National College of Ireland

Configuration Manual

MSc Research Project
Cyber Security

Taraka Raghavendra Sai Panchakarla
Student ID: X22150951

School of Computing
National College of Ireland

Supervisor: Dr Rohit Varma

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Taraka Raghavendra Sai Panchakarla
Student ID:	x22150951
Programme:	MSc in Cyber Security **Year:** 2023-2024
Module:	MSc Research Project
Lecturer:	Dr Rohit Varma
Submission Due Date:	25 April 2024
Project Title:	Secure Communication Using Quantum Key Distribution and Honey Encryption
Word Count:	718 excluding references and appendices **Page Count:** 9

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: Taraka Raghavendra Sai Panchakarla

Date: 25 April 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):	

Configuration Manual

Taraka Raghavendra Sai Panchakarla
Student ID: x22150951

1. Introduction

This handbook includes specifics about the setup and prerequisites for the suggested models, in addition to the required software and libraries. The configuration handbook also contains instructions on how to use the methods required to construct the suggested model.

2. System Configurations

Research Design and Approach

Processor	12th Gen Intel(R) Core (TM) i7-12700H 2.30 GHz
RAM	16.0 GB (15.7 GB usable)
System type	64-bit operating system, x64-based processor

Software and tools

OS	Windows 11 Home Single Language
Python	Python 3.12.1
Code Editor	Jupyter Notebook

3. Installation

Some of the python packages need to be installed like pip install python and pip install jupyter and next install pip jupyter notebook.

Figure 1: Opening Jupyter Notebook

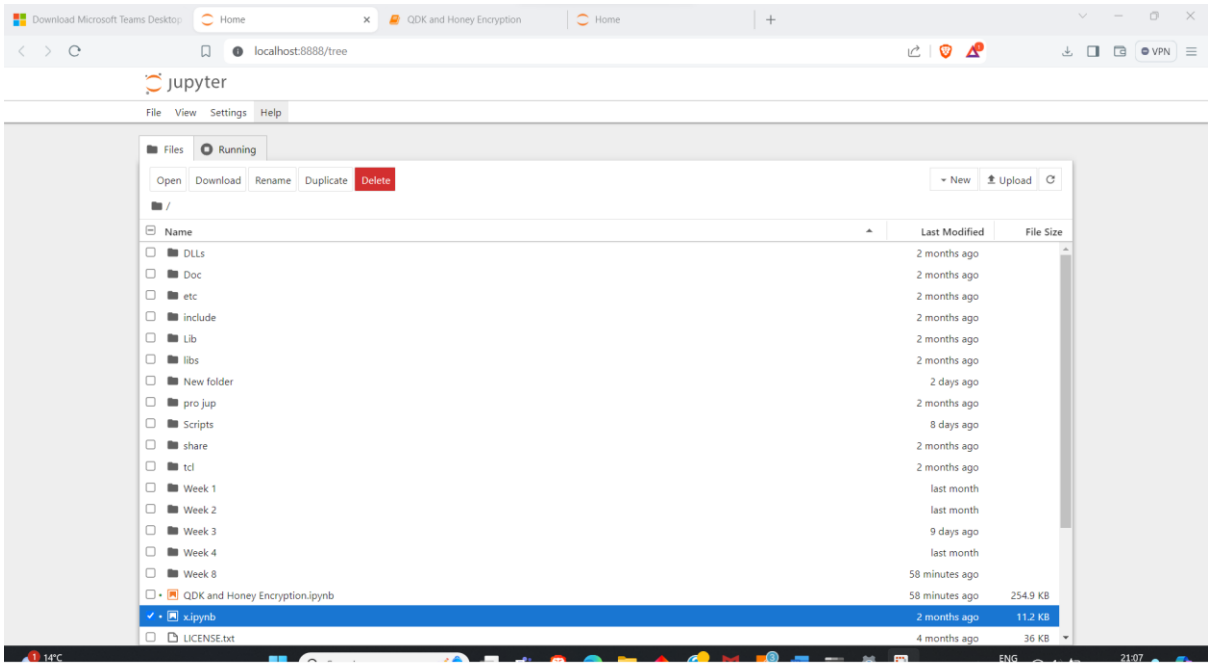

Figure 2: Jupyter Notebook

4. Implementation

Python Code: There are seven major steps in the Python code.

1) The performing of Quantum Key Distribution (QKD). This has three more steps as follows:
   - First step contains code used for generating Qubits.
   - Next step will be consisting of the program for comparing of the qubits generated.
   - The third step will be of generating of key which is used to encryption of the secret key.
2) The second phase will be execution of Honey Encryption (HE) and Decryption. This will be having two steps as follows:
   - Once the secret key is generated that needs to be encrypted and protecting it with a password. This encryption is the fourth step.
   - The next step will be to decrypt the encrypted secret key to the legitimate source user.
3) The sixth step will running the main function program contains the commands and displaying all the results as per the program requirements.
4) The last step will be showing the performance and Evaluation reports such as graphs and other things.

1.
As we mentioned above the first step for generating the qubits is to run the below code shown in Figure 3. As per BB84 protocol Qubits are generated with bits and bases as per Table 1. if the rectilinear basis (y) compared with bit value 0 it is called 0_p. else if rectilinear basis (y) compared with bit value 1 it is called 45_p. else if diagonal basis (x) compared with bit value 0 it is called 90_p. else it is called as 135_p respectively as per the input values (Anusuya Devi V, 2021).

BB84 Protocol Polarization Scheme		
Rectilinear Polarization Basis **(Y)**	0_p	45_p
Diagonal Polarization Basis **(X)**	90_p	135_p
Bit Value	0	1

Table 1: BB84 polarization scheme

```
def generate_qubits(bits, basis):
    qubits = []
    for bit, base in zip(bits, basis):
        if base == 'y':
            if bit == '0':
                qubits.append('0_p')
            else:
                qubits.append('90_p')
        elif base == 'x':
            if bit == '1':
                qubits.append('135_p')
            else:
                qubits.append('45_p')
    return qubits
```

Figure 3 Code for generating Qubits

2.

Once the Qubits of A and B are formed, they much be compared if all qubits of A and B are the same print as open the message. else if no qubits are the same print do the generation of qubits again. else if some quits are same and some or not same then perform an XOR operation between matched bits value and A remaining not matched bits. The code for this will be as shown in Figure 4 (Agarwal, 2023).

```
def compare_qubits(qubits_a, qubits_b):
    if len(qubits_a) != len(qubits_b):
        return "Qubit lists have different lengths."

    matching_count = 0
    secret_key = ''
    for a, b in zip(qubits_a, qubits_b):
        if a == b:
            matching_count += 1
        else:
            secret_key += str(int(a.split('_')[0]) ^ int(b.split('_')[0]))
    if matching_count == len(qubits_a):
        return "Message for B: All qubits matched."
    elif matching_count > 0:
        return "Secret Key: " + secret_key
    else:
        return "Intruder Alert."
```

Figure 4 Code for comparing the Qubits

3.
If some quits are same and some or not same, then perform an XOR operation between matched bits value and A remaining not matched bits. A key need to be used for hashing and password so that for decryption. The code for creating the key is shown in Figure 5.

```
def generate_key(password, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        iterations=100000,
        salt=salt,
        length=32,
        backend=default_backend()
    )
```

Figure 5 Generating of key

4.
Once all the things such as bits, bases, qubits, secret key, key, and password provided by using these the secret key need to be encrypted. That encryption will be done by using code shown below Figure 6 (Nahri Syeda Noorunnisa, 2016).

```
def encrypt_message(message, key):
    cipher = Fernet(key)
    start_time = time.time()
    encrypted_message = cipher.encrypt(message.encode())
    end_time = time.time()
    encryption_time = end_time - start_time
    return encrypted_message, encryption_time
```

Figure 6 Code for encryption of secret key

5.
The encrypted message needs to be decrypted only for the receiver. Do so the receiver need to give the right password kept by the sender to decrypt and know the secret key. This type of code shown in Figure 7 (Campos, 2021).

```
def decrypt_message(encrypted_message, key):
    cipher = Fernet(key)
    start_time = time.time()
    decrypted_message = cipher.decrypt(encrypted_message).decode()
    end_time = time.time()
    decryption_time = end_time - start_time
    return decrypted_message, decryption_time
```

Figure 7 Code for decrypting

6.
The six step we have the main function where it contains the commands and displaying all the results as per the program requirements. The code is shown in Figure 8.

```
def main():
    # Eavesdropping attack detection
    eavesdropping_detected = False
    if input("Is there an eavesdropping attack? (True/False): ").lower() == 'true':
        print("Eavesdropping attack confirmed. Program execution stopped.")
        return

    # Secret key generation
    user_password = input("Enter the password to secure the secret key: ")
    salt = b'\x9e\xe1\xea\xb9\x05~\x8b\x06\x97\x8fI!\x89r\xb9\x12'

    user_length = int(input("Enter the length of the bits: "))
    basis_a = ''.join(random.choice(['x', 'y']) for _ in range(user_length))
    bits_a = ''.join(str(random.randint(0, 1)) for _ in range(user_length))
    basis_b = ''.join(random.choice(['x', 'y']) for _ in range(user_length))
    bits_b = ''.join(str(random.randint(0, 1)) for _ in range(user_length))

    print("User A Basis:", basis_a)
    print("User A Bits:", bits_a)
    print("User B Basis:", basis_b)
    print("User B Bits:", bits_b)

    qubits_a = generate_qubits(bits_a, basis_a)
    qubits_b = generate_qubits(bits_b, basis_b)

    print("User A Qubits:", qubits_a)
    print("User B Qubits:", qubits_b)

    comparison_result = compare_qubits(qubits_a, qubits_b)
    print(comparison_result)

    # Encryption and decryption of secret key
    user_message = input("Enter the message to be encrypted with the secret key: ")
    key, key_generation_time = generate_key(user_password, salt)
    print("Secret Key:", key)
    print("Time taken to generate key:", key_generation_time, "seconds")

    encrypted_message, encryption_time = encrypt_message(user_message, key)
    print("\nEncrypted Message:", encrypted_message)
    print("Time taken for encryption:", encryption_time, "seconds")

    entered_password = input("Enter the password to decrypt the message: ")
    entered_key = generate_key(entered_password, salt)[0]
    if entered_key == key:
        decrypted_message, decryption_time = decrypt_message(encrypted_message, key)
        print("Decrypted Message:", decrypted_message)
        print("Time taken for decryption:", decryption_time, "seconds")
    else:
        print("Intruder alert! Incorrect password.")
        # Provide a random key to A user
        random_key = generate_key(str(random.randint(0, 999999)), salt)[0]
        print("Random Key provided to A user:", random_key)
```

Figure 8 Main functions

7. last step will be showing the performance and Evaluation reports such as graphs

```
# Generate graphical representation of total execution according to bytes vs time
x = []
y = []
start_time = time.time()
total_bytes = 0
for _ in range(1000):
    key, key_generation_time = generate_key('password', salt)
    encrypted_message, encryption_time = encrypt_message('message', key)
    decrypted_message, decryption_time = decrypt_message(encrypted_message, key)
    total_bytes += len(key) + len(encrypted_message) + len(decrypted_message)
    elapsed_time = (time.time() - start_time) * 1000
    x.append(elapsed_time)
    y.append(total_bytes)

plt.plot(x, y)
plt.xlabel('Time (milliseconds)')
plt.ylabel('Total Bytes Exchanged')
plt.title('Total Bytes vs Time')
plt.show()

# Generate graph for encryption time
x_encryption = []
y_encryption = []
for _ in range(1000):
    _, encryption_time = encrypt_message('message', key)
    x_encryption.append(encryption_time * 1000)  # Convert to milliseconds
    y_encryption.append(len(encrypted_message))  # Length of encrypted message

plt.plot(x_encryption, y_encryption)
plt.xlabel('Time (milliseconds)')
plt.ylabel('Encrypted Message in MB')
plt.title('Encryption Time vs Encrypted Message')
plt.show()

# Generate graph for decryption time
x_decryption = []
y_decryption = []
for _ in range(1000):
    _, decryption_time = decrypt_message(encrypted_message, key)
    x_decryption.append(decryption_time * 1000)  # Convert to milliseconds
    y_decryption.append(len(decrypted_message))  # Length of decrypted message

plt.plot(x_decryption, y_decryption)
plt.xlabel('Time (milliseconds)')
plt.ylabel('Decrypted Message in MB')
plt.title('Decryption Time vs Decrypted Message')
plt.show()
```

Figure 9 Performance and Evaluation Graphs

5. References

V, A.D. and V, K. (2021) Enhanced BB84 Quantum Cryptography Protocol for secure communication in Wireless Body Sensor Networks for medical applications, Personal and ubiquitous computing. Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7971400/ (Accessed: 20 April 2024).

Agarwal, Neha & Verma, Vikas. (2023). Comparative Analysis of Quantum Key Distribution Protocols: Security, Efficiency, and Practicality. 10.1007/978-3-031-48774-3_10.

Noorunnisa, N.S. and Siddiqui, R.A. (2016) review on Honey Encryption Technique , International Journal of Science and Research. Available at: https://www.researchgate.net/publication/296806881_Review_on_Honey_Encryption_Technique (Accessed: 20 April 2024).

Campos, P.T. (2021) AES implementation in Python, Medium. Available at: https://medium.com/quick-code/aes-implementation-in-python-a82f582f51c2 (Accessed: 20 April 2024).