National College of Ireland

Quantum encryption to tackle brute force attacks from quantum encryption

MSc Research Project

Master of Science in Cyber Security

Sudhanshu

Student ID: 22219471

School of Computing

National College of Ireland

Supervisor: Jawad Salahuddin

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Sudhanshu		
Student ID:	22219471		
Programme:	Master of Science in Cyber Security	**Year:**	2023-24
Module:	MSc Research Practicum		
Supervisor:	Jawad Salahuddin		
Submission Due Date:	12th August 2024 14:00		
Project Title:	Quantum encryption to tackle brute force attacks from quantum encryption		
Word Count:	5186		
Page Count	17		

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Sudhanshu

Date: 12th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□

Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Quantum encryption to tackle brute force attacks for IoT devices

Sudhanshu

22219471

Abstract

Iot and small devices are increasing rapidly. More than 16 billion were online before start of 2024. As quantum computing technology advances, the need for robust, quantum-resistant encryption mechanisms becomes increasingly urgent. This thesis explores the potential of quantum encryption, specifically leveraging Quantum Key Distribution (QKD) and post-quantum cryptographic algorithms, to counteract brute force attacks. By integrating quantum encryption techniques with classical encryption methods, this research aims to develop a hybrid cryptographic framework that enhances security against both classical and quantum adversaries. The study includes a detailed analysis of the vulnerabilities of existing systems, the implementation of quantum safe encryption protocols, and a performance evaluation of these protocols in real world scenarios. The results demonstrate that quantum encryption not only provides a robust defence against brute force attacks but also establishes a secure foundation for futur proof cryptographic systems in the era of quantum computing.

1 Introduction

The Internet of Things (IoT) refers to a network of physical devices, vehicles, appliances, and other physical objects that are embedded with sensors, software, and network connectivity, allowing them to collect and share data (IBM, 2024). Now on they make everything from Operational technology to smart devices and homes. The potential applications of IoT are vast and varied, and its impact is already being felt across a wide range of industries, including manufacturing, transportation, healthcare, and agriculture (IBM, 2024). This makes it a prime target to cyber attacks.

However, the proliferation of IoT devices has also introduced new security vulnerabilities, particularly those susceptible to brute force attacks. Now days all the devices connected to the internet use public key cryptography (SSL/TLS) for communication. All the encryption we have are just made to make the guessing of password really hard instead of making it impossible using hard math problems.

In 1982, Richard Feynman introduced the concept of quantum computing, leveraging the principles of quantum mechanics (Toxvaerd, 2024). They are able to perform multiple tasks at once and are extraordinary at multitasking complex problems. This leads to a problem where traditional computer were able to break and encryption in about a trillion

years, now quantum computer using shor's algorithm can break them in mere hours rendering all previous encryption obsolete. This has led to research in a new branch of cryptography, Quantum cryptography.

This these aims to investigate the following research questions:

What are the limitations of traditional encryption methods in protecting IoT devices from brute force attacks?

How can quantum mechanics be applied to develop encryption algorithms that are resistant to brute force attacks?

What are the potential challenges and limitations of implementing quantum encryption for IoT devices?
What are the future prospects and implications of quantum encryption for IoT security?

By addressing these questions, this thesis will try to contribute to the ongoing research and development in quantum encryption of low end devices. We will divide the public key cryptography in 2 parts, Key exchange and encryption. We will try to make a better and robust encryption against quantum encryption using quantum key exchange and traditional encryption.

1.2 Importance

In (Hegde, Jamuar and Kulkarni, 2024) they have shown how easily RSA can be broken using quantum computer. Similarly Shor's algorithm and Grover's algorithm are able to break any encryption or hashing that we have right now with brute force. This year NIST has finalised to release the quantum encryption they see fit for public use (NIST, 2024). NIST has also said to make quantum encryption to be in use by 2026, Since that is when it's expected for Qunatum computers to break RSA-2048 (Azarderakhsh, n.d.). Making sure that no other Encryption remain strong for long.

As cyber threats continue to evolve, it is essential to explore new security technologies. Quantum encryption represents a promising avenue for addressing future challenges and ensuring the long term security of IoT devices. IoT devices often handle sensitive personal and business data. The potential for data breaches and unauthorised access poses significant risks. Quantum encryption can provide a robust defence against these threats.

Few examples where IoTs works are:

1. Nuclear reactors (International Atomic Energy Agency, 2021), show how Industrial Internet of things is important. How they are used to gather information that without there help will be very dangerous.

2. Predictive maintenance: IoT sensors can monitor equipment health, predicting failures and preventing downtime.

3. Remote patient monitoring: Wearable IoT devices such as heart rate monitors and glucose sensors allow healthcare providers to monitor patients' health remotely, improving patient outcomes and reducing the need for hospital visits.

4. Smart cities: Optimise traffic flow, manage parking, and improving the urban infrastructure. IoT devices like smart cameras and connected streetlights enhance public safety by enabling real-time monitoring and faster response times to incidents.

We can see how breaking the encryption in any of these cases can lead to a catastrophic result. Now there are more than 20 billion IoT devices online. If 20 billion devices on the internet are compromised the result would be devastating. Once compromised they can be used for anything.

2 Related Work

2.1 Theoretical Foundations

Bennett, C. H., & Brassard, G. (1984) (Bennett and Brassard, 2014). Quantum cryptography: Public key distribution and coin tossing. In Proceedings of the International Conference on Computers, Systems, & Signal Processing (pp. 175 179). IEEE. This seminal paper introduced the concept of quantum key distribution (QKD), a fundamental building block for quantum encryption.

Ekert, A. K. (1991) (Ekert, 1991). Quantum cryptography based on Bell's theorem. Physical Review Letters, 67(6), 661 663. Ekert proposed a different QKD protocol using entangled pairs of photons, providing alternative approaches to quantum encryption.

(Scarani et al., 2009) This comprehensive review provides a detailed overview of the theoretical foundations of quantum cryptography, including security proofs and potential vulnerabilities.

2.2 Experimental Demonstrations and Proof of Concept:

(Gisin et al., 2002) This review paper summarises the experimental progress made in quantum cryptography, including QKD systems and field tests.

(Lo, Ma and Chen, 2005) Decoy state QKD techniques have been developed to enhance the security and practical implementation of quantum encryption.

(Liu et al., 2023) Twin field QKD protocols offer improved performance and security compared to traditional QKD schemes.

2.3 Quantum Encryption for IoT Applications:

(Shafique et al., 2020)
This survey paper discusses the challenges and opportunities of applying quantum encryption to IoT devices, highlighting the potential benefits and limitations.

(Long, 2017) Quantum secure direct communication (QSDC) offers a different approach to quantum encryption, enabling direct transmission of secret messages without the need for a shared key.

3 Research Methodology

To be able to get appropriate result a lot of research and Theoretical framework needed to be developed. Especially in Encryption, current research done in it and also how quantum encryption works and what is the current landscape of the research. Keeping this in mind I have divided the Research in following sections:

3.1 Theoretical Framework Development

3.1.1: Encryption

TLS that we have uses both Public and Private key encryption. First Public key cryptography is used during the initial handshake to open up a secure channel (Transport Layer Security, n.d.). The client send a premaster secret using this method. So there are 2 encryptions here at work. First one to share the key for the second encryption and then the second one used to do rest of the communication.

TLS uses Elliptic Curve Diffie Hellman (ECDH) 256 bits (Kak, 2019) for the public key encryption and uses AES for the Symmetric encryption.

Symmetric Encryption Key Size in bits	*RSA and Diffie-Hellman "Key" size in bits*	*ECC "Key" Size in bits*
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Table 1:Current best estimates of the key sizes needed to achieve equivalent level of security with three different methods.

ECDH has a really strong encryption even in low key sizes. Making it ideal for low end devices. The computational overhead increases as O(N3) (Kak, 2019), where N is the length of the key. it takes far less computational overhead to use ECC on account of the fact that you can get away with much shorter keys. Table 1 compares the best current estimates of

the key sizes for three different approaches to encryption for comparable levels of security against brute force attacks.

Secondly we have AES which we will see in section 3.1.3

3.1.2: Kyber 512

Most widely use Key exchange algorithms today are based on hard mathematical problems. Which means that they are hard to break rather then impossible. these problems can be efficiently solved by a quantum computer. This is where Key encapsulation Mechanism (KEM) comes into place.

Two people could agree on a secret value if one of them could send the secret in an encrypted form to the other one, such that only the other one could decrypt and use it. This is what a KEM makes possible, through a collection of three algorithms (The Cloudflare Blog, 2022):

- A key generation algorithm, Generate, which generates a public key and a private key (a keypair)(The Cloudflare Blog, 2022).

- An encapsulation algorithm, Encapsulate, which takes as input a public key, and outputs a shared secret value and an “encapsulation” (a ciphertext) of this secret value (The Cloudflare Blog, 2022).

- A decapsulation algorithm, Decapsulate, which takes as input the encapsulation and the private key, and outputs the shared secret value (The Cloudflare Blog, 2022).

KEM is similar to Public Key Encryption since this also uses combination of publ;ic and private keys. The difference is the mathematical problem they both seek to solve. In a KEM, one uses the public key to create an “encapsulation” — giving a randomly chosen shared key and one decrypts this “encapsulation” with the private key.
KEM are more efficient than PKE and aree also post quantum encryption. They use lattices to create a mesh of points which needs to be exact to decrypt the data. Even if one of the lattice is wrong guessing the correct one will never work. Another reason to use lattice mathematics is because of the Learning with errors problem. The security of Kyber, like other lattice based cryptographic schemes, relies on the inherent difficulty of solving LWE problems.

The Learning With Errors problem is a problem in lattice based cryptography that involves solving a system of linear equations that have been perturbed by some small random errors. The difficulty lies in recovering the secret vector from the given matrix A and the perturbed vector b. The introduction of the error vector e makes the problem hard to solve because it "masks" the exact linear relationship. The LWE problem is considered hard both for classical and quantum computers, making it a strong foundation for cryptographic schemes.

Kyber is designed to be efficient in both software and hardware implementations. It requires fewer computational resources then some other post quantum schemes, which makes it suitable for a wide range of applications, including those with resource constraints like IoT devices. The security of Kyber against quantum attacks comes from the hardness of the

Module LWE problem. Quantum computers, which threaten traditional cryptographic schemes by efficiently solving problems like integer factorization and Elleptic curves, are not expected to solve LWE or Module LWE efficiently. This makes Kyber a strong candidate for post quantum cryptography.

We will be using Kyber 512 instead of ECDH in our algorithm to do key exchange.

3.1.3: AES

In TLS, AES is used to encrypt and decrypt the actual data being transmitted. The session keys generated during the handshake are used to encrypt and decrypt data using AES. AES is a symmetric key encryption algorithm, meaning only 1 key is used for both encryption and decryption. This key must be kept secret between the communicating parties.

AES operates on fixed size blocks of data (128, 192, or 256 bits), we will be using 256 bits.

Padding: Since AES requires fixed size blocks, if the plaintext does not align with the block size, padding is added to the last block. This padding must be removed after decryption.

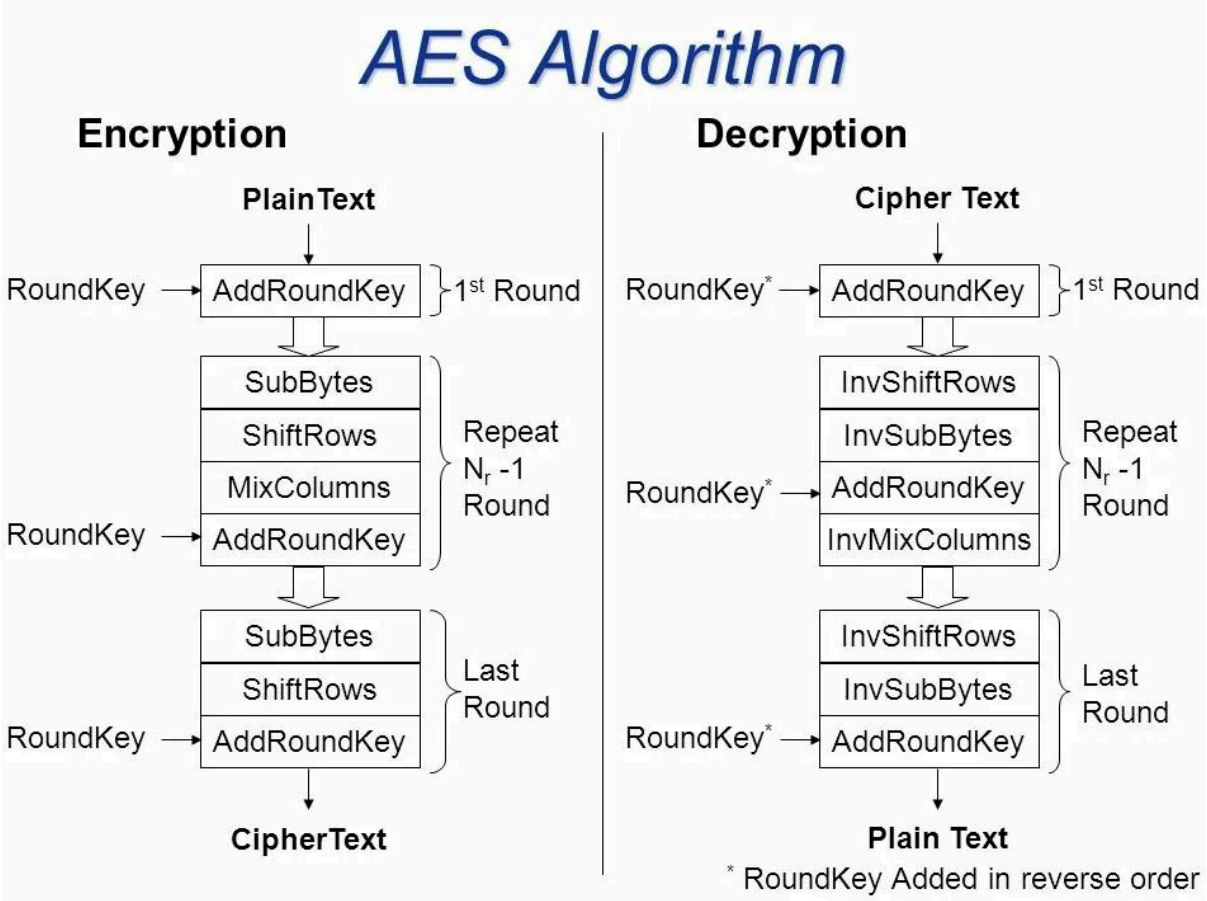

Fig 3.1.3.1: AES Encryption(Komarovskiy, 2023)

Each round in AES have 4 main operations:

Byte Substitution: Each byte in the state matrix is replaced by its corresponding value in a substitution box (S box), which is a non linear transformation.

Row Shifting: The rows of the state matrix are cyclically shifted to the left. The first row remains unchanged, the second row is shifted by one byte, the third row by two bytes, and the fourth row by three bytes.

Column Mixing: Each column of the state matrix is transformed by multiplying it by a fixed 4x4 matrix over GF(2^8). This operation mixes the bytes within each column.

Key Addition: The state matrix is XORed with a round key derived from the AES key schedule

When AES is used in conjunction with key exchange methods that provide forward secrecy (like Ephemeral Diffie Hellman as in TLS), it ensures that even if the long term private keys of the server are compromised, past communications remain secure because the session keys are not derivable from the long term keys. This is the reason AES can be used with kyber 512

3.1.4: IoT

Instead of an IoT we will evaluate the result of the algorithm on an low end device.

Device Configuration:

Intel Celeron
4gb ram
128 gb hard disk SSD
OS: Manjaro with Linux 6.16
Python: 3.12
Running on jupyter lab: 4.2.4

3.2 Qualitative Analysis

Qualitative analysis of an algorithm involves evaluating aspects of the algorithm that may not be easily quantifiable but are crucial for understanding its overall utility, efficiency, and applicability in various contexts.

For analysis of algorithm we will look at the Time and space complexity of the algorithm. Then we will run it and see if there are any edge cases oir any high processing requirement. Once we have this we will take a look at other aspects.

Once done we will look at the Readability of the algorithm and how easy it is to understand and debug. We will look how scaleable the algorithm is and if it can be easily adapted. Then we will look at ease of implementation and how correct it is. The usage is usually quantified in time and space complexity but is it acceptable that the algorithm uses more memory if it significantly reduces computation time? This trade off might be analysed based on the specific context in which the algorithm is used.

4 Design Specification

Since we have decided to use KEM instead of PKE and Kyber 512 instead of ECDH algorithm we will be looking at the algorithms for Kyber 512. We created the Kyber algorithm using 4 steps:

Key Generation: first we generate a public key and a private key

Encapsulation: then we use encapsulation on the public to receive a cipher text and a shared_secret. This is the cipher that will be shared with the other party to decapsulatee.

Decapsulation: The other party decapsulate the cipher. Decapsulating the cipher gives the shared_secret that was generated during the encapsulation phase. Now both ther parties have the shared secret

Verification: a function to verify if both the shared seecret are same or not.

Now wee will look at the changes we made to the AES encryption to accommodate the Kyber 512. it has 2 main steps:

Encryption:

1. Encapsulate to generate the cipher and the shared_secret. We will use the first 32 bytes of sheared secret as the AES encryption key.

2. Prepare the plaintext message for encryption by applying PKCS7 padding to make its length a multiple of the AES block size.

3. Set the Initialization Vector (IV), here for practicee we have used a static value but in practice it needs to be random.

4. Encrypt the padded message

Decryption:

1. We will first decapsulate the shared_secret using the cipher. Now we will use the first 32 bytes of the shared_secret at the AES key. SO we have successfully exchanged the keys with no problem.

2. Set the IV value and decrypt the message.

3. Attempt to remove the PKCS7 padding from the decrypted plaintext. If successful, convert the unpadded plaintext from bytes to a string.

This way we have incorporated the Kyber 512 in AES 256.

5 Implementation

5.1 Tools

The focus was on optimising memry usage, tracking performance, and ensuring secure encryption and key exchange processes. Below is a detailed explanation of the tools used:

Python is used as a programming language since it already has many of the libraries built of kyber and AES. It also made thee code a lot easier to understand and modular.

Jupyter Notebook: An interactive development environment used for coding, testing, and documenting the research in a literate programming format. It was particularly useful for integrating code execution with descriptive text and visualisations.

5.1.1 Libraries:

1. memory_profiler: The @profile decorator from memory_profiler was applied to key functions to monitor their memory consumption during execution. It gives a detailed analysis of the algorithm during the execution in a form of a table. By profiling memory, the research aimed to identify and optimise parts of the code that were using excessive memory, which is crucial in cryptographic implementations, particularly when working with resource constrained environments like IoT devices.

2. time: he time module was utilised to measure the execution time of various operations within the cryptographic algorithms. This was important for understanding the performance characteristics of the implemented encryption and key exchange processes. It gave thee entire execution time including the decryption of the code.

3. tracemalloc: was used to trace memory allocation and identify memory leaks. This tool was crucial in ensuring that the cryptographic implementations were not only efficient in terms of speed but also optimised for memory usage since the memory is a huge issue when it comes to IoT devices.

4. Kyber512: Kyber512 is part of the Kyber family of cryptographic algorithms that are based on lattice based cryptography, making them resistant to quantum attacks. Kyber512 was employed in the thesis to perform key generation, encapsulation, and decapsulation operations.

5. cryptography: This module from the Cryptography library was used to implement AES 256 encryption and decryption. he Cipher class, along with algorithms (AES) and modes (CBC), was utilised to seecure data during the encryption process. The default backend provided the underlying cryptographic operations that enabled the secure implementation of the AES algorithm within the research.

5.2 Kyber 512

We create a KEM_Kyper class that will contain all the function for key generation, encapsulate and decapsulate.

```
class KEM_Kyber:
    def __init__(self):
        self.public_key, self.secret_key = kyber.keygen()
```

Fig 5.2.1: Class initialisation of KEM_Kyper

In class initialisation we create the public and the private key using keygen(). Later on we use these keys for cipher and secret generation.

```
    def encapsulate(self):
        return kyber.enc(self.public_key)
```

Fig 5.2.2: Encapsulation

This function uses the public key to encrypt or encapsulate a randomly generated shared secret. Encapsulation is used to generated the cipher and shared_secret. Cipher when decapuslated will generate the same shared_secret. This function returns a tuple containing these 2 values.

```
    def decapsulate(self, cipher):
        return kyber.dec(cipher, self.secret_key)
```

Fig 5.2.3: Decapsulation

This function uses the private key to decrypt the ciphertext and retrieve the original shared secret.

```
    def check_shared_secret(self, cipher, shared_secret_enc):
        if shared_secret_enc == self.decapsulate(cipher):
            print("Decryption successful! Shared secrets match.")
        else:
            print("Decryption failed! Shared secrets do not match.")
```

Fig 5.2.4: Check

The method takes two inputs: cipher (the ciphertext) and shared_secret_enc (the original encapsulated shared secret). It then compares the decapsulated shared secret with the original shared secret. If they match it prints "Decryption successful! Shared secrets match."

This process allows the secure establishment of a shared secret between two parties which can then be used for symmetric encryption of further communications. The Kyber algorithm ensures tha this process is secure even against attackers with quantum computing capabilities.

5.3 AES

We create a AES class to implement AES 256 and a post quantum key encapsulation mechanism (KEM) based on the Kyber algorithm. This class is designed to handle the encryption and decryption of messages using a shared secret generated by the Kyber KEM.

```
[51]: class AES256:
          def __init__(self):
              self.kem = KEM_Kyber()
              self.kem.encapsulate()
```

Fig: 5.3.1: AES class constructor

The constructor method initialises an instance of the AES256 class by setting up a Kybe 512 for Key Encapsulation Mechanism (KEM). The 'self.kem.encapsulate()' method is called to initiate the key encapsulation process, generating an initial shared secret and ciphertext. However, this initial encapsulation does not store or use the results immediately—it prepares the object for further operations.

```
    @profile
    def encrypt(self, message):
            # Use the shared secret as the AES key (first 32 bytes for AES-256)
        ciphertext, shared_secret_enc = self.kem.encapsulate()
        aes_key = shared_secret_enc[:32]
            # Prepare the message for encryption by padding it
        padder = padding.PKCS7(algorithms.AES.block_size).padder()
        padded_message = padder.update(message.encode()) + padder.finalize()

            # Encrypt the padded message using AES in CBC mode with a zero IV
        iv = b'\x00' * 16  # Static IV for simplicity; in practice, this should be random

        cipher = Cipher(algorithms.AES(aes_key), modes.CBC(iv), backend=default_backend())
        encryptor = cipher.encryptor()
        encrypted_message = encryptor.update(padded_message) + encryptor.finalize()
        return ciphertext, encrypted_message
```

Fig 5.3.2: Encryption

this method encrypts a plaintext message using AES 256 in Cipher Block Chaining (CBC) mode. The AES key is derived from the shared secret generated by the Kyber KEM.

The method first calls 'self.kem.encapsulate()' to generate a new shared secret and its corresponding ciphertext. The first 32 bytes of the shared secret are extracted to be used as the AES 256 encryption key (aes_key).

The padding process involves adding extra bytes to the plaintext so that its length matches the required block size. The padded message is encrypted using AES 256 in CBC mode. ‘cryptography.hazmat.primitives.ciphers’ encrypts the message. The method returns the Kyber generated ciphertext (used for the shared secret) and the AES encrypted message.

```
def decrypt(self, cipher, encrypteed_msg):
    # Decapsulate the shared secret using the secret key
    shared_secret_dec = self.kem.decapsulate(cipher)
    aes_key = shared_secret_dec[:32]
        # Decrypt the message using AES in CBC mode with a zero IV
    iv = b'\x00' * 16
    cipher = Cipher(algorithms.AES(aes_key), modes.CBC(iv), backend=default_backend())
    decryptor = cipher.decryptor()
    padded_message = decryptor.update(encrypteed_msg) + decryptor.finalize()

        # Unpad the decrypted message
    try:
        unpadder = padding.PKCS7(algorithms.AES.block_size).unpadder()
        message = unpadder.update(padded_message) + unpadder.finalize()
        return message.decode()
    except ValueError as e:
        print(f"Decryption failed: {e}")
        return None
```

Fig 5.3.3.Decryption

This method decrypts a ciphertext encrypted using the encrypt method, restoring the original plaintext message.

The method calls self.kem.decapsulate(cipher) to retrieve the shared secret that was originally used to encrypt the message.

The first 32 bytes of the shared secret are extracted to be used as the AES 256 decryption key (aes_key).

The decryption process involves reversing the encryption steps to retrieve the padded plaintext.

The PKCS7 padding is removed from the decrypted plaintext to restore the original message. The method attempts to unpad the message and decode it to a string. If successful, it returns the decoded plaintext message.

The use of Kyber 512 for key encapsulation ensures that the key exchange process is secure even in the face of quantum computing threats providing long term security for the encrypted communications.

6 Evaluation

When executing this in a terminal we get that it took 1.5 seconds of total time in encryption and decryption from KEM key generation to AES Decryption.

The total memory usage increases from 57.1 MB at the start to 59.3 MB by the end of the encrypt method. This represents an overall increase of 2.2 MB during the method's execution. The encryption process itself also requires a notable amount of memory (0.8 MB). The method appears to be relatively efficient in its use of memory, with most operations not causing significant spikes in memory usage. However, the creation of cryptographic objects (such as the AES cipher) understandably increases memory consumption but it still is under control.

In the memory it shows that the highest amount of memory was used by the libraries that were iomported. The binding.py file from the cryptography library is a significant consumer of memory, which is expected as cryptographic operations, especially those that involve binding to lower level libraries like OpenSSL, are resource intensive.

The inspect and enum modules also consume a fair amount of memory, indicating that the script may involve introspection and the use of enumerated types, both of which can be memory intensive.

6.1 Recommendation

If the memory usage by the cryptographic bindings is a concern, consider exploring optimisations in how these operations are performed. Performance can also be significantly improved it we use a low level language like C++. This will however increase the complexity and readability of the code.

Since we are working with IoT devices and this being a first vertison there can be better optimisation techniques that can be applied here. OPENSSL can be better utilised here instead of having to use a wrapper around it and can be configured to be better performing for this use case.

There is a need to perform a comparison between this approach dn other methods that are being researched on to make sure this implementation is is competitive in terms of efficiency.

7 Conclusion and Future Work

The performance and memory profiling of the implemented cryptographic operations, specifically within the AES256 class and its integration with the Kyber key encapsulation mechanism, reveals critical insights into the efficiency and security of the system. This also shows that to have a quantum secure encryption we do not have to update the infrastructure. IoT devices even though less in memory and processing have enough capacity to run these algorithms.

The current implementation provides a solid foundation for secure post quantum cryptography, but targeted improvements can further enhance its suitability for a wide range of applications, including those in memory constrained environments like IoTs.

In Future work algorithms like NTRU can be tested since it has a low memory profile and has shown to be highly efficient. The limitation we faced in this study was understanding as to how these algorithms work. NTRU seems promising to provide a competitive result.

Research hybrid cryptographic schemes that combine classical and post quantum algorithms to offer a smooth transition to post quantum security while maintaining compatibility with existing systems. But need for a encryption to replace AES is also on hand. There also needs to be a Investigation in the potential benefits of hardware acceleration for cryptographic operations, such as using GPUs or dedicated cryptographic hardware, to significantly boost performance. even in IoT devices we can embed a chip for better cryptographic performance.

References

7enTropy7 (2020). *GitHub - 7enTropy7/NTRU_cryptography: A Post-Quantum Encryption Algorithm*. [online] GitHub. Available at: https://github.com/7enTropy7/NTRU_cryptography [Accessed 10 Aug. 2024].

Azarderakhsh, R. (n.d.). *Hardware Deployment of Hybrid PQC: SIKE+ECC*. [online] Available at: https://csrc.nist.gov/CSRC/media/Presentations/hardware-deployment-of-hybrid-pqc/images-media/session-3-reza-hardware-deployment.pdf [Accessed 21 Aug. 2024].

Bennett, C.H. and Brassard, G. (2014). Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, [online] 560, pp.7–11. doi:https://doi.org/10.1016/j.tcs.2014.05.025.

Ekert, A. (1991). Quantum Cryptography Based on Bell's Theorem. *Physical review letters*, [online] 67(6), pp.661–663. doi:https://doi.org/10.1103/PhysRevLett.67.661.

Gisin, N., Ribordy, G., Tittel, W. and Zbinden, H. (2002). Quantum cryptography. *Reviews of Modern Physics*, 74(1), pp.145–195. doi:https://doi.org/10.1103/revmodphys.74.145.

Hegde, S.B., Jamuar, A. and Kulkarni, R. (2024). Post Quantum Implications on Private and Public Key Cryptography. *IEEE*. [online] doi:https://doi.org/10.13140/RG.2.2.29101.87521/2.

IBM (2024). *What is the internet of things?* [online] IBM. Available at: https://www.ibm.com/topics/internet-of-things [Accessed 3 Aug. 2024].

Innovate Skills Software Institute (2023). *What Is KEM? Key Encapsulation and Decapsulation. Hybrid Cryptography.* [online] YouTube. Available at: https://www.youtube.com/watch?v=qLnVBycHH1c [Accessed 11 Aug. 2024].

International Atomic Energy Agency (2021). *Maintenance of Nuclear Power Plants*. International Atomic Energy Agency.

Kak, A. (2019). *Lecture 14: Elliptic Curve Cryptography and Digital Rights Management Lecture Notes on 'Computer and Network Security'*. [online] Available at: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture14.pdf.

Komarovskiy, V. (2023). *Enforcing AES 256 Bit Encryption for Domain Authentication*. [online] www.linkedin.com. Available at: https://www.linkedin.com/pulse/enforcing-aes-256-bit-encryption-domain-valentin-komarovskiy-mba.

Liu, Y., Zhang, W.-J., Jiang, C., Chen, J.-P., Ma, D., Zhang, C., Pan, W.-X., Dong, H., Xiong, J.-M., Zhang, C.-J., Li, H., Wang, R.-C., Lu, C.-Y., Wu, J., Chen, T.-Y., You, L., Wang, X.-B., Zhang, Q. and Pan, J.-W. (2023). 1002 km twin-field quantum key distribution with finite-key analysis. *Quantum frontiers*, 2(1). doi:https://doi.org/10.1007/s44214-023-00039-9.

Lo, H.-K., Ma, X. and Chen, K. (2005). Decoy State Quantum Key Distribution. *Physical Review Letters*, 94(23). doi:https://doi.org/10.1103/physrevlett.94.230504.

Long, G.-L. (2017). *Quantum Secure Direct Communication: Principles, Current Status, Perspectives*. [online] ieeexplore.ieee.org. Available at: https://ieeexplore.ieee.org/document/8108697.

Mazex (2023). *kyber-py*. [online] PyPI. Available at: https://pypi.org/project/kyber-py/ [Accessed 11 Aug. 2024].

NIST. (2024). *NIST Releases First 3 Finalized Post-Quantum Encryption Standards | NIST*. [online] Available at: https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards.

Piétri, Y. and Halliwell, J. (2020). *QFFF DISSERTATION Quantum Cryptography*. [online] Available at: https://www.imperial.ac.uk/media/imperial-college/research-centres-and-groups/theoretical-physics/msc/dissertations/2020/Yoann-Pietri-Dissertation.pdf.

ProtDos (2023). *GitHub - ProtDos/pq-ntru: A quantum proof (post-quantum) implementation of the NTRU algorithm*. [online] GitHub. Available at: https://github.com/protdos/pq-ntru [Accessed 10 Aug. 2024].

Scarani, V., Bechmann-Pasquinucci, H., Cerf, N.J., Dušek, M., Lütkenhaus, N. and Peev, M. (2009). The security of practical Quantum Key Distribution. *Reviews of Modern Physics*, [online] 81(3), pp.1301–1350. doi:https://doi.org/10.1103/revmodphys.81.1301.

Schwabe, P. (n.d.). *Kyber*. [online] pq-crystals.org. Available at: https://pq-crystals.org/kyber/ [Accessed 10 Aug. 2024].

Shafique, K., Khawaja, B.A., Sabir, F., Qazi, S. and Mustaqim, M. (2020). Internet of Things (IoT) for Next-Generation Smart Systems: A Review of Current Challenges, Future Trends and Prospects for Emerging 5G-IoT Scenarios. *IEEE Access*, [online] 8(8), pp.23022–23040. doi:https://doi.org/10.1109/ACCESS.2020.2970118.

Sluis, P.A. van der (2020). *The Effect of Sparse Ternary Secrets on the Hardness of Learning with Errors*. [online] Available at: https://studenttheses.uu.nl/bitstream/handle/20.500.12932/38368/Master%20Thesis%20PA%20van%20der%20Sluis.pdf?sequence=1 [Accessed 11 Aug. 2024].

The Cloudflare Blog. (2022). *Deep dive into a post-quantum key encapsulation algorithm*. [online] Available at: https://blog.cloudflare.com/post-quantum-key-encapsulation/.

Toxvaerd, S. (2024). Simulating Physics with Computers. *arXiv (Cornell University)*. doi:https://doi.org/10.48550/arxiv.2405.20780.

Transport Layer Security. (n.d.). Available at: https://www.handsonsecurity.net/files/chapters/edition3/sample-tls.pdf.