Post Quantum Cryptography: Public Key and Key Exchange Methods using LWE and Ring-LWE

Edinburgh Napier
UNIVERSITY

Coursework Submission for MSc ASDF CSN11131

Edinburgh Napier University 2022-2023 Trimester 2

Alessandro Ravizzotti

40621497

Abstract

In recent years, NIST decided to start to look for Cryptography resistant against Quantum Computers. In this paper have been analyzed a few aspects of Post Quantum Cryptography, considering only the Lattice-based branch. After understanding the basics of Lattice-based cryptography the Public Key and Key Exchange methods based of LWE and Ring-LWE have been explored comparing those to recent Exchange Key methods like DH and ECDH. Furthermore, the practical implementation has been studied and developed.

Keywords: PQC, Lattice, Public Key, Key Exchange, Learning With Errors, Ring Learning With Errors

1 Introduction

This paper aims to research information about Post Quantum Cryptography, with a specific interest in Lattice-bassed cryptography. More precisely we want to explore the new algorithms for Public Key and Key Exchange that make use of the Learning With Error problem, with direct comparisons with the methods used nowadays (EC and ECDH). In addition to the theoretical aspect, implementations of the studied methods are carried out in Python language.

2 Literature Review

2.1 Post Quantum Cryptography

In the last decades, the research concerning quantum computers has grown enormously because they can solve, through quantum mechanics, difficult problems which are impossible for classical computers. These difficult mathematical problems, such as discrete logarithm, integer factorization and elliptic curve, are the basis of currently used cryptographic systems.

The reason why quantum computers would be able to solve the mathematical problems used in current cryptography is due to two algorithms: Shor's algorithm and Grover's algorithm. Through Shor's algorithm (Shor 1994) it is possible to perform, in polynomial time, integer factorization and discrete logarithms, while Grover's algorithm (Niederhagen & Waidner 2017) increases the speed in search problems of $\sqrt{N}$ where N is the total number items.

The advent of quantum computers could therefore compromise the integrity and confidentiality of digital communications currently in use.
Quantum-resistant cryptography (also known as post-quantum cryptography) is the solution to this problem, as by definition it is resistant against attacks carried out by quantum and classical computers. By making this secure cryptographic systems, its goal is achieved. Another consideration to be made in this regard consists in the fact that the PQC must be able to be used with the currently existing protocols. (Alagic et al. 2020)

Today there are still no quantum computers large enough to solve these difficult mathematical problems, but it is assumed that it will be possible in the next 20 years or so. For this reason, the National Institute of Standards and Technology (NIST) in 2016, decided to start researching new cryptographic standards, safe for both types of computers, classical and quantum.

There are five main branches of post-quantum cryptography (Niederhagen & Waidner 2017):

- Lattice-based
- Code-based
- Hash-based
- Multivariate-quadratic
- Supersingular elliptic curve isogenies

but this paper is going to be focused on specific aspects of Lattice-based PQC. It is important to know that the security of lattice-based cryptography, as all of the other PQC branches, is not yet well-understood against quantum-computer attacks.

Lattice based

Lattice-based is the first candidate branch from the NIST PQC because of the simplicity of the algorithms, furthermore, they are efficient, highly parallelizable and most important the encryption is fully homomorphic, this means that any kind of mathematical operation can be performed on the cypher text, instead of on the actual data itself, thus data can remain confidential while it is processed. (Kumar & Pattnaik 2020)(Armknecht et al. 2015) (Gentry 2009) (Van Dijk et al. 2010) (Yi et al. 2014)
The hard problems regarding lattice-based cryptography are the Closest Vector Problem (CVP) and the Shortest Vector Problem (SVP).

The lattice used in Post-Quantum Cryptography is composed of a number of dimensions, which makes these problems hard problems. To simplify the explanation process and because of representations problems, just 2 dimensions are going to be used in the following lattice creation.

At first, a few points must be declared to build the lattice, the centre point is called $O(0,0)$ and two points: $P_1(3,2)$ and $P_2(2,7)$ must be chosen; from which are created the two main vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$: $\overrightarrow{v_1} = \overrightarrow{OP_1}$ and $\overrightarrow{v_2} = \overrightarrow{OP_2}$.

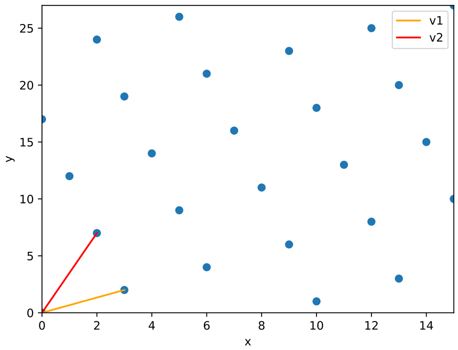

Figure 1: Lattice: creation

Considering a and b two integer values, starting from the two main vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$, it is possible to apply math operations using the following formula:

$$\overrightarrow{v_R} = a \cdot \overrightarrow{v_1} + b \cdot \overrightarrow{v_2}$$

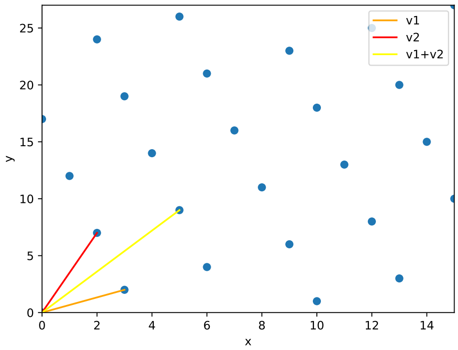

Figure 2: Lattice: math operation with the main vectors

After all these passages the lattice is ready to be used, so a point on the lattice has to be picked P_A and the vector from it to the origin will be calculated $\overrightarrow{OP_A}$. Afterwards a small error vector $\overrightarrow{e}$ is added onto $\overrightarrow{OP_A}$ that introduce a new point P_B that leads to a new vector $\overrightarrow{OP_B}$. The hard CVP problem consist into find the original vector $\overrightarrow{OP_A}$ knowing only $\overrightarrow{OP_B}$ and the lattice (the error vector $\overrightarrow{e}$ is unknown).

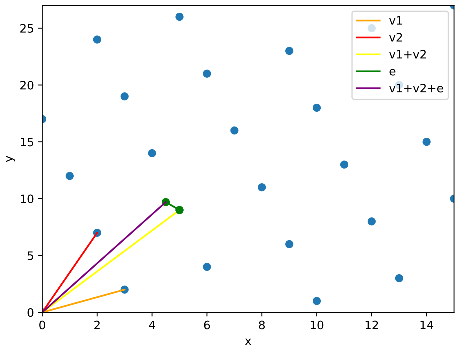

Figure 3: Lattice: add the error

Learning With Errors (LWE)

The Shortest Vector Problem is what the Learning with Errors (LWE) method is based on.

Considering n, m two integers, q a prime number, and the following equation $b = A \times s$, A being a two-dimensional random matrix in $\mathbb{Z}_q^{n \times m}$ and b a vector in $\mathbb{Z}_q^n$, it is relatively easy to do the reverse calculation and find the secret vector s in $\mathbb{Z}_q^m$. However, if the previous equation changes to $b = A \times s + e$ so that a new error (or noise) vector e in $\mathbb{Z}_q^n$ is been added, this becomes a hard problem (Cheon et al. 2018)(An & Seo 2020)(Buchanan 2023a). The mathematical calculation is shown below considering $n = 5$, $m = 4$ and $q = 11$. It should be noted that in this case A is a two-dimensional matrix and consequently b, s and e

are one-dimensional matrices, but it is possible to increase (or decrease) the number of dimensions used for the arrays (by adding new variables such as n and m), keeping always the number of dimensions of A greater than 1 compared to the number of dimensions of the other variables b, s and e.

$$\begin{array}{ccccccc} A \in \mathbb{Z}_{11}^{5\times 4} & & s \in \mathbb{Z}_{11}^{4} & & e \in \mathbb{Z}_{11}^{5} & & b \in \mathbb{Z}_{11}^{5} \\ \begin{bmatrix} 3 & 2 & 1 & 5 \\ 6 & 6 & 9 & 5 \\ 8 & 3 & 7 & 9 \\ 7 & 4 & 6 & 0 \\ 7 & 9 & 0 & 9 \end{bmatrix} & \times & \begin{bmatrix} 5 \\ 8 \\ 7 \\ 3 \end{bmatrix} & + & \begin{bmatrix} 0 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} & = & \begin{bmatrix} 9 \\ 1 \\ 7 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

2.2 Key Exchange Methods

The methods used so far for key exchange are the Diffie-Hellman algorithm and the elliptic curve DH (ECDH) algorithm. In the first case, the discrete logarithm problem is used, while in the second the elliptic curves are used.

Diffie-Hellman

The algorithm consists of the following points:

1. Alice and Bob agree on the generator number g and the prime number p
2. Alice and Bob generate their secret numbers respectively x and y
3. Alice and Bob calculate respectively $A = g^x\,mod(p)$ and $B = g^y\,mod(p)$
4. Alice and Bob share A and B
5. Alice and Bob calculate the shared key $g^{x\cdot y}\,mod(p)$

A graphical view of the Diffie-Hellman key exchange method, is shown in Table 1.

Alice		Bob
	$\xleftrightarrow{g\,,p}$	
$x \leftarrow gen$		$gen \rightarrow y$
$A = g^x\,mod(p)$		$g^y\,mod(p) = B$
	$\xrightarrow{A}$	
	$\xleftarrow{B}$	
$B^x\,mod(p)$	$\xleftrightarrow{shared\,key}$	$A^y\,mod(p)$

Table 1: DH Exchange Key

Shared key:

$$B^x\,mod(p) = g^{x\cdot y}\,mod(p) = A^y\,mod(p)$$

Elliptic Curve Diffie-Hellman

The algorithm consists of the following points:

1. Alice and Bob agree on the elliptic curve EC and the point G
2. Alice and Bob generate their secret numbers respectively d_A and d_B
3. Alice and Bob calculate respectively $A = d_A \cdot G$ and $B = d_B \cdot G$
4. Alice and Bob share A and B
5. Alice and Bob calculate the shared key $d_A \cdot d_B \cdot G$

A graphical view of the ECDH key exchange method, is shown in Table 2.

Alice		Bob
	$\xleftrightarrow{G\,,EC}$	
$d_A \leftarrow gen$		$gen \rightarrow d_B$
$A = d_A \cdot G$		$d_B \cdot G = B$
	$\xrightarrow{A}$	
	$\xleftarrow{B}$	
$d_A \cdot B$	$\xleftrightarrow{shared\,key}$	$d_B \cdot A$

Table 2: ECDH Exchange Key

Shared key:

$$d_A \cdot B = d_A \cdot d_B \cdot G = d_B \cdot A$$

2.3 Public Key - LWE

LWE can be used for creating a public key encryption method. In this paper, the algorithm defined by Oded Regev (Regev 2010) has been analyzed. The following points describe the process:

1. Alice chooses a random vector A of length n, a small random error vector e of length n and two values: s, the secret key and q the prime number
2. Alice calculates the vector B so that $\forall i \in n \Rightarrow B_i = A_i \cdot s + e_i \, mod(q)$. (A, B) is Alice's public key
3. Alice sends (A, B) to Bob
4. Bob takes a sample of both vector A_{sample} and B_{sample}
5. Bob calculates $u = \sum(A_{sample}) \, mod(q)$ and $v = \sum(B_{sample}) + \frac{q}{2}M \, mod(q)$, in which M stands for a bit of the plain text he wants to send to Alice
6. Bob sends Alice the encrypted bit (u, v)
7. Alice uses the formula $dec = v - su \, mod(q)$ to decrypt the bit. If $dec < \frac{q}{2} \Rightarrow M = 0$ otherwise if $dec > \frac{q}{2} \Rightarrow M = 1$

A graphical view of the LWE public key method, is shown in Table 3.

Alice		Bob
$A, e \leftarrow random$		
$s \leftarrow gen$		
$q \leftarrow genPrime$		
$\forall i \in n \Rightarrow$		
$B_i = A_i \cdot s + e_i \, mod(q)$		
	$\xrightarrow{(A,B)}$	
		$\sum(A_s) \, mod(q) = u$
		$\sum(B_s) + \frac{q}{w}M \, mod(q) = v$
	$\xleftarrow{(u,v)}$	
$dec = v - su \, mod(q)$		
$if \, dec < \frac{q}{2} \Rightarrow M = 0$		
$if \, dec > \frac{q}{2} \Rightarrow M = 1$		

Table 3: Public Key - LWE

2.4 Key Exchange - Ring LWE

Ring-LWE is very similar to basic LWE but, as the name says, it uses the ring $\mathbb{Z}[x]/(x^n+1)$. Because some polynomials operation must be used during the process of the key exchange method, the ring is important because it set a boundary to the degree of the polynomial. It should be noted that this method does not carry out any identity checks for which it is necessary to use it in combination with other techniques. (Guo et al. 2022)(Fluhrer 2016)(Bos et al. 2015)(Buchanan 2023*d*) The following points describe the process of key exchange, using Ring-LWE.

1. Alice and Bob agree on a value n which will be used to calculate $q = 2^n - 1$ which is the modulus that will be applied to all the coefficients of the polynomial
2. Alice generates a random polynomial $A = a_{n-1}x^{n-1} + a_1x + a_1x^2 + a_0$
3. Alice divides the polynomial A by the ring $x^n + 1$. The ring will set a boundary to the level of the polynomial
4. Alice shares A with Bob
5. Alice generates her secret polynomial s_A and the error polynomial e_A
6. Alice will calculate $b_A = A \times s_A + e_A$
7. Bob does the same last two steps as Alice, generating s_B, e_B and $b_B = A \times s_B + e_B$
8. Alice and Bob share their respective b values: b_A and b_B
9. Alice and Bob calculate their shared values, respectively Alice calculates $shared_A = b_B \times s_A \div (x^n + 1)$ and Bob calculates $shared_B = b_A \times s_B \div (x^n + 1)$. To notice that $shared_A = (A \times s_B + e_B) \cdot s_A \approx shared_B = (A \times s_A + e_A) \cdot s_B$, $shared_A$ and $shared_B$ do not have the same values because of the different errors.
10. Alice and Bob have to remove the error to share the exact same value using a formula shown in the code of Section 3.3 (Key Exchange - Ring LWE)

A graphical view of the Ring-LWE key exchange method, is shown in Table 4.

Alice		Bob
	$\xleftrightarrow{n,q}$	
$A \leftarrow gen$		
$A \leftarrow A \div (x^n + 1)$		
	$\xrightarrow{A}$	
$s_A, e_A \leftarrow gen$		$gen \rightarrow s_B, e_B$
$b_A = A \times s_A + e_A$		$A \times s_B + e_B = b_B$
	$\xrightarrow{b_A}$	
	$\xleftarrow{b_B}$	
$b_B \times s_A \div (x^n + 1)$	$\approx$	$b_A \times s_B \div (x^n + 1)$

Table 4: Key Exchange - LWE

3 Implementation

Since, during the literature review, we saw different explanations and uses regarding Learning With Error, it was decided to implement more than one of the aspects seen. In all three cases, we started from an already existing code, to which some modifications or improvements were made.

Test Environment

In all three implementations, Python 3.9 was chosen as the coding language, on a Windows 11 machine. The choice of this programming language is due to the ease of writing, knowledge of the language and the quality of the already existing libraries and scripts.

3.1 Learning With Errors

The code from which we started and made some modifications is (Buchanan 2023*b*). In the following code the two-dimensional matrix $A \in \mathbb{Z}_q^{n \times m}$ and the noise vector e are generated in a pseudo-random way, using the numpy library. In order for the code to work, the length of the secret vector s needs to be m, for this purpose, a check has been inserted with possible pseudo-random creation of the secret vector as well. Finally the code generates the vector $b = A \times s + e$.
The following code was used to create the schema in section 2.1 - Learning With Errors.

—— LWE.py

```
import numpy as np

def changeS():
    global s
    global m
    global q
    i = input(f”secret␣vector␣wrong␣size!␣Press␣[s]␣to␣generate␣it.”)
    if i == ’s’:
        s = np.random.randint(0, high=q, size=(m,1))
        print(f”New␣secret␣vector:\n{s}”)
    else:
        exit()

def checkS():
    global s
    global m
    global q
    if len(s) != m:
        changeS()
    for i in s:
        if i >= q:
                changeS()
    return

n = 5
m = 4
q = 11

A = np.random.randint(0, high=q, size=(n,m))
s = np.array([[5], [ 8], [ 7], [ 3]])
e = np.random.randint(-1, high=2, size=(n,1))

checkS()

b = np.matmul(A, s)%q
b = np.add(b, e)%q
print(f"A:␣{A}\n")
print(f"s:␣{s}\n")
print(f"e:␣{e}\n")
print(f"b:␣{b}")
```

3.2 Public Key - LWE

The code from which we started and made some modifications is (Buchanan 2023*c*). The following code is an example of a public key using LWE. The entire procedure explained in section 2.3 is implemented below. Again the numpy python library is used. The code is separated into sections: first Alice then Bob and finally Alice again in order to differentiate the subject who should execute the underlying commands.

–– pub_key-LWE.py

```
import numpy as np

n = 25
s = 72
q = 127

M = 0

A = np.random.randint(0, q+1, n)
e = np.random.randint(0, 5, n)

B = np.array([(A[i]*s+e[i])%q for i in range(n)])

print(f"\n———␣Alice␣———")
print(f"Message␣bit:␣\t\t{M}")
print(f"Public␣Key␣(A):\t\t{A}")
print(f"Public␣Key␣(B):\t\t{B}")
print(f"Error␣vector␣(e):\t{e}")
print(f"Secret␣key␣(s):\t\t{s}")
print(f"Prime␣number␣(q):\t{q}")

sample = np.random.randint(0, n, n//5)
u = 0
v = 0
for i in range(len(sample)):
    u += A[sample[i]]
    v += B[sample[i]]
u = u % q
```

```
v = (v + (int(q//2) * M)) % q

print(f"\n———␣BoB␣———")
print(f"Sample␣vector␣(sample):{sample})")
print(f"Cipher␣text␣(u):\t{u}")
print(f"Cipher␣text␣(v):\t{v}")

print(f"\n———␣Alice␣———")
res = (v − s*u) % q
if res>q/2:
    print(f"The␣message␣bit␣is␣a␣1")
else:
    print(f"The␣message␣bit␣is␣a␣0")
```

3.3 Key Exchange - Ring LWE

The code from which we started and made some modifications is (Buchanan 2023*d*). The following code is the implementation of the exchange key using Ring-LWE. Differently from the two previous cases, instead of using matrices, they use polynomials. Also in this case, as in the previous one, the document is divided into sections, where in each one there are the operations that Alice or Bob must carry out or the variables/secrets shared between the parts. At the end of the code, we check that the keys created, following the method explained in section 2.4, are equal.

—– key_ex-RLWE.py

```
import numpy as np
from numpy.polynomial import polynomial as p

def gen_poly(n,q):
    global xN_1
    l = 0
    poly = np.floor(np.random.normal(l, size=(n)))
    while (len(poly) != n):
        poly = np.floor(np.random.normal(l, size=(n)))
        poly = np.floor(p.polydiv(poly,xN_1)[1]%q)
    return poly

def extractError(shared):
    global u
    for i in range(len(u)):
        if (u[i] == 0):
            #Region 0 (0 —— q/4 and q/2 —— 3q/4)
                if (shared[i] >= q*0.125 and shared[i] < q*0.625):
                        shared[i] = 1
                else:
                        shared[i] = 0
        elif (u[i] == 1):
            #Region 1 (q/4 —— q/2 and 3q/4 —— q)
                if (shared[i] >= q*0.875 and shared[i] < q*0.375):
                        shared[i] = 0
                else:
                        shared[i] = 1
    return shared

# ——— SHARED VALUES ———
n = 15
q_exp = n
if n>50:
    q_exp = 32 # n is too big for computation
q = 2**q_exp-1 # 2^n-1

xN_1 = np.array([1] + [0] * (n-1) + [1]) # (x^n + 1)

# ——— ALICE ———
A = np.floor(np.random.random(size=(n))*q)%q
A = np.floor(p.polydiv(A,xN_1)[1])

sA = gen_poly(n,q)
eA = gen_poly(n,q)

bA = p.polymul(A,sA)%q
bA = np.floor(p.polydiv(sA,xN_1)[1])
bA = p.polyadd(bA,eA)%q

# ——— BOB ———
sB = gen_poly(n,q)
eB = gen_poly(n,q)

bB = p.polymul(A,sB)%q
bB = np.floor(p.polydiv(sB,xN_1)[1])
bB = p.polyadd(bB,eB)%q

u = np.array([0] * n)
for i in range(n-1):
    if (int(bB[i]/(q/4)) == 0): u[i] = 0
    elif (int(bB[i]/(q/2)) == 0): u[i] = 1
    elif (int(bB[i]/(3*q/4)) == 0): u[i] = 0
    elif (int(bB[i]/(q)) == 0): u[i] = 1

# ——— SHARED SECRETS ———
#Alice
sharedAlice = np.floor(p.polymul(sA,bB)%q)
sharedAlice = np.floor(p.polydiv(sharedAlice,xN_1)[1])%q

#Bob
sharedBob = np.floor(p.polymul(sB,bA)%q)
sharedBob = np.floor(p.polydiv(sharedBob,xN_1)[1])%q

# ——— REMOVE ERROR ———
sharedAlice = extractError(sharedAlice)
sharedBob = extractError(sharedBob)

print("——— Shared values ———")
print(f" n: {n}")
print(f" q (2^{q_exp}-1): {q}")
print(f" xN_1: {len(xN_1)} | {xN_1}")
print(f" A: {len(A)} | {A}")

print("\n——————— Alice ———————")
print(f" s: {len(sA)} | {sA}")
print(f" e: {len(eA)} | {eA}")
print(f" b: {len(bA)} | {bA}")

print("\n——————— Bob ———————")
print(f" e: {len(sB)} | {sB}")
print(f" e: {len(eB)} | {eB}")
print(f" b: {len(bB)} | {bB}")
print(f" u: {len(u)} | {u}")
print("\n------ Shared secret ------")
print(f" Shared Secret Alice: {len(sharedAlice)} | {sharedAlice}")
print(f" Shared Secret Bob:   {len(sharedBob)} | {sharedBob}")

print("\n\n------ Verification ------")
err = 0
for i in range(len(sharedBob)):
    if (sharedAlice[i] != sharedBob[i]):
        print(f" Error at index: {i}")
        err +=1
if err == 0:
        print(" Everything correct!")
```

4 Evaluation

As previously mentioned, the programming language used is Python 3.9 but it is possible to increase the performance of the code by developing a version written in the C language. This would result in greater speed and management of the variables but also an increase in the difficulty of writing the code.

Learning With Error

The program has been tested with various values of n, m, q and s and in all cases it works as it should. To prevent any errors, the *checkS*() function has been added, which is used to prevent the size and values of the secret vector from conforming to the other set variables.

Public Key - LWE

The code in question was the one modified the most from the original, as some structures or functions could be improved from a logical and comprehension point of view. The program has been tested several times, setting the value of the bit message M equal to 0 and 1 and using different values for the variables n, s and q.

Key Exchange - Ring LWE

The last program was the least modified compared to the others. Unlike the first two cases, this program fails to run if $q\_exp > 50$. This is due to the fact that the values of q grow exponentially with respect to q_exp. By changing the other variables, the code works perfectly. The major modification made to this program is that of dividing it into parts as in the algorithm previously explained in section 2.4.

5 Conclusions

By studying this topic and trying to put the acquired information into practice, it can be understood that Post Quantum Cryptography is a very vast topic and there are multiple uses of this relatively new technology. Currently, NIST has selected possible PQC algorithms that may become standards in the future (Alagic et al. 2020). Lattice-based Post Quantum Cryptography, considered in this paper, seems to be very promising, also considering its homomorphic characteristic. The Key Exchange method, using Ring-LWE, shown in this research is only one of the possible algorithms underlying Lattice-based Key Exchange as there are other complex methods, such as LWR and MLWE, not explored in this article. Surely in the coming years, there will be a change regarding the cryptographic methods used, especially those concerning Public Key and Key Exchange, abandoning the current ones, in order to prevent possible violations through quantum computers.

References

Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Kelsey, J., Liu, Y.-K., Miller, C., Moody, D., Peralta, R. et al. (2020), 'Status report on the second round of the nist post-quantum cryptography standardization process', *US Department of Commerce, NIST* **2**.

Alkim, E., Ducas, L., Pöppelmann, T. & Schwabe, P. (2016), Post-quantum key exchange-a new hope., *in* 'USENIX security symposium', Vol. 2016.

An, S. & Seo, S. C. (2020), 'Efficient parallel implementations of lwe-based post-quantum cryptosystems on graphics processing units', *Mathematics* **8**(10), 1781.

Armknecht, F., Boyd, C., Carr, C., Gjøsteen, K., Jäschke, A., Reuter, C. A. & Strand, M. (2015), 'A guide to fully homomorphic encryption', *Cryptology ePrint Archive* .

Bernstein, D. J. & Lange, T. (2017), 'Post-quantum cryptography', *Nature* **549**(7671), 188–194.

Bos, J. W., Costello, C., Naehrig, M. & Stebila, D. (2015), Post-quantum key exchange for the tls protocol from the ring learning with errors problem, *in* '2015 IEEE Symposium on Security and Privacy', IEEE, pp. 553–570.

Buchanan, W. J. (2023*a*), 'Learning with errors (lwe)', `https://asecuritysite.com/pqc/lwe`. Accessed: May 10, 2023.
URL: *https://asecuritysite.com/pqc/lwe*

Buchanan, W. J. (2023*b*), 'Learning with errors (lwe) and ring lwe', `https://asecuritysite.com/pqc/lwe_output`. Accessed: May 12, 2023.
URL: *https://asecuritysite.com/pqc/lwe_output*

Buchanan, W. J. (2023*c*), 'Public key encryption with learning with errors (lwe)', `https://asecuritysite.com/pqc/lwe2`. Accessed: May 12, 2023.
URL: *https://asecuritysite.com/pqc/lwe2*

Buchanan, W. J. (2023*d*), 'Ring learning with errors for key exchange (rlwe-kex)', `https://asecuritysite.com/encryption/lwe_ring`. Accessed: May 13, 2023.
URL: *https://asecuritysite.com/encryption/lwe_ring*

Chen, L., Chen, L., Jordan, S., Liu, Y.-K., Moody, D., Peralta, R., Perlner, R. A. & Smith-Tone, D. (2016), *Report on post-quantum cryptography*, Vol. 12, US Department of Commerce, National Institute of Standards and Technology . . . .

Cheon, J. H., Kim, D., Lee, J. & Song, Y. (2018), Lizard: Cut off the tail! a practical post-quantum public-key encryption from lwe and lwr, *in* 'Security and Cryptography for Networks: 11th International Conference, SCN 2018, Amalfi, Italy, September 5–7, 2018, Proceedings', Springer, pp. 160–177.

Ding, J., Xie, X. & Lin, X. (2012), 'A simple provably secure key exchange scheme based on the learning with errors problem', *Cryptology ePrint Archive* .

D'Anvers, J.-P., Karmakar, A., Sinha Roy, S. & Vercauteren, F. (2018), Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem, *in* 'Progress in Cryptology–AFRICACRYPT 2018: 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, Proceedings 10', Springer, pp. 282–305.

Fluhrer, S. (2016), 'Cryptanalysis of ring-lwe based key exchange with key share reuse', *Cryptology ePrint Archive* .

Gentry, C. (2009), Fully homomorphic encryption using ideal lattices, *in* 'Proceedings of the forty-first annual ACM symposium on Theory of computing', pp. 169–178.

Guo, S., Kamath, P., Rosen, A. & Sotiraki, K. (2022), 'Limits on the efficiency of (ring) lwe-based non-interactive key exchange', *Journal of Cryptology* **35**(1), 1.

Kumar, M. (2021), 'Quantum computing and post quantum cryptography', *International Journal of Innovative Research in Physics* **2**(4), 37–51.

Kumar, M. & Pattnaik, P. (2020), Post quantum cryptography(pqc) - an overview: (invited paper), *in* '2020 IEEE High Performance Extreme Computing Conference (HPEC)', pp. 1–9.

Niederhagen, R. & Waidner, M. (2017), 'Practical post-quantum cryptography', *Fraunhofer SIT* .

Oder, T. & Güneysu, T. (2019), Implementing the newhope-simple key exchange on low-cost fpgas, *in* 'Progress in Cryptology–LATINCRYPT 2017: 5th International Conference on Cryptology and Information Security in Latin America, Havana, Cuba, September 20–22, 2017, Revised Selected Papers 5', Springer, pp. 128–142.

Regev, O. (2010), 'The learning with errors problem', *Invited survey in CCC* **7**(30), 11.

Shor, P. W. (1994), Algorithms for quantum computation: discrete logarithms and factoring, *in* 'Proceedings 35th annual symposium on foundations of computer science', Ieee, pp. 124–134.

Stebila, D. (2018), 'Introduction to post-quantum cryptography and learning with errors [powerpoint slides]', *McMaster University, University of Waterloo* .

Stebila, D. & Mosca, M. (2017), Post-quantum key exchange for the internet and the open quantum safe project, *in* 'Selected Areas in Cryptography–SAC 2016: 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers', Springer, pp. 14–37.

Van Dijk, M., Gentry, C., Halevi, S. & Vaikuntanathan, V. (2010), Fully homomorphic encryption over the integers, *in* 'Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29', Springer, pp. 24–43.

Yi, X., Paulet, R., Bertino, E., Yi, X., Paulet, R. & Bertino, E. (2014), *Homomorphic encryption*, Springer.

AI Declaration

All material written in this paper is produced by Alessandro Ravizzotti and no AI has been used in the process. All the Figures have been made by Alessandro Ravizzotti, using Python. All the Python code was cited and not produced by AI.