Active Implementation of End-to-End Post-Quantum Encryption

Anton Tutoveanu

University of Wollongong
amt597@uowmail.edu.au

Abstract

Constant advancements in quantum computing bring closer the reality of current public key encryption schemes becoming computationally feasible to be broken. Many developers working in the industry are just finding out about this and will be rapid to look into changing their web applications to be secure in the quantum era. This paper presents a tried and tested construction for a quantum-resistant, end-to-end encryption scheme which has been implemented in an online web application. The implementation is shown to work well without significant impact on the performance time in comparison to its pre-quantum counterpart.

1. Introduction

As reported by NIST in their Report on Post-Quantum Cryptography [1] the existence of large scale quantum computers are expected to become viable in the next 15 years or so. This means that current cryptographic primitives such as Diffie-Hellman, elliptic curves and the RSA cryptosystem won't be sufficient enough to provide the security needed for privacy and confidentiality in digital communications.

While this is a popular topic in the cryptographic community only some developers in the industry have heard of the concerns that quantum computers will bring and are subsequently not prepared for making the transition to post-quantum security. Despite the estimates indicating still some years away, the goal is to start preparing early to ensure a smooth change over. The initial stage of progress has started with researchers and cryptographers designing new quantum-resistant algorithms to replace the current affected ones. These algorithms have been submitted to NIST's Post-Quantum Cryptography project [2] which has since been short-listed [3]. Researchers are now in the process of analysing, testing and evaluating these finalist algorithms for future standardised use.

This paper addresses the gap between a widely implemented security feature called end-to-end encryption and post-quantum era security. Outlined is the standard end-to-end encryption model and the specific application where this has been implemented. Quantum-resistance is covered with a brief background on quantum computation and how it can break current public key cryptosystems. An overview of the popular RSA public key exchange and the underlying hard problem of integer factorisation that is broken with Shor's algorithm in polynomial time is shown. Then, from going over how current pre-quantum PKE cryptography is not sufficient anymore, a promising replacment, called lattice-based cryptography, is introduced. Some of the hard mathematical problems in lattices are mentioned, such as the shortest vector problem, short integer solution problem and learning-with-errors. Afterwards the CRYSTALS-KYBER key exchange algorithm and its functionality is described. The construction for a complete implementation of a post-quantum secure transmission channel between a client and server is then specified in detail with codes made available for a JavaScript frontend (client) and Go language backend (server). An isolated time performance benchmark is also then listed in comparison to its 'pre-quantum' counterpart.

2. End-to-End Encryption

An additional layer of security in online communications between client and servers when related to web applications is a feature called end-to-end encryption (E2EE) [4]. End-to-end encryption is used by many applications and devices to provide the user with an extra assurance of security and data privacy on top of the protocols offered by TLS and IPSec. Internet banking, financial transactions, ecommerce, etc should all implement E2EE [5], [6], [7].

The standard model of E2EE as viewed between a client and server can be described as at least two end-devices that communicate via a completely secure transmission channel where all information is encrypted before it leaves one end-point and after receiving on the other end-point. The client and server both have the capabilities to encrypt data for sending and decrypt data that is received. This is useful in applications where the transmission channel isn't secure or as an extra layer of security on top of existing ones. It could be the case that a layer gets compromised so it also acts as a safety net. The two parties can confidently and securely communicate without worrying about revealing or transmitting data to other unauthorised or unintended parties. However, in some web applications or in cloud computing/storage, it is required that the information remain encrypted on the external server or end-device to maintain full and complete privacy of user data (no developer access). The standard client-server model can be extended to preserve the confidentiality of the data from when leaving the client and continuing onto the end-point. The server won't be able to decrypt or obtain any information about the plaintext data that was made by the client. In these infrastructures the external server storing or processing data from the user, must be kept hidden from people managing the end-device such as developers or system administrators.

In some cases the server might require to do processing on the user's data. As of end-2020 in the specific web application [8] that this end-to-end encryption is implemented, the model looks like this:

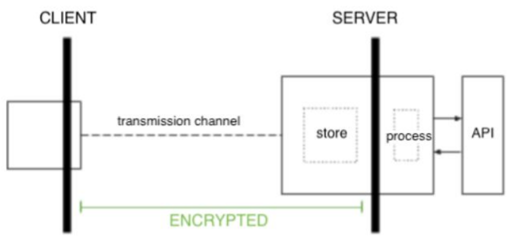


Fig. 1: Application Specific Model

User data is retrieved from an API which is subsequently processed into the format required by the application. Immediately after, it is encrypted and stored in a database to then be sent to a client upon request. Is it possible to preserve data confidentiality while also achieving server-processing functionality? Yes. [9] However this is yet to be considered for the E2EE construction for this application. Regardless, it is inevitable that raw data is received by the server from the API at some initial point. The goal then is to minimise the existence of this sort of data while on the server. As such the design settles on encrypting 'data-at-rest' [10].

Acknowledging a range of different applications and their requirements, both models described earlier and the model in *Fig. 1* can be implemented with tweaks (if needed) of the codes that are available and referenced in this paper for the construction of an end-to-end encryption scheme.

3. Quantum-Resistance

The security of current pre-quantum encryption algorithms rely on the hard mathematical problems of integer factorisation and discrete logarithms [11]. With large enough prime numbers and over carefully chosen number sets these problems cannot be solved within a feasible amount of time by classical computers. This is what currently makes public key encryption methods such as RSA and elliptic-curves cryptographically secure.

Classical computation is based on two binary states, being 0 and 1. Quantum computation is also based on these two states and an additional state referred to as superposition [12]. A state of superposition allows for the representation of many logical states simultaneously. Classical information units are referred to as bits, while quantum information units are called qubits. Quantum computing is realised with the construction of quantum circuits which are made up of quantum gates in addition to classical logic control. These circuits still ultimately work with 0's and 1's but its these specialised quantum gates which facilitate the property of superposition that allow one to write quantum algorithms that speed up computational problem solving. A quantum algorithm can consider multiple logical states at once which lowers the time complexity of the problem reaching the output solution. This is essentially what threatens public key encryption schemes. The hard mathematical problems underlying them that weren't previously able to be feasibly solved by classical computers are now theoretically able to be solved by quantum computers in a short amount of time.

A prominent quantum algorithm called Shor's algorithm [13] can solve the integer factorisation problem in polynomial time. The RSA public key cryptosystem that is widely used for encryption and distributing symmetric keys between two parties relies on this specific problem for its security [14]. With a sufficiently large enough quantum computer running Shor's algorithm, this cryptosystem is broken. RSA typically uses 1024-4096 size keys (which is the size in bits of the modulus n), but NIST recommends using upwards of 2048 bits [15]. RSA is quite slow and is usually used in conjunction with a symmetric key algorithm (like AES) to provide a secure transmission channel between parties. In practise it is primarily used in a hybrid mode for key distribution.

RSA works with each party generating a public and private key pair. An entity wishing to send an encrypted message to another entity known as the receiver must encrypt the message with the receiver's public key. Then after sending, only the receiver is able to decrypt the message with their private key that only they know of. If one can somehow calculate the private key linked to the public key that is known to everyone, then all encrypted messages are able to be decrypted hence breaking the cryptosystem.

To generate an RSA public and private key pair:

Alg. 1: RSA Key Generation

```
1. Generate two large and distinct primes, $p$ and $q$.
2. Compute $n = pq$.
3. Compute $\phi(n) = (p-1)(q-1)$.
4. Choose $e$ such that $\gcd(e, \phi(n)) = 1$.
5. Compute $d = e^{-1} \bmod \phi(n)$.

Public key $\leftarrow (e, n)$
Private key $\leftarrow (d, n)$
```

The best approach to breaking RSA encryption would be to first factor n (which is public) to find the two large primes p and q. From there $\phi(n)$ can be calculated, and then d (since e is also public). Due to this, computing the RSA decryption exponent d from the public key (e, n) and the problem of factoring n can be considered as computationally equivalent [16].

To break RSA using Shor's algorithm, we can assume that n is the product of two prime integers. The algorithm is as follows:

Alg. 2: Shor's Algorithm (classical section) [13][17]

```
1. Choose a random integer $x$ such that $1 < x < n$.
2. Compute gcd($x$,$n$).
3. If gcd($x$,$n$) $\neq$ 1, then $x$ is a factor of $n$. (end)
4. If gcd($x$,$n$) = 1, then use the quantum
period-finding subroutine to find the smallest integer
$r$ such that $x^r \equiv 1 \bmod n$.
5. If $r$ is odd, go back to step 1.
6. If $x^{r/2} \equiv -1 \bmod n$, go back to step 1.
7. Otherwise gcd($x^{r/2}$+1,$n$) and gcd($x^{r/2}$-1,$n$) are factors
of $n$.
```

Steps 1-3 and 5-7 can be executed sufficiently on a classical computer using implementations of basic arithmetic operations, the Euclidean Algorithm and simple if statements. In the classical section, if the random guess of x is not a factor of n, solving the factorisation problem can be boiled down to finding the order of the element x which is r in $x^r \equiv 1 \bmod n$ (step 4), this is the period-finding problem. To solve this problem Shor utilised quantum phase estimations and quantum fourier transforms [18], these however won't be explained here.

4. Lattices

Now that its shown how current pre-quantum algorithms won't be resistant to future quantum computing attacks, what makes other potential future encryption algorithms resistant and safe in the quantum era? A promising and popular contender for post-quantum cryptographic schemes are lattice-based cryptosystems [19]. A lattice is a set of points in n-dimensional space with a periodic structure [20]. It is also known as the set of all integer linear combinations of basis vectors $b_1, \ldots, b_n \in \mathbb{R}^n$, in notation as:

$$L(b_1, \ldots, b_n) = \left\{\sum_{i=1}^{n} x_i b_i : x_i \in \mathbb{Z}\right\}$$

For a 2-dimensional lattice, only 2 basis vectors are needed to generate the entire lattice over a finite or infinite space.

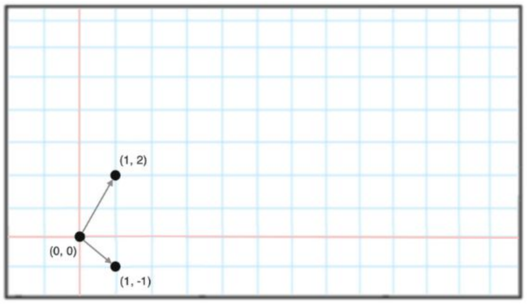


Fig. 2a: 2D Lattice Basis

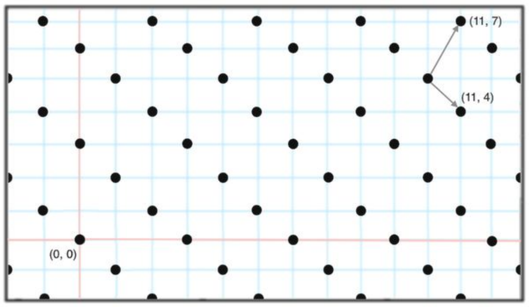


Fig. 2b: 2D Lattice from Basis

A lattice can be generated by many different possible basis vector sets [21]. In general, a good basis is one where the vectors are short in distance and are nearly orthogonal (or almost perpendicular). A bad basis is where the vectors are long, very askew and too close together. It should be mentioned that any n number of vectors cannot be a basis to make a specific lattice as some points will not be generated by integer linear combinations of those n number of vectors. Examples of a good, bad and invalid basis in a lattice are shown below.

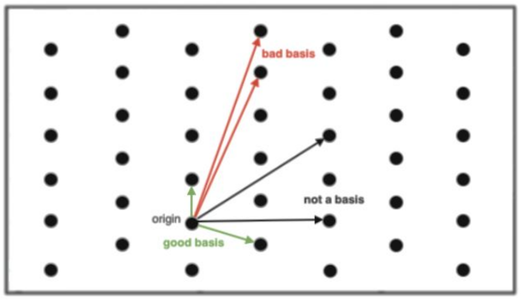


Fig. 3: Good, Bad and Invalid Bases

4.1 Shortest Vector Problem

Lattices have many hard mathematical problems that are deemed to be one-way functions and could be used to construct quantum-resistant cryptosystems. The most well-known is the shortest vector problem (SVP) which is hard in worst-case [22]. SVP is defined as: when given any arbitrary lattice basis (usually bad) as input, find the shortest non-zero vector in the lattice generated. The length of each vector is measured by a norm which is often the Euclidean norm:

$$L^2 = ||v||_2 = \sqrt{x_1^2 + \ldots + x_n^2},$$

where $v = (x_1, \ldots, x_n)$ represents the vector's coordinates or cartesian values on an n-dimensional plane. The shortest nonzero vector, also known as the successive minima, is denoted by $\lambda_1(L)$. In some lattice cases there can be more than one shortest vector, however only one value will exist for the shortest distance. In a more general definition, $\lambda_k(L)$ represents the value of the smallest radius in a circular area containing k linearly independent vectors. A variant of the shortest vector problem SVP$_\gamma$ uses γ as an approximation factor that can be adjusted to make the problem easier or harder. It is defined as: when given any arbitrary lattice basis (usually bad) as input, find a vector in the lattice generated whose length is equal to or less than $\gamma \cdot \lambda_1(L)$. The larger γ is, the easier the problem is to solve as there are more vectors existing as valid solutions in the given circular area of a lattice.

4.2 Short Integer Solution Problem

Another known hard problem in lattices is the short integer solution problem [23], [24]. The SIS problem is shown to be secure in average case, which is needed in cryptographic constructions for randomised instances. SIS$_{n,q,m,\beta}$ is defined as: given an $m\times n$ matrix A that contains random entries from Z_q, find a vector x whose norm is bounded by β and which results in a zero vector when taking the matrix-vector product of A and x.

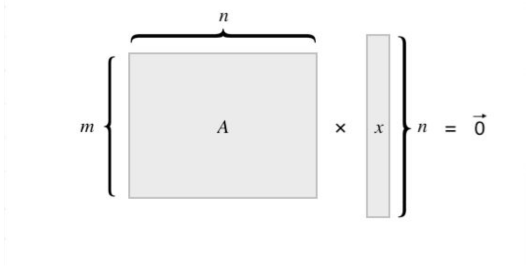


Fig. 4: Short Integer Solution Problem

Denoted as $f_A(x) = Ax = \vec{0}$ where $A \in \mathbb{Z}_q^{m\times n}$, $x \in \mathbb{Z}^n$, $||x|| \le \beta$ and matrix-vector mulitplication [25] is:

$$Ax = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Ax is essentially a system of linear equations where the vector $(x_1, x_2, ..., x_n)$ are values that satisfy all the equations in the system. This problem is able to be solved using Gaussian elimination [26] which can find a solution for any arbitrary vector x, but will give a relatively large value. But for a 'short' solution for vector x, meaning the entries are -1, 0, 1, etc, this problem is difficult. If the parameters are large enough, then the SIS problem makes a one-way function. It is easy to compute the result of $f_A(x) = Ax = y(mod\, q)$, but it is difficult to do the inverse operation where given A and y, find a 'short' vector x.

Relating the short integer solution problem back to lattices, the $A \in \mathbb{Z}_q^{m\times n}$ matrix can be used to construct a 'q-ary' or modulo lattice. The $\mathbb{Z}_q^n$ modulo lattice can then be used in a tiled or block-like manner to generate a full unbounded lattice. Because of this, generating a random lattice can be reduced to randomly generating a matrix $A \leftarrow_\$ \mathbb{Z}_q^{m\times n}$ where $\leftarrow_\$$ denotes random uniform sampling, n is the dimension of the lattice and m is the number of equations. Using the matrix A with the SIS problem: $Ax = \vec{0}$, results in a system of linear equation(s) that all pass through the origin on a cartesian plane of n-dimension. Below is a 2-dimensional example for a matrix $A \in \mathbb{Z}_7^{1\times 2}$ which is modulo 7:

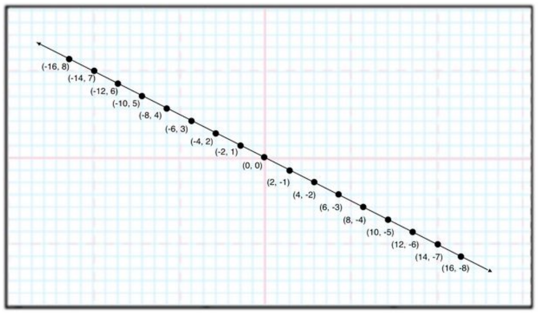


Fig. 5a: Linear Equation from Matrix A

The matrix is $A = [1 \quad 2]$ which corresponds to the linear equation $x + 2y = 0$ (plotted above). Taking the modulus q of all integer points that reside on the line and plotting them on the plane will give you the q-ary lattice base $\mathbb{Z}_q^n$ (not to be confused with lattice basis).

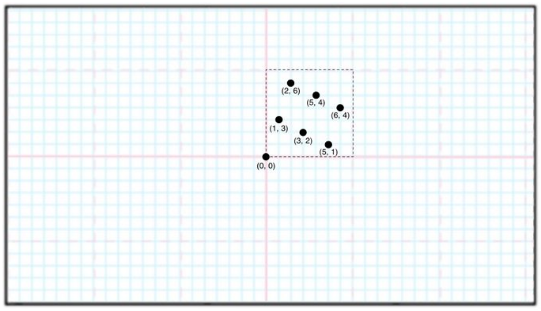


Fig. 5b: Q-ary Lattice Base

Using the q-ary lattice base as a repeating tile, the rest of the lattice can be generated [27].

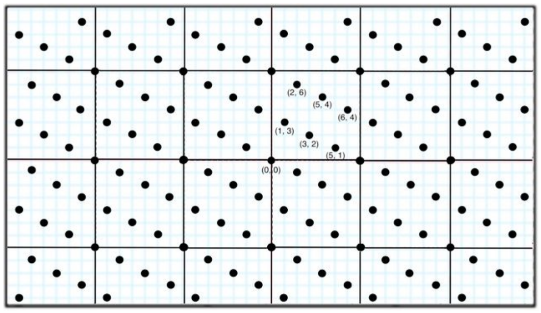


Fig. 5c: Q-ary Lattice from Base

It should be mentioned that there needs to be some constraints on the $m\times n$ variables of matrix A for this to work. Particularly that $m < n$: there should not be more equations m than the dimension n. Another similar constraint also seen is: $n > m \log q$ [28].

4.3 Learning With Errors Problem

Adding on from the short integer solution problem leads to the learning with errors (LWE) problem [29], [30]. This problem is used to construct a public key cryptosystem. Let $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ be the quotient group where mod 1 of any real number will result with a value in the segment [0, 1). Let $s \in \mathbb{Z}_q^n$ be a fixed 'secret' vector and $A \leftarrow_\$ \mathbb{Z}_q^{m \times n}$ be a public matrix that represents a system of linear equations. Let χ be a probability distribution (typically a Gaussian-like distribution) over $\mathbb{T}$. Some noise or error is denoted by $e \in \mathbb{T}$ from χ. Calculate $B = \frac{1}{q} \cdot As + e$. $\mathrm{LWE}_{q,\chi}$ is defined as when given access to many samples of $(a_i, b_i) \in A, B$, find $s \in \mathbb{Z}_q^n$ that satisfies the system of equations. This problem is hard in average case like with the short integer solution problem. To visualise how the problem looks like:

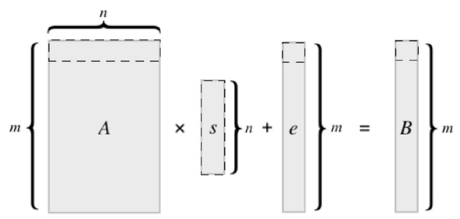


Fig. 6: Learning with Errors

The dotted lines represent what a sample from $(a_i, b_i) \in A, B$ corresponds to in the system of linear equations shown above. Using the LWE problem to construct PKE, the public key will be (A, B) and the secret key will be s. To encrypt:

$$(u, v) \leftarrow \{ u : u = \sum a_i, v : v = \sum b_i - \frac{q}{2} \cdot M \}$$

where $M \in \{0, 1\}$ is a single bit and (u, v) is the ciphertext pair which encrypts the bit. To decrypt:

$$d = v - su \, (mod \, q)$$

where $d < \frac{q}{2} \Rightarrow M = 0$, or if $d > \frac{q}{2} \Rightarrow M = 1$. This encryption works bit-by-bit giving "1-bit security". An adversary can't distinguish between a bit that was encrypted in comparison with a randomly chosen one by more than a negligible probability. By having a 1-bit secure PKE, you can then construct an IND-CPA secure public key encryption scheme [31]. For the implementation of the post-quantum end-to-end encryption scheme in this paper, the NIST PQC finalist CRYSTALS-KYBER is used.

5. CRYSTALS-KYBER

CRYSTALS-KYBER (version 3) is an IND-CCA2 post-quantum key exchange protocol. This protocol is used to securely establish symmetric keys between two parties using a key-encapsulation mechanism (KEM) [32]. It is based on the learning with errors problem in module lattices (M-LWE). Kyber's original design comes in 512, 768, 1024 security strengths. Kyber-768 will distribute a 256-bit symmetric key between two parties which is a sufficient key size to be used with symmetric key encryption (eg. AES).

5.1 Key Encapsulation Mechanism

Starting with the high-level functionality of Kyber is the Key Encapsulation Mechanism (KEM) which contains three core functions:

1. *Key Generation*
   $(pk, sk) \leftarrow KeyGen()$

2. *Encrypt/Encapsulate*
   $(c, ss) \leftarrow Encap.(pk)$

3. *Decrypt/Decapsulate*
   $ss \leftarrow Decap.(c, sk)$

These represent the typical functionality of public key encryption. A party wanting to receive an encrypted message, will generate a public and private key pair, the private key is kept secret. The public key is known and used by senders to encrypt their message into a ciphertext. The ciphertext is then sent to the receiver where only they can decrypt it with their private key. The high-level description of this protocol is more specific to the context of key exchanging and is described as a KEM. The intended symmetric key is 'encapsulated' by encrypting with the public key. Corresponding with decrypting a ciphertext in PKE, the key is then 'decapsulated' by decryption with the private key.

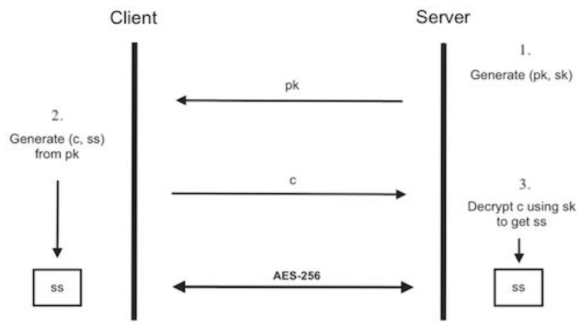


Fig. 7: Key Encapsulation Mechanism

The exchange between the client and the server is shown above. In web based applications or server-client type communication, the server typically generates the initial public/private key pair and sends the public key *pk* to the client. The client then generates a symmetric key *ss* (for 'shared secret') and an encrypted version of this key *c* (for 'cipher') with the public key from the server. The encrypted symmetric key is sent to the server. The server then decrypts this with the initial private key and obtains the same symmetric key as client. It can also work vice-versa depending on which party generates the initial public/private key pair. After the client and server both have the symmetric key, data can be securely sent over the channel using the encryption standard AES-256. AES with a key length of 256 bits is secure against future anticipated quantum computer attacks [33]. For the rest of this document, the Kyber-768 parameter set is analysed and detailed.

5.2 Key Generation

The first step in generating a new public and private key pair is generating random seeds for the public matrix *A*, the secret *s* and the noise *e*. This is done by reading random byte values {0, ..., 255} into an array 32 in length. The 32 random byte array is then hashed using SHA3-512 to produce a 64 byte output digest. This output is then spilt into the public seed ϱ (bytes 0-31) and the noise seed σ (bytes 32-63).

I. SEED GENERATION

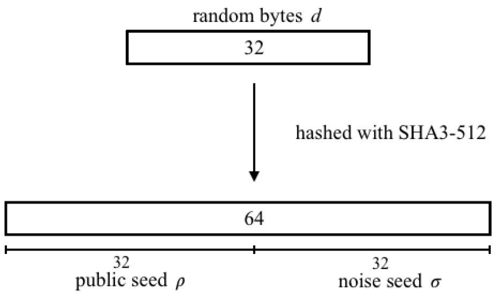


Fig. 8a: Generating Random Seeds

The public seed is now used to generate public matrix *A*. For Kyber-768, *A* is a 3×3 matrix that contains 256 length mod *q* array of coefficients, one for each entry (9 total). To generate each entry of the matrix, the public seed is hashed with SHAKE-128 along with its index in the matrix (*i, j*) concatenated together. SHAKE-128 is an XOF or extendable-output function [34]. An XOF works similar to a usual hash function but allows the output to be of a varying length. The output in this implementation is a 504 byte array. The array from the XOF is then put through a sampling function that accepts or rejections values calculated based on parsing the given byte array. The sampling function is designed to take any length input byte array and output a polynomial mod *q* of length 256. This is now entry (*i, j*) of matrix *A*. The generation of matrix *A* is deterministic, so when given the public seed ϱ, the matrix from this should always be the same upon each generation with the same code.

II. MATRIX *A* GENERATION

$$A^{3\times 3} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

For each $a_{ij} \in \hat{a}_0 + \hat{a}_1 X + \ldots + \hat{a}_{255} X^{255}$:

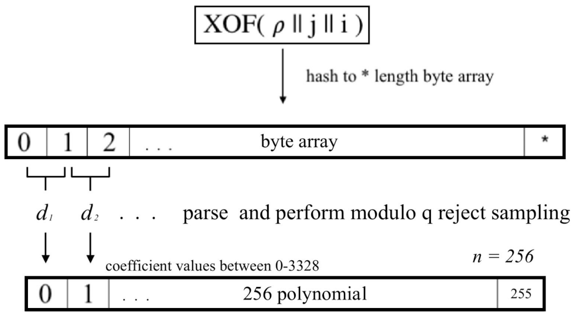

XOF(ρ || j || i)
hash to * length byte array
0 1 2 . . . byte array *
d1 d2 . . . parse and perform modulo q reject sampling
coefficient values between 0-3328
n = 256
0 1 . . . 256 polynomial 255

Fig. 8b: Generating Public Matrix A Entries

Now that matrix *A* has entries $a_{i,j} \in \mathbb{Z}_q^n$, it is done initialising. Secret vectors *s* and noise vectors *e*, need to be sampled. This is first done by using a pseudo-random function (PRF) which takes the noise seed σ and a nonce value *N*, which is a integer value that is simply incremented each time. The output is a byte array $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in B^{64\eta}$ where $\eta = 2$, giving a length of 128. This byte array is then input to a function which generates a polynomial sampled from a centered binomial distribution (CBD). First the byte array is converted to an array of bits, which results in a 1024 length bit array. Each 4 bits in the array is then reduced to a single value $f_i \in \{-2, -1, 0, 1, 2\}$ to give a 256 length output array. This is done by computing $f_i = a - b$, where $a = \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$ and $b = \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$. Secret *s* and noise *e* are both sampled in the same way.

III. SECRET AND NOISE SAMPLING

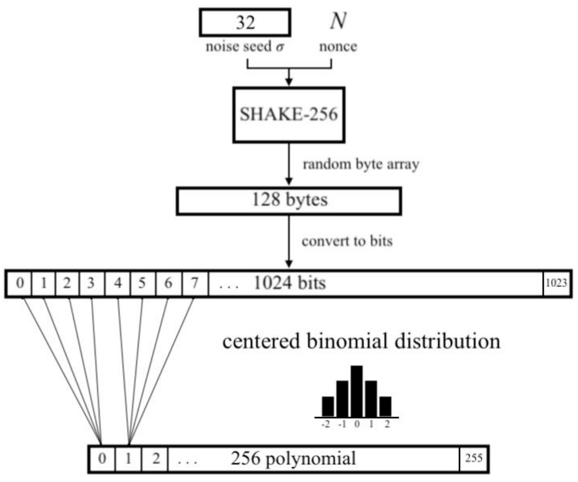

32
noise seed σ
N
nonce
SHAKE-256
random byte array
128 bytes
convert to bits
0 1 2 3 4 5 6 7 . . . 1024 bits 1023
centered binomial distribution
-2 -1 0 1 2
0 1 2 . . . 256 polynomial 255

Fig. 8c: Generating Secret and Noise

Now the secret s and noise e vectors go through a number theoretic transform (NTT) [35]. NTT significantly speeds up the multiplication of large polynomial rings of which is needed when doing key computations. Each entry in the matrix A and the vectors s and e is made of a polynomial ring denoted by $\mathbb{Z}_q[X]/(X^n+1)$ where $q = 3329$ and $n = 256$. The initial polynomial ring is able to be reduced to smaller ones. Efficient multiplications are then possible using these smaller degree polynomials. In Kyber, you first start with the reduction polynomial $(X^{256} + 1)$ and reduce it into two by finding:

$$(X^{256} + 1) = (X^{128} + \alpha)(X^{128} - \alpha)$$

this results in $\alpha^2 = -1$, squared to get $\alpha^4 = 1$, which then gives $\alpha = \sqrt[4]{1}$. Reducing further:

$$(X^{128} + \alpha) = (X^{64} + \beta_1)(X^{64} - \beta_1)$$

results in $\alpha = -\beta_1^2$ rearranging to get $\beta_1 = \sqrt{-\alpha}$ and since $\alpha^2 = -1$, substitution gives $\beta_1 = \sqrt{\alpha^3}$. Doing the same for $(X^{128} - \alpha)$ will result in $\beta_2 = \sqrt{\alpha}$. A pattern emerges where for every subsequent reduction polynomial, the values will generally be $y_1 = \sqrt{x^3}$ and $y_2 = \sqrt{x}$ of the previous reduction polynomial. For a more succinct definition, the symbol zeta is introduced to represent the k-th primitive root of 1 denoted as: $\zeta_k = \sqrt[k]{1}$. The k-th primitive root of unity is some number ζ which holds: $\zeta^k = 1\,(mod\,q)$ where $\zeta^l \neq 1\,(mod\,q),\ 1 \leq l < k$. Following this, the previous values turn into $\beta_1 = \sqrt{\alpha^3} = \sqrt{\zeta_4^3} = \zeta_8^3$ and $\beta_2 = \sqrt{\alpha} = \sqrt{\zeta_4} = \zeta_8$. Reducing the factors will be:

$$(X^{128} + \zeta_4)(X^{128} - \zeta_4)$$
$$\downarrow$$
$$(X^{64} + \zeta_8^3)(X^{64} - \zeta_8^3)(X^{64} + \zeta_8)(X^{64} - \zeta_8)$$

This reduction is typically continued until small degree polynomials are reached. When starting with a polynomial of $(X^{256} + 1)$, the last zeta value will be the 512-th primitive root of 1, denoted ζ_{512}. In Kyber with $q = 3329$, there doesn't exist a primitive 512th root of unity, so the 256th one is taken, giving $\zeta = 17$. This reduction can then be represented by:

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

$$(X^2 - \zeta)(X^2 - \zeta^3)(X^2 - \zeta^5)\ldots(X^2 - \zeta^{253})(X^2 - \zeta^{255})$$

After reduction we now have a vector of polynomials that are ready to be multiplied in a point-wise fashion using modular reductions such as Barrett and Montgomery reductions for efficient modular arithmetic calculations. After multiplication is complete, an inverse NTT is applied to bring the polynomials back to the standard form to continue the rest of the computations.

The key defining computation of $As + e$ for the public key generation is done. This outputs a 1×3 vector of n-degree polynomials mod q. This is then encoded with a function that serializes the polynomials into byte arrays, concatenates them and then appends the public seed ϱ. The result is a public key of length 384×3 + 32 = 1184 bytes. The secret s vector is then also encoded to make a byte array of length 384×3 = 1152 bytes, which will be the private key.

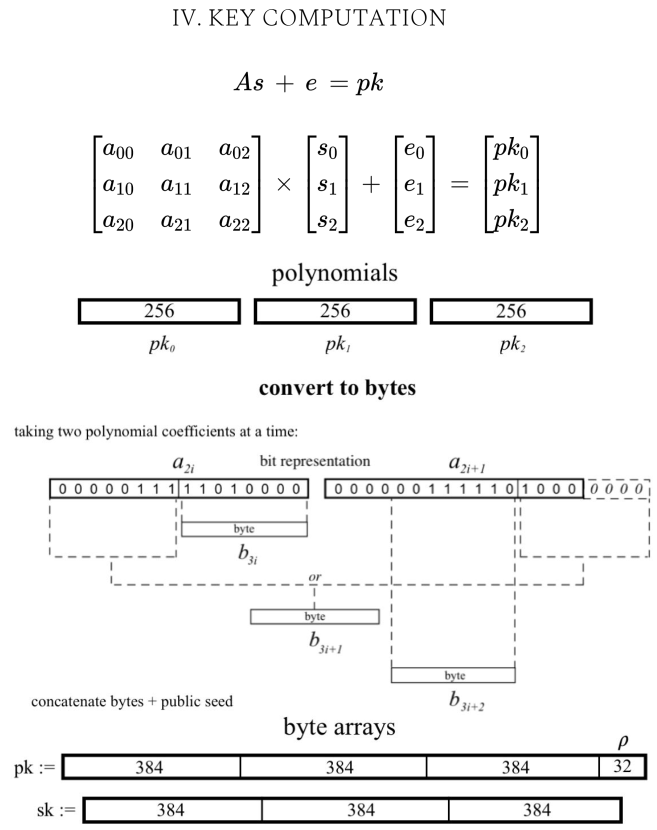


Fig. 8d: Computing Public and Private Keys

The public and private keys are now ready for IND-CPA encryption/decryption.

5.3 Encapsulation

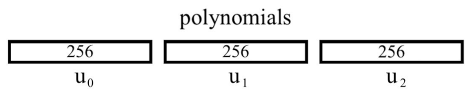
Since the main functionality of Kyber is to distribute keys securely between two parties, the encryption function is more of an encapsulation of a shared secret (symmetric key). For this, the public key is taken along with a 32 byte message and random 32 byte array referred to as 'coins' due to the 0's and 1's being an analogy for tosses of coins being random. The output will be a ciphertext byte array of length 1088.

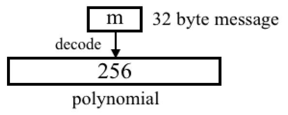
The public key which is in byte array format needs to be decoded back into polynomial form along with the public seed ϱ (kept in byte format). This is just the inverse of the encoding function that converts polynomials to bytes shown in IV. KEY COMPUTATION. Then, with the public seed ϱ the transpose public matrix A^T is reconstructed similar to II. MATRIX *A* GENERATION except *j* and *i* are switched around in order to produce the transpose of the matrix necessary for inverse calculation. Now a 1×3 vector of polynomials *r* is sampled according to III. SECRET AND NOISE SAMPLING. Another 1×3 vector of polynomials e_1 is sampled in the same way. Sampled again is one polynomial which is then assigned to e_2. Next, the number theoretic transform is applied to polynomial vector *r* to prepare it for efficient multiplication. Now the ciphertext pair (c_1, c_2) are calculated and encoded to form the final ciphertext that becomes the encapsulated symmetric key.

V. ENCRYPTION

$$A^T r + e_1 = u$$

$$\begin{bmatrix} a_{00} & a_{10} & a_{20} \\ a_{01} & a_{11} & a_{21} \\ a_{02} & a_{12} & a_{22} \end{bmatrix} \times \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} + \begin{bmatrix} e_0 \\ e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix}$$





$$pk^T r + e_2 + m = v$$

$$\begin{bmatrix} pk_0 & pk_1 & pk_2 \end{bmatrix} \times \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} + e_2 + m = v$$

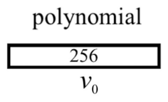


Fig 9: Encrypting Message

The polynomials of *u* and *v* are then compressed by taking each coefficient individually and discarding some of their low-order bits to make the ciphertext sizes smaller. The result is then concatenated together after encoding to byte array form and is sent as the ciphertext.

5.4 Decapsulation

Decapsulation of the symmetric key is then performed by the initiating party (typically the server) by taking their private key from their generated key pair and the received ciphertext from the client. After decoding and decompressing (*u*, *v*) back from the ciphertext, the original message *m*, is calculated by $v - s^T u$ where m=1 if $v - s^T u$ is closer to $\lceil q/2 \rfloor$ than to 0, and m=0 if otherwise. Rounding is denoted by $\lceil x \rfloor$ where x is any decimal point number and rounding is to the closest integer with 0.5 values rounding up (ie: $\lceil 3.4 \rfloor \Rightarrow 3$, $\lceil 3.5 \rfloor \Rightarrow 4$, $\lceil 3.6 \rfloor \Rightarrow 4$).

VI. DECRYPTION

$$v - \begin{bmatrix} s_0 & s_1 & s_2 \end{bmatrix} \times \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix} = m \in \{0, 1\}^{256}$$

Fig 10: Decrypting Message

The message bit array is then encoded to a byte array by taking 8 bits at a time and converting them to byte form giving a 32 length byte array message.

5.5 Fujisaki-Okomoto Transform

The scheme described above is only an IND-CPA secure construction. To make it CCA2 secure, the scheme goes through a Fujisaki-Okamoto transform. This requires only the addition of some hash functions, and can be proven secure in the random oracle model [36]. How the transform is implemented in the key exchange algorithm is illustrated below. Taking (*pk*, *sk*) from the IND-CPA scheme, the new public and private keys become:

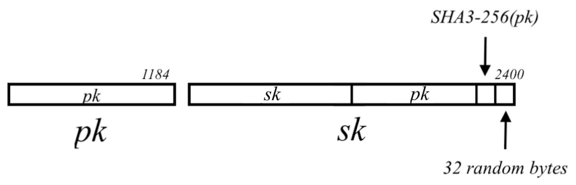


Fig. 11a: CCA2 KeyGen

To then produce a random 32 byte array representing the symmetric key and its corrreponding encapsulation/ciphertext:

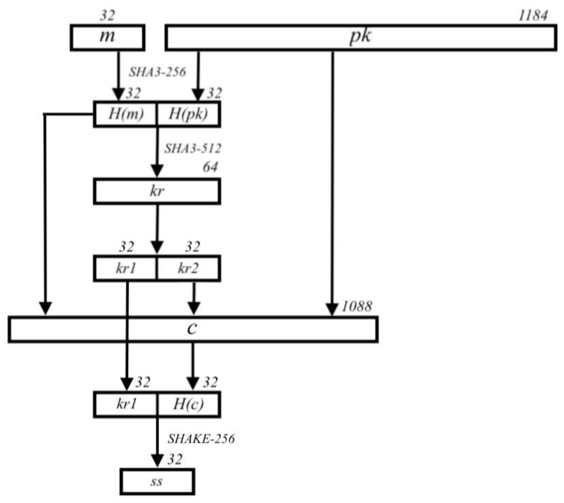


Fig. 11b: CCA2 Encryption

To decapsulate the ciphertext and retrieve the same 32 byte symmetric key:

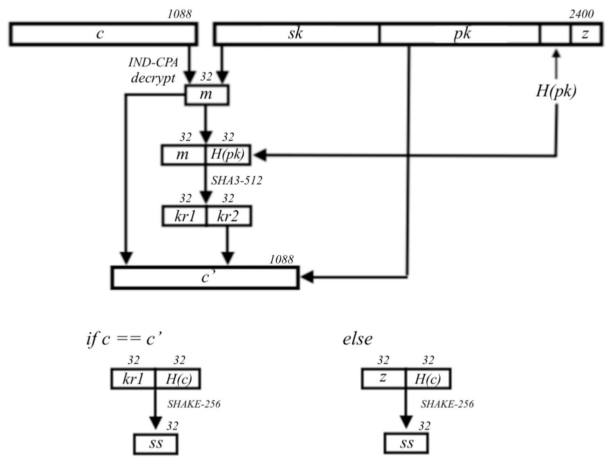


Fig. 11c: CCA2 Decryption

The scheme is now providing security in an IND-CCA2 model. This ends the description of the Kyber-768 post-quanum key exchange algorithm. Now, specifics of the implementation in the application will be covered.

6. Key Storage

Upon initialisation of the system, the server generates a public and private key pair by Kyber's *KeyGen()*. The client-side then has the server's public key hardcoded on the frontend code to save the need of the server sending it each time. The server keeps it's private key hidden on the server in a file. When a client tries to authenticate for the first time, a new user is created with the platform. A random symmetric key is generated which is then stored in user's database entry. This symmetric key remains persistent throughout the life of the user's account. To send the user this key, the client requests it via the Kyber key exchange protocol. A temporary symmetric key is generated by the client with *Encap.(pk)*. After sending the encapsulated key *c*, to the server for decapsulation, the server now has the same temporary symmetric key as the client-user. This temporary key is then used to encrypt the main symmetric key stored on the server system's database using AES-256 as a sort of hybrid encryption scheme for secure key distribution. This same establishment protocol is initiated again with the main key being sent out to the user if:

a) they lose the key on their end-device or,
b) log in from a new device or browser.

In most cases the main symmetric key stays persistent on the end-device. The client will store the symmetric key locally in their web browser with `window.localStorage` [37]. At this point in development, all user symmetric keys will be stored in the backend database, this means developers and anyone with root-access to the backend server is able to view the user's symmetric keys and decrypt stored ciphertexts. Future development of the application will look into solutions that prevent access to user keys from the internal programming team for further security.

7. Server Implementation

This implementation is for a backend written in Go language. An implemented version of Kyber called Kyber-K2SO by Nadim Kobessi [38] is used as a base code for the server implementation. The Kyber-768 security level is used. The code has since been updated to version 3 of Kyber and is available here [39].

https://github.com/antontutoveanu/kyber-k2so

The API is simple to use with just needing to import the module and using the functions as:

```
sk, pk, _ := kyberk2so.KemKeypair768()
c, ssA, _ := kyberk2so.KemEncrypt768(pk)
ssB, _ := kyberk2so.KemDecrypt768(c, sk)
```

AES-256 Go crypto library [40] is also used for the symmetric key encryption/decryption for the server-side. The block mode used in this AES is CBC mode and the padding standard used is PKCS7. Exact use of the encrypting code can be seen here as an example [41].

8. Client Implementation

All client-side websites are made being compatible with JavaScript. Especially a lot of web apps are written in JavaScript based frameworks such as React, Vue.js, etc [42]. For this application, the frontend is built with React. NPM [43] is a supported software registry containing many code packages from developers all over the world that contribute free, open source code to assist others with developing their projects. This registry is utilised to bring Kyber-768 easily into the React frontend application. The JavaScript code has been setup into an NPM package which is able to be imported and used for key exchanging. Only the 768 parameter set of Kyber is available in JavaScript at this point as it is sufficient enough to construct a secure transmission channel with the backend server (which is also running Kyber-768). The code can be viewed in the first link, while the second link is for the NPM package version:

https://github.com/antontutoveanu/crystals-kyber-javascript

https://www.npmjs.com/package/crystals-kyber

Using the JavaScript code via NPM in React is simple as with the Go implementation API:

```
var pk_sk = K768_KeyGen();
var pk = pk_sk[0];
var sk = pk_sk[1];

var c_ss = K768_Encrypt(pk);
var c = c_ss[0];
var ss1 = c_ss[1];

var ss2 = K768_Decrypt(c,sk);
```

After the symmetric key is established, the aes-js npm package library [44] is used with CBC block mode and the PKCS7 padding scheme [45] to encrypt/decrypt data to and from the server. By having key exchange and encryption/decryption possible on both the client-side (frontend) and the server-side (backend), this leads to an end-to-end encryption implementation for online and web-based application platforms.

9. Verification of Code

Since these are all new implementations based off the original submitted C code to NIST PQC standardisation. To check these against each other to show they produce the same results is important. The original Kyber code (written in C) comes with test run case files with determinstically generated executions of itself. These can then be used to ensure compatibility across other implementations. Testing the Go implementation is already confirmed using `go test -v`. This was done by taking the *ss*, *sk*, *c* values of each test run from the original code and then passing it to the Go code to perform:

```
ssB, _ := kyberk2so.KemDecrypt768(c, sk)
```

Then, a simple check whether the produced `ssB` is the same as the *ss* value from the original implementation is enough to confirm a successful test case run. This is run 100 times to test all cases in the provided `PQCkemKAT_2400.rsp` file. The JavaScript test function is written in a similar way testing the Kyber-768 outputs. The test is run for all 100 runs same as in `PQCkemKAT_2400.rsp` for the Go code. All tests are successful and are available in the GitHub repositories to replicate the same results. Additionally, in the application, the decryption seems to work consistently on the client-side showing that the whole key exchange and encryption process is working as should.

10. Performance

The application currently only needs to encrypt relatively low amounts of data. A typical user account will have 150,000 bytes (0.15mb) of data that needs to be decrypted. Since the server's public key is already hardcoded on the frontend, there is no need to send it out which reduces some performance time. The performance test carried out will include the client requesting the main symmetric key (the key re-establishment protocol) and the time taken to decrypt all data being sent from the server onto the frontend. The data recorded is the isolated running time for the specific key exchange and decryption code executed on a local device. This factors out the varying internet speeds and connection lag from a live-online performance for more consistent and accurate run-time results. Simple benchmarks were taken with `performance.now()` for JavaScript and `time.Now()` for Go.

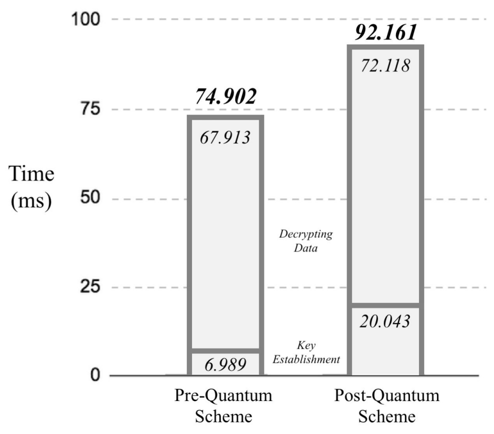


Fig. 12: Benchmarks

The results obtained show that there is no significant increase in run-time of the cryptographic processes of the end-to-end encryption. The post-quantum setup will only add an extra 23% of the time the pre-quantum counterpart will take. This is quite quick in the context of web applications and can be considered negligible. Key establishment is performed with RSA-2048 [46][47] (resp. KYBER-768) for pre-quantum (resp. post-quantum) setup. Pre-quantum symmetric encryption/decryption is performed with AES-128 while AES-256 is used for post-quantum setup. This is just one comparison between implemented codes of a pre-quantum and post-quantum schemes. Specifics of host device, hardware and variations in code can alter run-time on different platforms. The device used is an iMac (27-inch, Mid 2011) model with 2.7 GHz Intel Core i5 processor and AMD Radeon HD 6770M 512 MB graphics. The codes used are the ones mentioned in this paper.

11. Conclusion/Future Work

Based of the work done in this paper, the shift to a quantum-resistant cryptographic era is looking promising. There's definitely room for continuation: a better understanding of the mysterious inner workings of CRYSTALS-KYBER, further analysis for implementation vulnerabilities, enhancing security of key storage, testing for larger data sizes, code optimisations... to name a few. The main purpose of this paper was to document a current implementation to provide future developers with a starting point for making the transition to post-quantum security for their web applications.

References

[1] National Institute of Standards and Technology (2016) *Report on Post-Quantum Cryptography* Available at: https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf

[2] National Institute of Standards and Technology (2020) *Post-Quantum Cryptography - Project Overview* Available at: https://csrc.nist.gov/projects/post-quantum-cryptography

[3] National Institute of Standards and Technology (2020) Status Report on the Second Round of the NIST *Post-Quantum Cryptography Standardization Process* Available at: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf

[4] Proton Team (2018) *What is end-to-end encryption and how does it work?* Available at: https://protonmail.com/blog/what-is-end-to-end-encryption/

[5] Miranda Russell (2019) *End-to-End Encryption for the Payment Industry* Available at: https://www.helcim.com/article/what-is-end-to-end-encryption/

[6] Meredith Galante (2019) *What Is End-to-End Encryption and Why You Really Need It* Available at: https://squareup.com/us/en/townsquare/end-to-end-encryption

[7] Steve Elefant (2011) *Secure online payment system requires end-to-end encryption* Available at: https://searchsecurity.techtarget.com/magazineContent/Secure-online-payment-system-requires-end-to-end-encryption

[8] insightBox (2020) *insightBox Repository* Available at: https://github.com/iommu/insightBox

[9] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, Martin Strand (2016) *A Guide to Fully Homomorphic Encryption* Available at: https://eprint.iacr.org/2015/1192.pdf

[10] Ken Beer, Ryan Holland (2014) *Encrypting Data at Rest* Available at: https://d1.awsstatic.com/whitepapers/AWS_Securing_Data_at_Rest_with_Encryption.pdf

[11] John Aaron Gregg (2003) *On Factoring Integers and Evaluating Discrete Logarithms* Available at: https://wstein.org/projects/john_gregg_thesis.pdf

[12] Qiskit (2020) *Quantum computing in a nutshell* Available at: https://qiskit.org/documentation/qc_intro.html

[13] Peter W. Shor (1995) *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer* Available at: https://arxiv.org/pdf/quant-ph/9508027.pdf

[14] A. Menezes, P. van Oorschot and S. Vanstone (1996) *Handbook of Applied Cryptography, RSA public-key encryption* p. 285 Available at: http://cacr.uwaterloo.ca/hac/about/chap8.pdf

[15] National Institute of Standards and Technology (2020) *Recommendation for Key Management: Part 1 - General* p. 54 Available at: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf

[16] A. Menezes, P. van Oorschot and S. Vanstone (1996) *Handbook of Applied Cryptography, Security of RSA* p. 287 Available at: http://cacr.uwaterloo.ca/hac/about/chap8.pdf

[17] Wikipedia (2021) *Shor's Algorithm, Procedure* Available at: https://en.wikipedia.org/wiki/Shor%27s_algorithm#Procedure

[18] Abraham Asfaw (2020) *9. Shor's Algorithm I: Understanding Quantum Fourier Transform, Quantum Phase Estimation - Part 3* Available at: https://www.youtube.com/watch?v=5kcoaanYyZw

[19] Daniel J. Bernstein (2009) *Introduction to post-quantum cryptography* Available at: http://www.pqcrypto.org/www.springer.com/cda/content/document/cda_downloaddocument/9783540887010-c1.pdf

[20] Daniele Micciancio, Oded Regev (2008) *Lattice-based Cryptography* Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.4862&rep=rep1&type=pdf

[21] Vinod Vaikuntanathan (2015) *The Mathematics of Lattices I* Available at: https://www.youtube.com/watch?v=LlPXfy6bKIY

[22] Oded Regev (2012) *Winter School on Cryptography: Introduction to Lattices - Oded Regev* Available at: https://www.youtube.com/watch?v=4ulHOV8iLls

[23] Miklos Ajtai (1996) *Generating Hard Instances of Lattice Problems* Available at: https://eccc.weizmann.ac.il//eccc-reports/1996/TR96-007/index.html

[24] Henry Corrigan-Gibbs, Sam Kim, David J. Wu (2018) *Lecture 9: Lattice Cryptography and the SIS Problem* Available at: https://crypto.stanford.edu/cs355/18sp/lec9.pdf

[25] Duane Q. Nykamp (2021) *Multiplying Matrices and Vectors* Available at: https://mathinsight.org/matrix_vector_multiplication

[26] Eric W. Weisstein (2021) *Gaussian Elimination* From MathWorld--A Wolfram Web Resource. Available at: https://mathworld.wolfram.com/GaussianElimination.html

[27] Grasiele C. Jorge, Antonio Campello, Sueli I. R. Costa (2013) *Q-ary Lattices in the L_p norm and a Generalization of the Lee Metric* Available at: http://www.ii.uib.no/~matthew/Talks/Tuesday/Afternoon/JorgeCamppeloCosta.pdf

[28] Chris Peikert (2020) *The Learning With Errors Problem and Cryptographic Applications* Available at: https://www.youtube.com/watch?v=K_fNK04yG4o

[29] Bill Buchanan (2018) *Public Key Encryption using Learning With Errors (LWE)* Available at: https://www.youtube.com/watch?v=MBdKvBA5vrw

[30] Adeline Langlois, Damien Stehle (2012) *Worst-Case to Average-Case Reductions for Module Lattices* Available at: https://eprint.iacr.org/2012/090.pdf

[31] Ryan O'Donnell (2020) *Learning With Errors (LWE) and Public Key Encryption || @ CMU || Lecture 25d of CS Theory Toolkit* Available at: https://www.youtube.com/watch?v=QcVns57MTxg&t=999s

[32] Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehle (2020) *CRYSTALS-KYBER Algorithm Specifications And Supporting Documentation (version 3.0)* Available at:

https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf

[33] X. Bonnetain, M. Naya-Plasencia, A. Schrottenloher (2019) *Quantum Security Analysis of AES* Available at: https://eprint.iacr.org/2019/272.pdf

[34] William J Buchanan (2021) *SHAKE Asecuritysite* Available at: https://asecuritysite.com/encryption/shake

[35] Daan Sprenkels (2020) *The Kyber/Dilithium NTT* Available at: https://dsprenkels.com/ntt.html

[36] Eiichiro Fujisaki, Tatsuaki Okamoto (1999) *Secure Integration of Asymmetric and Symmetric Encryption Schemes* Available at: https://link.springer.com/content/pdf/10.1007/3-540-48405-1_34.pdf

[37] W3Schools (2021) *HTML Web Storage API* Available at: https://www.w3schools.com/html/html5_webstorage.asp

[38] Nadim Kobessi (2020) *Kyber-K2SO (v2)* Available at: https://github.com/symbolicsoft/kyber-k2so

[39] Nadim Kobessi, Anton Tutoveanu (2021) *Kyber-K2SO (v3)* Available at: https://github.com/antontutoveanu/kyber-k2so

[40] Go (2021) *Package aes* Available at: https://golang.org/pkg/crypto/aes/

[41] insightBox (2020) *encryption_hooks.go* Available at: https://github.com/iommu/insightBox/blob/master/server/graph/model/encryption_hooks.go

[42] altexsoft (2020) *What is Front-End Development: Key Technologies and Concepts* Available at: https://www.altexsoft.com/blog/front-end-development-technologies-concepts/

[43] W3Schools (2021) *What is npm?* Available at: https://www.w3schools.com/whatis/whatis_npm.asp

[44] Ricmoo (2018) *AES-JS* Available at: https://www.npmjs.com/package/aes-js

[45] Neeh (2018) *pkcs7-padding* Available at: https://www.npmjs.com/package/pkcs7-padding

[46] Rzcoder (2020) *node-rsa* Available at: https://www.npmjs.com/package/node-rsa

[47] Go (2020) *Package rsa* Available at: https://golang.org/pkg/crypto/rsa/