TLS has been suggested through Liboqs with a guide on what fundamental modifications will be required to integrate it. Finally, the implemented PQKE is suboptimal (but not impractical). It achieves an acceptable runtime and throughput metric but with further optimisations applied to matrix construction through AVX2 [57] or through memory optimisations (like those done in Frodo [2]) the Key Exchange could show even greater performance.

# 8 Conclusions and Future Work

## 8.1 Future Work

A PQKE must be implemented in OpenSSL to test connections in TLS so that future utilizations of the PQKE can be realized in Web Browser applications. While this project may have succeeded in setting up most of the required library materials and configurations needed to implement such a protocol in liboqs [5] there is still some modification required in the core liboqs library to allow it to be directly interfaced with OpenSSL algorithms.

### 8.1.1 Integration with Liboqs

For the reasons discussed above this paper concludes that the integration of the PQKE into OpenSSL is essential. This can be achieved by using the Liboqs library which is available under a free license. This section will aim to provide some approaches which would allow a developer to integrate the PQKE into Liboqs.

Liboqs is a large library organised by a standard TLS cipher-suite hierarchy. It makes use of the following block diagram structure:

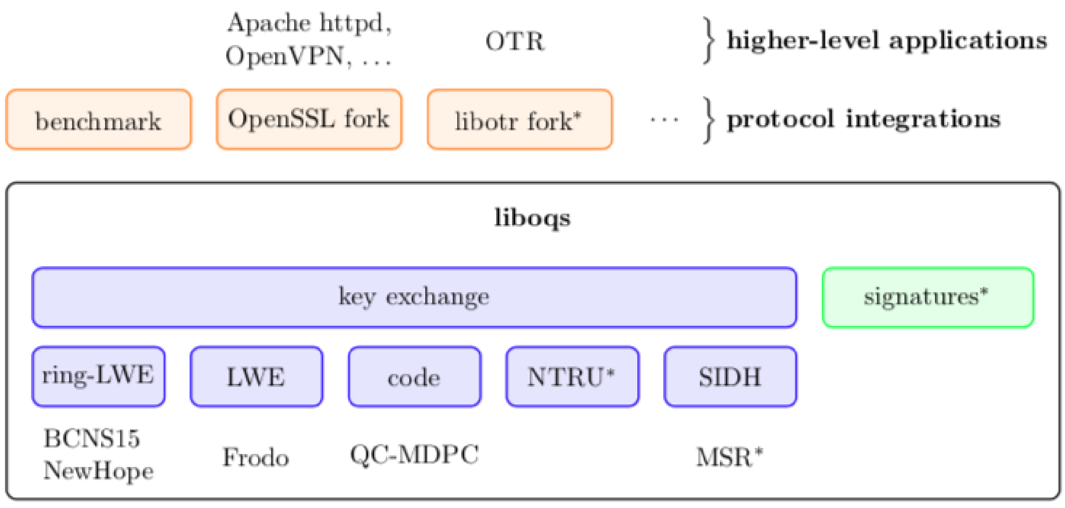


Figure 5: Architecture of the Open Quantum Safe project [5].

There is a fork of OpenSSL v1.0.2 which has wrapper code embedded around most of the OpenSSL files that invoke public-key cryptography algorithms. This allows liboqs to compile a public-key algorithm into the open quantum safe's fork of OpenSSL and use OpenSSL commands like `speed`. If a developer were to directly interface their own algorithm into OpenSSL without using liboqs it would involve changing a large number of files and adding over 2100 lines of code which would be significantly tougher than using the liboqs API.

In order to add a program to the API a developer must first segregate all of the parameter functions of the key exchange into separate functions without synthesising all of them in the `main()` execution script. This is due to the `test` script present in liboqs which times each parameter and requires them to be in a functional format whilst updating global variables. Apart from the modifications required in the source code discussed in section 2.6 a developer will also modify files like `travis.yml`, `Makefile.am` and `Configure.ac` so that the updated files can appear in the Build process. This step in particular is quite difficult and requires direct comparison against a protocol like Frodo with each of its relevant lines of code organised in these files tallied and compared with a custom PQKE. Finally, a developer will wrap their key exchange in the `kex` folder and modify the `kex.c` and `kex.h` files so that they can reflect clearly in the benchmark testing scripts. These scripts are capped at timeout values so it is imperative that the PQKE performs optimally so that these scripts can showcase correct results.

### 8.1.2 Optimising the PQKE further

For the reasons discussed in section 6.4, the PQKE performs worse than existing LWE algorithms. This section provides 2 specific optimisations which will definitely ensure the PQKEs runtime and memory optimality for future usage.

- Use the AVX2 instruction set: The new AVX2 instruction set released by Intel performs faster matrix multiplication calculations, a feat which is required by this PQKE as the longest runtime lies in a matrix multiplication. This method will definitely optimise the PQKE further as the SIDH method documented in liboqs performs poorly without an AVX2 instruction set while SIDH in itself performs much faster with the instruction set and has similar matrix multiplications as the PQKE implemented in this paper.

- Permute matrix elements based on modular arithmetic: A technique used in Frodo to balance memory consumption and runtime with security relied heavily on modular arithmetic. In that technique the transmitted matrix only generates the first 15 rows of the matrix based on a discrete gaussian distribution and generated the rest of the entries based on a polynomial ring which is randomly permuted. The same principle can be applied to the PQKE implemented in this paper, though further research is warranted into the security of such an optimisation.

## 8.2 Concluding Remarks

Quantum computing is on the rise and modern cryptographic schemes are now at a razor's edge. Many researchers have started exploring various new mathematical fundamentals to generate quantum resistant cryptographic protocols. This work adds further research and effort towards PQC and successfully implements a Key Exchange of varying lengths which is both computationally fast and incurs acceptable communicational complexity. It also laid significant groundwork for the integration of this protocol into liboqs to solidify its applicability in internet applications. Additionally, the integration of discrete gaussian distributions can open new doors to researchers, cryptographers and mathematicians. Once this PQKE is optimised in memory complexity and matrix operations, it can provide exceptional performance in modern Internet Applications and make Post-Quantum Cryptography more secure and robust.

# References

[1] Write Opinions. A two dimensional lattice. http://www.writeopinions.com/lattice-based-cryptography, 2017. Accessed on 2018-01-27.

[2] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. Cryptology ePrint Archive, Report 2016/659, 2016. https://eprint.iacr.org/2016/659.

[3] Imperva Incapsula TLS Handshake. https://www.incapsula.com/cdn-guide/cdn-and-ssl-tls.html.

[4] Xiaodong Lin Jintai Ding, Xiang Xie. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive, Report 2012/688, 2012. https://eprint.iacr.org/2012/688.

[5] Open Quantum Safe. https://openquantumsafe.org/.

[6] NASA and Google make a quantum computer. https://www.popsci.com/google-and-nasa-score-new-quantum-computer. Accessed: 2018-01-16.

[7] Quantum Computer Principles. http://www.thphys.nuim.ie/staff/joost/TQM/QvC.html. Accessed: 2018-01-16.

[8] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[9] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[10] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[11] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on post-quantum cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016.

[12] Cryptography Explained. https://www.techopedia.com/definition/1770/cryptography. Accessed: 2018-01-17.

[13] Symmetric Cryptography explained. https://www.cl.cam.ac.uk/~jac22/books/mm/book/node333.html. Accessed: 2018-01-20.

[14] Miles E Smid and Dennis K Branstad. Data encryption standard: past and future. *Proceedings of the IEEE*, 76(5):550–559, 1988.

[15] Frederic P Miller, Agnes F Vandome, and John McBrewster. Advanced encryption standard. 2009.

[16] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996.

[17] Threats Posed by Quantum Computers. https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/pqcrypto-2016-presentation.pdf. Accessed: 2018-01-20.

[18] Asymmetric Cryptography explained. https://www.globalsign.com/en/ssl-information-center/what-is-public-key-cryptography/. Accessed: 2018-01-20.

[19] PUB Fips. 186-2. digital signature standard (dss). *National Institute of Standards and Technology (NIST)*, 20:13, 2000.

[20] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17. ACM, 2015.

[21] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 417–426. Springer, 1985.

[22] S McCURLEY Kevin. The discrete logarithm problem. *Cryptology and computational number theory*, 42:49, 1990.

[23] Eric Rescorla. Diffie-hellman key agreement method. 1999.

[24] Ian F Blake and Theo Garefalakis. On the complexity of the discrete logarithm and diffie–hellman problems. *Journal of Complexity*, 20(2):148–170, 2004.

[25] RSA Key Exchange. http://sergematovic.tripod.com/rsa1.html. Accessed: 2018-01-25.

[26] Joe P Buhler, Hendrik W Lenstra Jr, and Carl Pomerance. Factoring integers with the number field sieve. In *The development of the number field sieve*, pages 50–94. Springer, 1993.

[27] Clara Löh. *Geometric Group Theory: An Introduction*. Springer, 2017.

[28] Dr. Cong Ling. Lattice-based cryptography. part ii of quantum information and post-quantum cryptography, 2017.

[29] Daniele Micciancio. Lattice algorithms & applications, 2010. https://cseweb.ucsd.edu/classes/wi10/cse206a/lec1.pdf.

[30] Chris Peikert et al. A decade of lattice cryptography. *Foundations and Trends® in Theoretical Computer Science*, 10(4):283–424, 2016.

[31] Paul E. Black and Vreda Pieterse. Algorithms and theory of computation handbook, ”np-hard”. https://xlinux.nist.gov/dads/HTML/nphard.html, 1999.

[32] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.

[33] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[34] Chris Peikert. Lattice cryptography for the internet. In *International Workshop on Post-Quantum Cryptography*, pages 197–219. Springer, 2014.

[35] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.

[36] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. Cryptology ePrint Archive, Report 2013/293, 2013. `https://eprint.iacr.org/2013/293`.

[37] IND-CPA Security. `http://cse.iitkgp.ac.in/~debdeep/courses_iitkgp/FCrypto/scribes/scribe8.pdf`.

[38] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[39] Hermite Normal form of LWE. `https://web.eecs.umich.edu/~cpeikert/pubs/slides-barilan5.pdf`.

[40] Tim Van Erven and Peter Harremos. Rényi divergence and kullback-leibler divergence. *IEEE Transactions on Information Theory*, 60(7):3797–3820, 2014.

[41] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie-hellman. In *Annual Cryptology Conference*, pages 572–601. Springer, 2016.

[42] William Whyte, Mark Etzel, and Peter Jenney. Open source ntru public key cryptography algorithm and reference code, 2013. *Available under the Gnu Public License (GPL) at https://github. com/NTRUOpenSourceProject/ntru-crypto*, page 71.

[43] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of tls-dhe in the standard model. In *Advances in Cryptology–CRYPTO 2012*, pages 273–293. Springer, 2012.

[44] Joppe W Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 553–570. IEEE, 2015.

[45] Tim Dierks and Christopher Allen. The tls protocol, version 1.0. internet engineering task force. *RFC 2246*, 1999.

[46] IBM explains TLS. `https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm`. Accessed Online at 2018-06-15.

[47] Open SSL for TLS simulations. `https://www.openssl.org/`.

[48] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In *International Conference on Selected Areas in Cryptography*, pages 14–37. Springer, 2016.

[49] Martin R. Albrecht and Michael Walter. dgs, Discrete Gaussians over the Integers. Available at `https://bitbucket.org/malb/dgs`, 2018.

[50] NumPy, Matrix Library for Python. http://www.numpy.org/.

[51] Takashi Shinzato. Box muller method. *Hitotsubashi University, Tokyo*, 2007.

[52] James Howe, Ayesha Khalid, Ciara Rafferty, Francesco Regazzoni, and Maire O'Neill. On practical discrete gaussian samplers for lattice-based cryptography. *IEEE Transactions on Computers*, 2016.

[53] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Advances in Cryptology–CRYPTO 2013*, pages 40–56. Springer, 2013.

[54] GNU Multi-precision MPFR Library. https://www.mpfr.org/. Accessed Online at 2018-05-28.

[55] GNU Multi-precision GMP Library. https://gmplib.org/. Accessed Online at 2018-05-28.

[56] Norman D Jorstad and TS Landgrave. Cryptographic algorithm metrics. In *20th National Information Systems Security Conference*, 1997.

[57] AVX2 Instruction Set Benefits - intel. https://software.intel.com/en-us/articles/how-intel-avx2-improves-performance-on-server-applications. Accessed Online at 2018-06-16.

[58] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange-a new hope. In *USENIX Security Symposium*, volume 2016, 2016.

[59] Spyros Angelopoulos, Reza Dorrigiv, and Alejandro López-Ortiz. List update with locality of reference. In *Latin American Symposium on Theoretical Informatics*, pages 399–410. Springer, 2008.

[60] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM Journal on Computing*, 37(1):267–302, 2007.

# A Appendix

The GitHub repository with complete implementations and scripts to automate results can be found at https://github.com/Afrazinator/fypafraz. The repository used for Discrete Gaussians can be found at https://github.com/malb/dgs. The liboqs repository which was used to compare other PQKEs can be found here https://github.com/open-quantum-safe/liboqs/tree/master

## A.1 Mathematical Lemmas for a Simple Provable secure Key Exchange based on the Learning With Errors Problem

These mathematical Lemmas are part of the paper "A simple provably secure key exchange based on the learning with errors problem" [4] by Jintai et. al and are only provided as reference in this document to justify the parameter selection discussed in the Implementation section of this report.

The robust extractor and signal functions are recalled in their full form from section 5.2.2. The Lemmas stated here are not provided with their full proof.

A bound [60] of the norm of the Gaussian Distribution will be used as follows:

**Lemma 1** *For any $s \geq \omega(\sqrt{\log n})$, we have:*

$$\Pr_{x \leftarrow \mathcal{D}_{\mathbb{Z}^n,s}}[||x|| > s\sqrt{n}] \leq 2^{-n} \tag{3}$$

**Lemma 2** *Let $q > 8$ be an odd integer, the function $E$ represents a robust extractor with respect to $S$ and has error tolerance $\frac{q}{4} - 2$.*

**Lemma 3** *For any odd $q > 2$, if $x$ is uniformly random in $\mathbb{Z}_q$ then E(x) is uniformly random, condition on $\sigma$, where $\sigma \leftarrow S(x)$.*

**Lemma 4** *If $8(\alpha q)^2.n \leq \frac{q}{4} - 2$, then $S_{KA} = S_{KB}$ with overwhelming probability*

## A.2 Test Results

All the test results discussed in section 6.

### A.2.1 Execution Time vs Lattice Dimensions

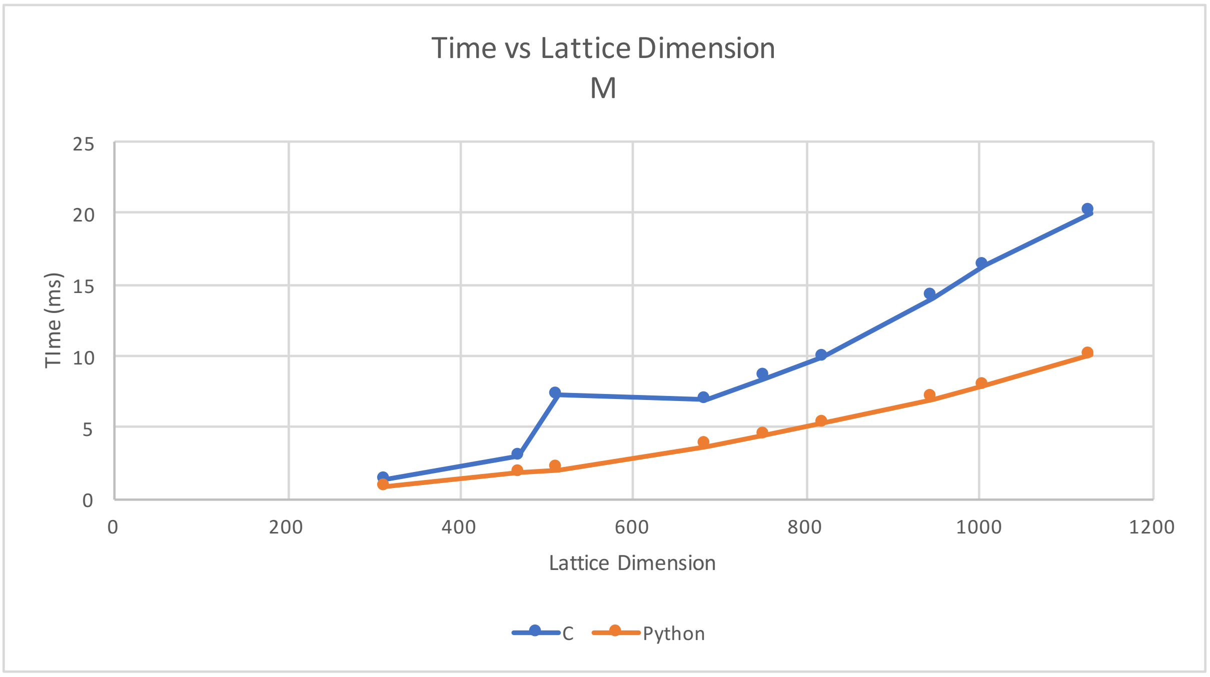


Figure 6: Parameter **M**'s runtime performance with growing Lattice Dimensions

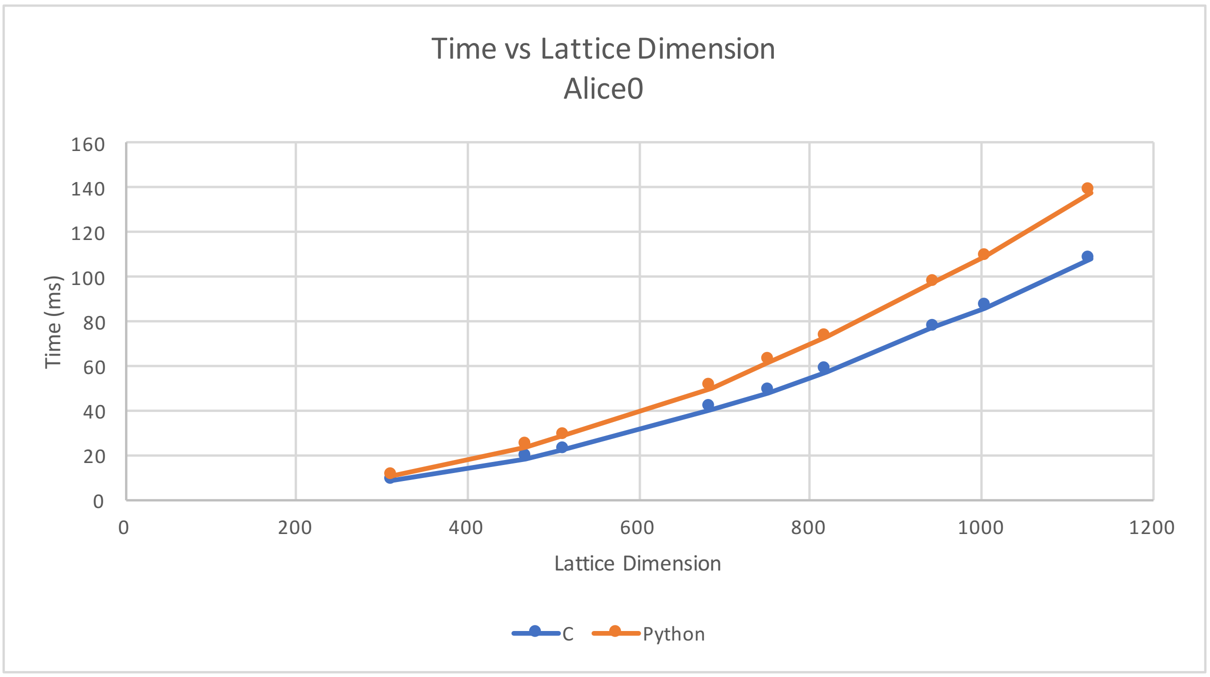


Figure 7: Parameter **Alice0**'s runtime performance with growing Lattice Dimensions

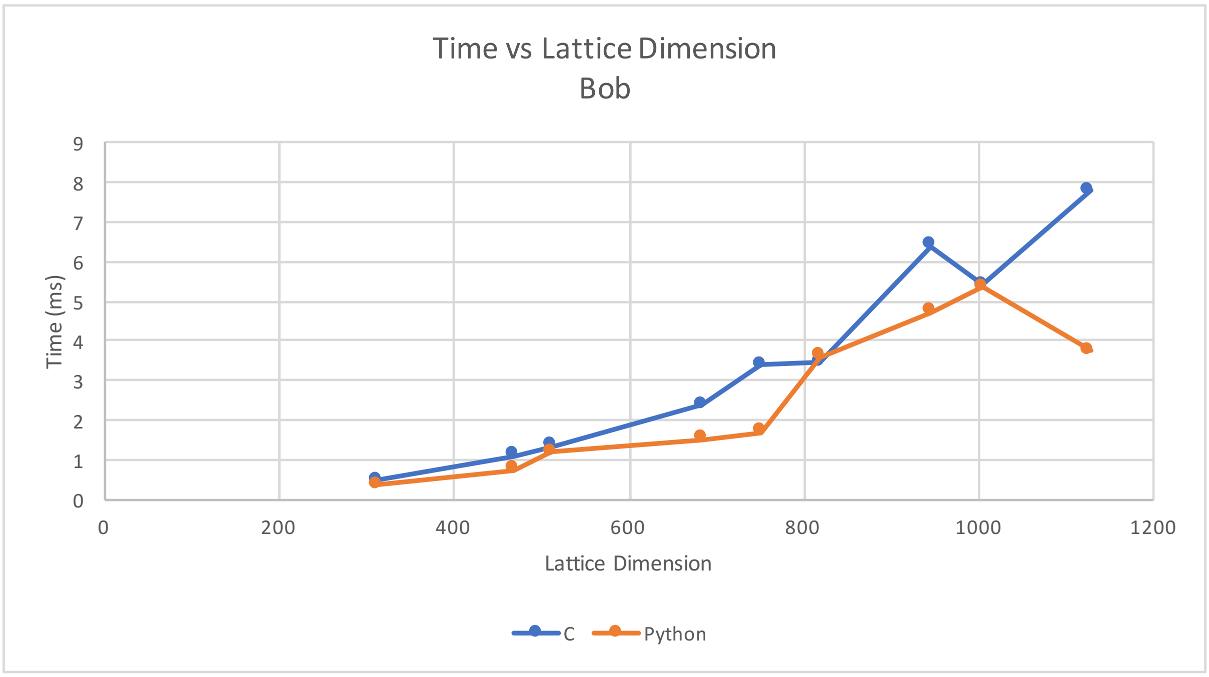


Figure 8: Parameter **Bob**'s runtime performance with growing Lattice Dimensions

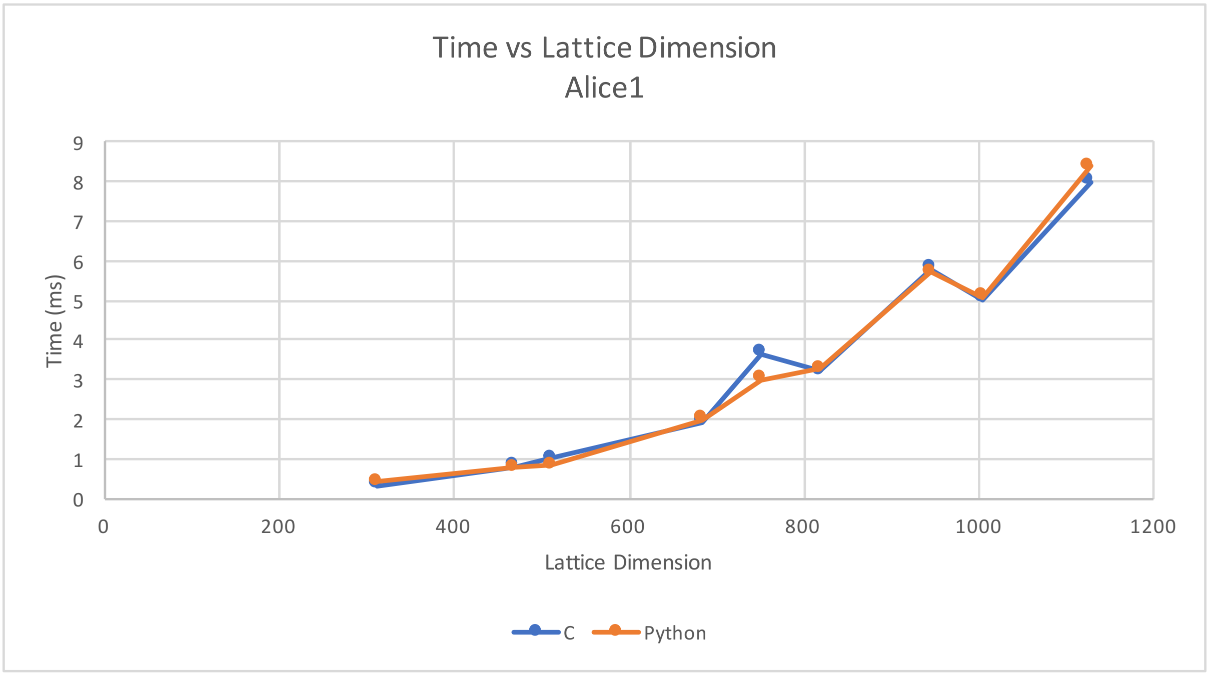


Figure 9: Parameter **Alice1**'s runtime performance with growing Lattice Dimensions

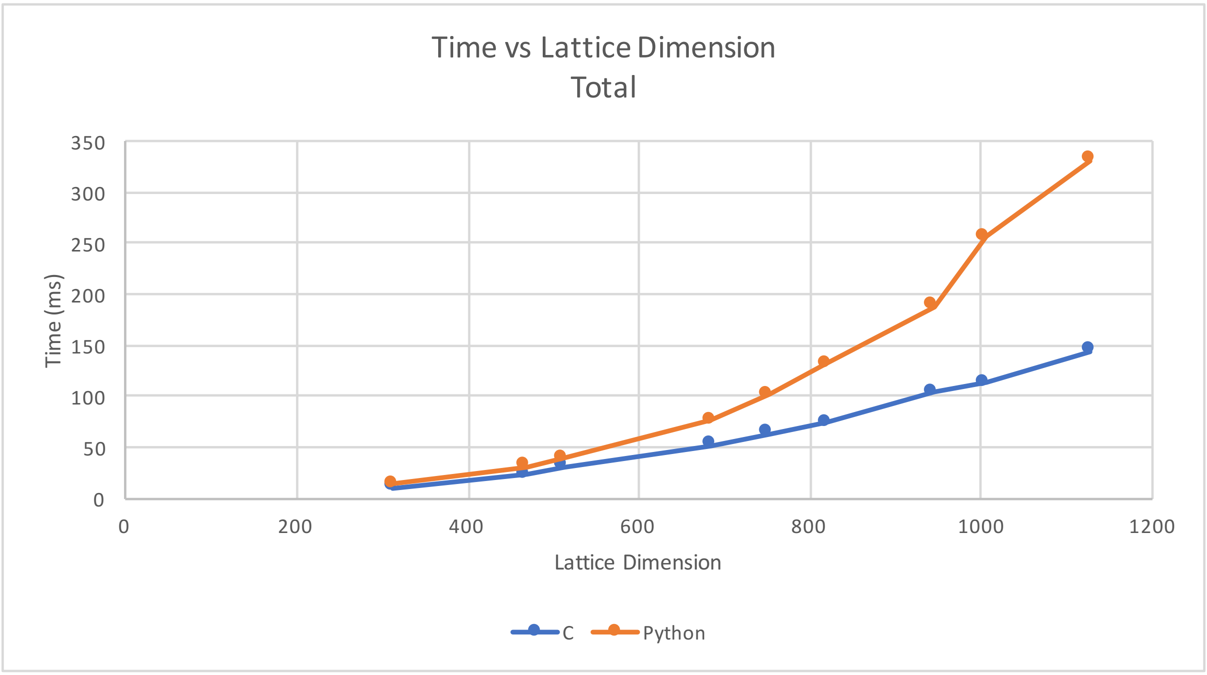


Figure 10: Parameter **Total**'s runtime performance with growing Lattice Dimensions

### A.2.2 Execution Time vs DGS

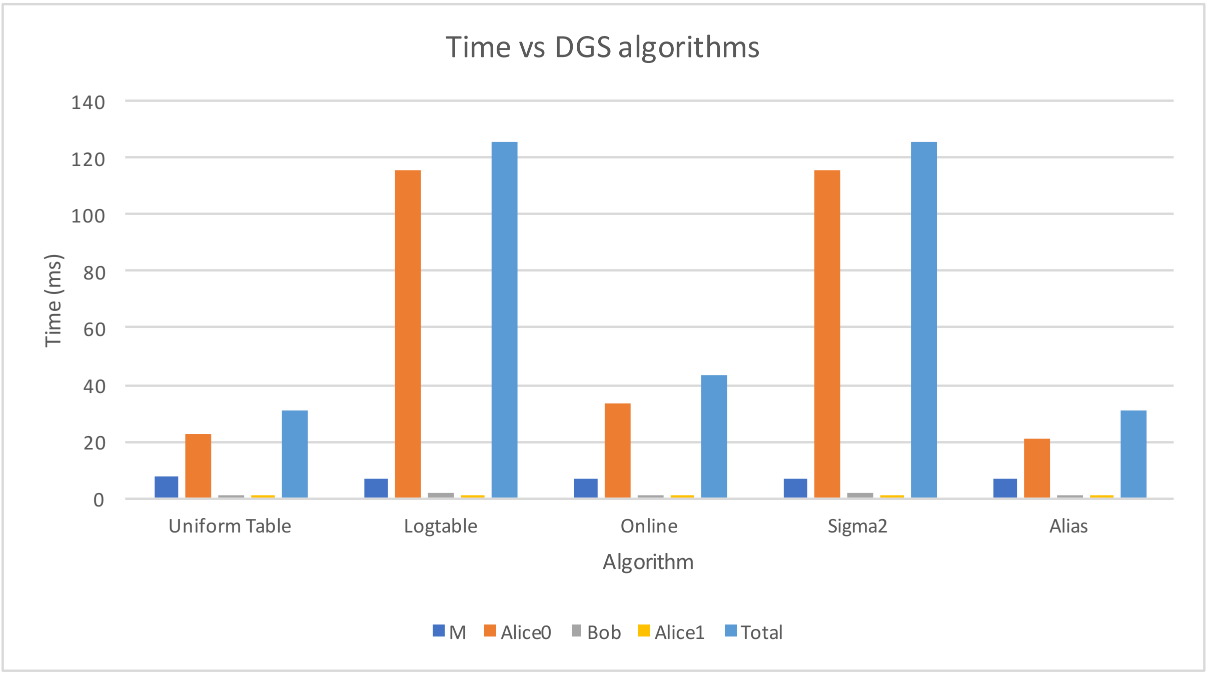


Figure 11: DGS Algorithms and their runtimes (in ms).