Quantum Computing Capstone Projects

Encryption in Python

QXQ 2024 Quantum Computing Program!

Student

Macari Magdy Fayez

April , 2024

Introduction

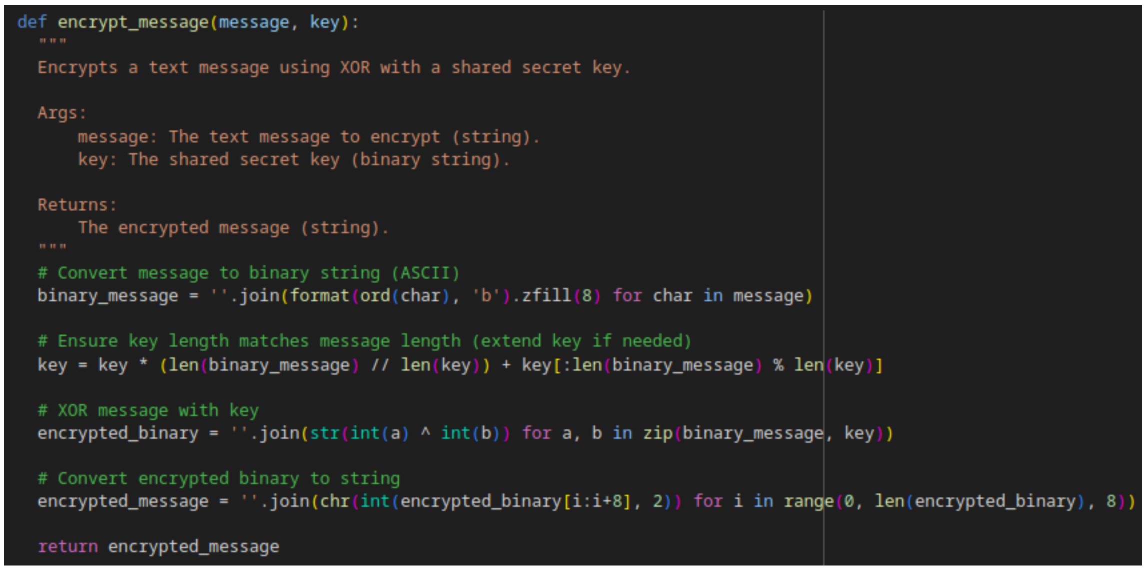
In crafting the encryption and decryption functions, I aimed to create a straightforward yet effective way to secure messages using a shared secret key. The approach I adopted was based on the XOR operation, a fundamental concept in cryptography.

For the `encrypt_message` function, I began by converting the input text message into a binary string representation using ASCII encoding. This ensured that each character in the message could be represented as a sequence of 8 bits. Then, I ensured that the length of the key matched the length of the binary message by repeating or truncating the key as necessary. Next, I performed a bitwise XOR operation between each bit of the binary message and the corresponding bit of the key to generate the encrypted binary message. Finally, I converted the encrypted binary back into a string representation to produce the encrypted message.

Similarly, for the `decrypt_message` function, I followed a similar process but in reverse. I converted the encrypted message back into a binary string, ensuring that the key length matched the message length. Then, I performed the XOR operation between each bit of the encrypted binary and the key to recover the original binary message. Finally, I converted the decrypted binary back into a string representation to obtain the decrypted message.

Throughout the development process, I paid careful attention to ensuring that the functions handled edge cases effectively, such as cases where the key was shorter or longer than the message. I also aimed to keep the implementation as simple and understandable as possible while maintaining functionality. One challenge I encountered was managing the conversion between binary strings and ASCII characters, but this was addressed by breaking down the process into smaller, manageable steps.

Overall, the creation of these functions involved a balance between simplicity, efficiency, and effectiveness in providing basic message encryption and decryption capabilities.



```python
def encrypt_message(message, key):
  """
  Encrypts a text message using XOR with a shared secret key.

  Args:
      message: The text message to encrypt (string).
      key: The shared secret key (binary string).

  Returns:
      The encrypted message (string).
  """
  # Convert message to binary string (ASCII)
  binary_message = ''.join(format(ord(char), 'b').zfill(8) for char in message)

  # Ensure key length matches message length (extend key if needed)
  key = key * (len(binary_message) // len(key)) + key[:len(binary_message) % len(key)]

  # XOR message with key
  encrypted_binary = ''.join(str(int(a) ^ int(b)) for a, b in zip(binary_message, key))

  # Convert encrypted binary to string
  encrypted_message = ''.join(chr(int(encrypted_binary[i:i+8], 2)) for i in range(0, len(encrypted_binary), 8))

  return encrypted_message
```

Encrypt Message Function Breakdown

1. **Convert Message to Binary String**:

   - The function converts the input text message into a binary string representation using ASCII encoding. Each character in the message is converted to its corresponding 8-bit binary representation.

2. **Ensure Key Length Matches Message Length**:

   - Next, the function ensures that the length of the key matches the length of the binary message. If the key is shorter than the message, it is repeated to match the length of the message. If the key is longer than the message, it is truncated to match the length of the message.

3. **XOR Operation**:

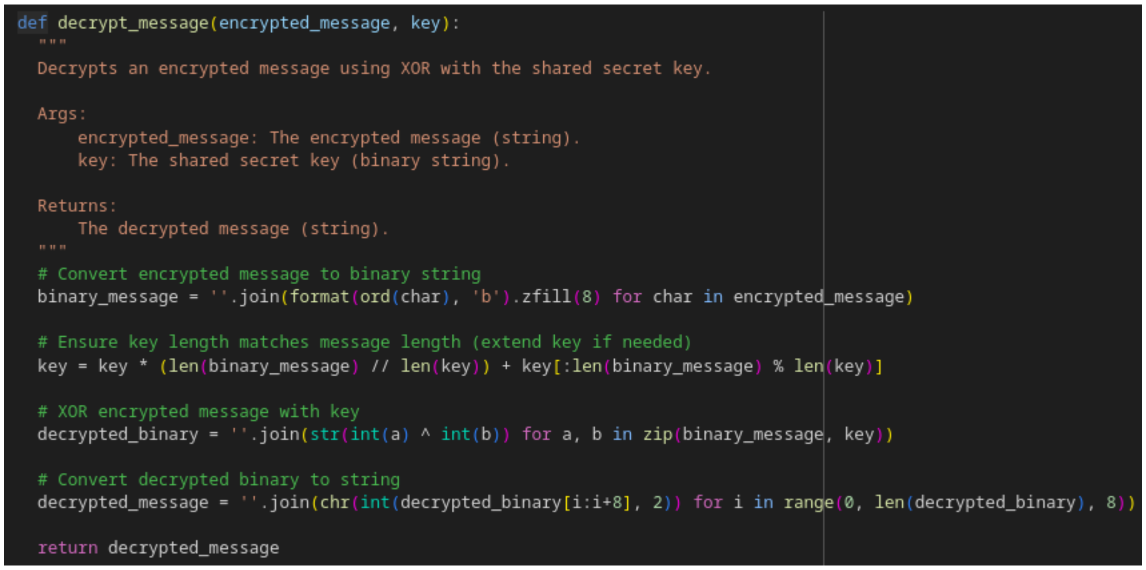
   - The function performs a bitwise XOR (exclusive OR) operation between each bit of the binary message and the corresponding bit of the key. This operation combines the bits of the message and the key to produce the encrypted binary message.

4. **Convert Encrypted Binary to String**:

   - The encrypted binary message is then converted back to a string representation by grouping the binary digits into chunks of 8 bits (1 byte) and converting each chunk into its corresponding ASCII character. This process results in the encrypted message.

5. **Return Encrypted Message**:

    - Finally, the function returns the encrypted message as a string.



```python
def decrypt_message(encrypted_message, key):
    """
    Decrypts an encrypted message using XOR with the shared secret key.

    Args:
        encrypted_message: The encrypted message (string).
        key: The shared secret key (binary string).

    Returns:
        The decrypted message (string).
    """
    # Convert encrypted message to binary string
    binary_message = ''.join(format(ord(char), 'b').zfill(8) for char in encrypted_message)

    # Ensure key length matches message length (extend key if needed)
    key = key * (len(binary_message) // len(key)) + key[:len(binary_message) % len(key)]

    # XOR encrypted message with key
    decrypted_binary = ''.join(str(int(a) ^ int(b)) for a, b in zip(binary_message, key))

    # Convert decrypted binary to string
    decrypted_message = ''.join(chr(int(decrypted_binary[i:i+8], 2)) for i in range(0, len(decrypted_binary), 8))

    return decrypted_message
```

Explanation of the decrypt_message Function

The `decrypt_message` function decrypts an encrypted message using XOR with a shared secret key. Here's a breakdown of how it works:

1. **Convert Encrypted Message to Binary String**:

    - The function converts the input encrypted message into a binary string representation using ASCII encoding. Each character in the encrypted message is converted to its corresponding 8-bit binary representation, padded with leading zeros if necessary, and concatenated to form the binary message.

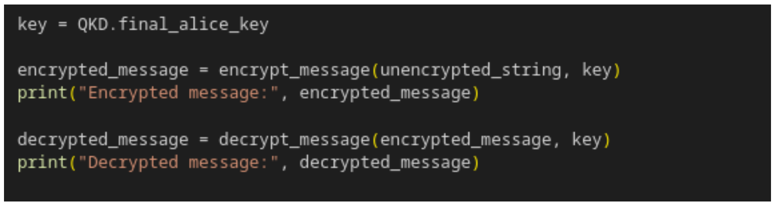
2. **Ensure Key Length Matches Message Length**:

    - Similar to encryption, the function ensures that the length of the key matches the length of the binary message. If the key is shorter than the message, it is repeated to match the length of the message. If the key is longer than the message, it is truncated to match the length of the message.

3. **XOR Operation**:

    - The function performs a bitwise XOR (exclusive OR) operation between each bit of the binary message and the corresponding bit of the key. This operation reverses the encryption process, recovering the original binary message.

4. **Convert Decrypted Binary to String**:
   - The decrypted binary message is converted back to a string representation by grouping the binary digits into chunks of 8 bits (1 byte) and converting each chunk into its corresponding ASCII character. The resulting characters are concatenated to form the decrypted message.
5. **Return Decrypted Message**:
   - The function returns the decrypted message as a string.



```
key = QKD.final_alice_key

encrypted_message = encrypt_message(unencrypted_string, key)
print("Encrypted message:", encrypted_message)

decrypted_message = decrypt_message(encrypted_message, key)
print("Decrypted message:", decrypted_message)
```

Explanation of the Encryption and Decryption Process

The code snippet performs the encryption and decryption process using the `encrypt_message` and `decrypt_message` functions, respectively. Here's a breakdown of the steps involved:

1. **Generate Key using QKD**:
   - The `QKD` function is used to generate a shared secret key. This key is obtained from the `final_alice_key` attribute of the `QKD` object.
2. **Encrypt the Message**:
   - The `encrypt_message` function is called with the unencrypted string and the generated key as arguments. This function encrypts the message using XOR with the shared secret key and returns the encrypted message.
3. **Print Encrypted Message**:
   - The encrypted message is printed to the console.
4. **Decrypt the Message**:
   - The `decrypt_message` function is called with the encrypted message and the same key used for encryption. This function decrypts the encrypted message using XOR with the shared secret key and returns the decrypted message.

5. **Print Decrypted Message**:
    - The decrypted message is printed to the console.

Analysis

Impact of Eavesdropper on Encryption and Decryption

An eavesdropper, also known as an attacker or adversary, could significantly impact the encryption and decryption of data if they manage to intercept the communication channel between the sender and receiver. Here's how an eavesdropper could affect the security of the encryption and decryption process:

1. **Interception of Encrypted Data**: If an eavesdropper intercepts the encrypted data being transmitted between the sender and receiver, they may attempt to decrypt it using various cryptographic attacks. Depending on the strength of the encryption algorithm and key length, the attacker may succeed in decrypting the data and obtaining the original message.

2. **Interception of Key Exchange**: In the case of symmetric key encryption, where both parties use the same key for encryption and decryption, an eavesdropper may attempt to intercept the key exchange process. If successful, the attacker can obtain the secret key and decrypt all future communications encrypted with that key.

3. **Man-in-the-Middle Attacks**: In a man-in-the-middle (MITM) attack, the eavesdropper intercepts communication between the sender and receiver, posing as both parties. This allows the attacker to decrypt and read the messages, modify them, and even inject malicious content into the communication channel without the knowledge of the sender or receiver.

4. **Brute Force Attacks**: If the encryption key is weak or short, an eavesdropper may attempt a brute force attack to guess the key by trying all possible combinations. This is especially effective if the key space is small or if the encryption algorithm has vulnerabilities that can be exploited.

5. **Cryptanalysis**: Advanced attackers may employ cryptanalysis techniques to analyze the encryption algorithm itself and discover weaknesses or vulnerabilities that can be exploited to decrypt the data more efficiently than

brute force. This could involve exploiting flaws in the algorithm's design or implementation.

Overall, the presence of an eavesdropper poses a significant threat to the confidentiality and integrity of encrypted data. To mitigate these risks, it's essential to use strong encryption algorithms, ensure proper key management practices, authenticate communication parties, and employ secure communication protocols to protect against eavesdropping attacks.

Conclusion

Importance of Quantum Encryption

The development of quantum encryption is crucial for the future of secure communication due to several key reasons:

1. **Unbreakable Security**: Quantum encryption offers the promise of unbreakable security by leveraging the fundamental principles of quantum mechanics. Quantum key distribution (QKD) protocols, such as BB84, utilize the properties of quantum particles to create keys that are immune to eavesdropping attacks. This ensures the confidentiality and integrity of sensitive information in a way that is theoretically impossible to compromise.

2. **Resistance to Quantum Attacks**: With the advent of quantum computers, traditional encryption algorithms, such as RSA and ECC, are at risk of being broken by powerful quantum algorithms like Shor's algorithm. Quantum encryption provides a solution to this threat by offering cryptographic primitives that are resistant to quantum attacks. By harnessing quantum properties such as entanglement and superposition, quantum encryption schemes offer security guarantees that are not achievable with classical cryptography.

3. **Global Communication Security**: As digital communication becomes increasingly ubiquitous and interconnected on a global scale, ensuring the security of data transmission becomes paramount. Quantum encryption holds the potential to provide secure communication channels across vast distances, enabling confidential communication between parties separated by continents or even interplanetary distances. This has profound implications for sectors such as finance, healthcare, government, and defense, where data privacy and security are of utmost importance.

4. **Technological Advancements**: The development of quantum encryption technologies drives innovation in quantum information science and technology. Advancements in quantum computing, quantum communication, and quantum cryptography contribute to the broader field of quantum technologies, with applications ranging from secure communication

to quantum computing and quantum networking. Investing in quantum encryption research and development fosters technological progress and strengthens the capabilities of future quantum-enabled systems.

5. **Long-Term Security Solutions**: Quantum encryption offers long-term security solutions that are future-proof against advancements in computing power and cryptographic attacks. By harnessing the inherent randomness and unpredictability of quantum systems, quantum encryption protocols provide a robust foundation for securing sensitive information in an increasingly digital and interconnected world.

In summary, the development of quantum encryption is vital for ensuring the security, privacy, and integrity of digital communication in the face of evolving threats and technological advancements. By leveraging the principles of quantum mechanics, quantum encryption holds the promise of revolutionizing secure communication and shaping the future of cybersecurity.