Building a Quantum QR Code System

A Step-by-Step Guide to Future-Proof Data Transfer

---

1. Introduction: The Problem with QR Codes

Standard QR codes are the digital equivalent of a **postcard**. They are designed for convenience, not security. Anyone who scans a typical QR code can read its contents, making them completely unsuitable for transferring sensitive data like passwords, financial information, or private keys.

This project solves that problem. We have built a **Quantum-Resistant Secure QR Code System**.

Instead of a postcard, our QR code is an **armored, hybrid-locked box**. It is designed to be **confidential** (no one else can read it) and **future-proof** (it is secure against attacks from both today's computers and tomorrow's quantum computers).

This document details the complete, end-to-end process of building this system from scratch, explaining every component, term, and line of code.

---

2. Core Concepts: The Three Layers of Security

Our system is not a *quantum object*; it's a classical QR code that holds data secured by **three layers of modern cryptography**.

Layer 1: AES-256 GCM (The Message Lockbox)

- **What it is:** A **Symmetric Encryption** algorithm. This is a fast, powerful, and universally trusted standard for encrypting data.
- **Analogy:** A high-security steel safe. You lock your secret message inside this safe using a unique key (our aes_key). As long as only you and the receiver have that key, the message is secure.
- **Term: Symmetric** means the *same key* is used to both lock (encrypt) and unlock (decrypt) the data.

Layer 2: CRYSTALS-Kyber (The Key's Lockbox)

- **What it is:** A **Post-Quantum Cryptography (PQC)** algorithm and a **Key Encapsulation Mechanism (KEM)**, standardized by the U.S. National Institute of Standards and Technology (NIST).
- **The Problem it Solves:** How do you securely give the aes_key (the key to the steel safe) to your receiver? You can't just tape it to the outside.
- **Analogy:** Kyber is a special, "quantum-proof" mailbox.
  - The receiver has a **Public Key** (the mailbox slot, which anyone can see and use to drop a message in) and a **Private Key** (the *only* key in the universe that can open the mailbox).
  - We use the receiver's Public Key to lock our aes_key in a PQC "capsule" (ciphertext_kem).
- **Term: PQC** (Post-Quantum Cryptography) refers to classical encryption algorithms (like Kyber) that are mathematically designed to be secure against attacks from future, large-scale quantum computers.

Layer 3: QRNG (The Un-guessable Key)

- **What it is:** A **Quantum Random Number Generator**.
- **Why it's needed:** The strength of any encryption system depends on the randomness of its key. Standard computers use **Pseudo-Random Number Generators (PRNGs)**, which are just very complex mathematical formulas. In theory, they are predictable.
- **Our Solution:** We use the principles of quantum mechanics (via **Qiskit**) to generate a key. By measuring a qubit in superposition, we get a result that is **truly, fundamentally random** and physically unpredictable. This creates a provably un-guessable aes_key.

3. Project Blueprint: Generating the Secure QR Code (Sender's Side)

This phase details how to take a secret message and turn it into the final quantum_qr.png.

Step 1: Generate the Quantum-Random Key (QRNG)

We use Qiskit to simulate a 1-qubit circuit 256 times. This is fast, practical, and provides 256 bits of true randomness.

- **Tool:** qiskit
- **Concept:** H (Hadamard) gate creates superposition. Measurement collapses it to a 0 or 1 with 50/50 probability.

Step 2: Encrypt the Secret Message (AES)

We use the aes_key to lock our secret message inside the "steel safe."

- **Tool:** cryptography
- **Term: Nonce (Number used once):** A random value that ensures encrypting the same message twice results in different ciphertexts. It is not secret and is sent with the message.

Step 3: Create the Receiver's PQC Keys

The receiver must run this code *once* to generate their "mailbox" keys.

- **Tool:** pypqc
- **Concept:** The receiver generates a **Public Key** (which they share with the sender) and a **Secret Key** (which they *must* keep private).

Step 4: Encapsulate the Quantum Key (Kyber)

Now, the sender uses the receiver_public_key to securely package the aes_key.

- **Tool:** pypqc

- **Concept:** We use Kyber's encap function to generate a **new** shared secret (shared_secret_kem) and its "lockbox" (ciphertext_kem). We then "hide" our aes_key by XORing it with this new secret.

Step 5: Bundle Payload and Generate QR Code

Finally, we package all three binary components (ciphertext_kem, encrypted_aes_key, encrypted_data) into a single text string and create the image.

- **Tools:** json, base64, qrcode, pillow
- **Terms:**
  - **Base64:** A standard way to convert raw binary data into plain text strings.
  - **JSON:** A text format for structuring data.

---

4. Project Blueprint: Decrypting the Secure QR Code (Receiver's Side)

This phase represents the "secure scanner app" that the receiver uses.

Step 1: Scan and Parse the QR Code

The app scans the image, reads the JSON text, and decodes the Base64 strings back into raw bytes.

- **Tools:** pyzbar, Pillow, json, base64

Step 2: Decrypt the Quantum Key (PQC)

This is the most critical step. The receiver uses their **private receiver_secret_key** to unlock the "mailbox" (ciphertext_kem_bytes).

- **Tool:** pypqc
- **Concept:** The decap (decapsulate) function is the reverse of encap. It uses the secret key to re-generate the *exact same* shared_secret_kem that the sender used.

Step 3: Recover the Original AES Key

The receiver now reverses the XOR operation to get the original aes_key.

Step 4: Decrypt the Final Message (AES)

With the recovered aes_key, the receiver can now unlock the "steel safe" (enc_data_bytes) and read the secret message.

- **Tool:** cryptography

---

5. Technology Stack & Key Terms

A summary of all tools and concepts used in this project.

- **Programming Language:** Python 3
- **Libraries:**
  - qiskit / qiskit-aer: For Quantum Random Number Generation (QRNG) simulation.
  - pypqc: For Post-Quantum Cryptography (CRYSTALS-Kyber 768).
  - cryptography: For symmetric encryption (AES-256 GCM).
  - qrcode / pillow: For generating the QR code image.
  - pyzbar: For reading the QR code image.
  - json: For data structuring.
  - base64: For encoding binary data to text.
- **Key Terms:**
  - **PQC (Post-Quantum Cryptography):** Classical algorithms secure against quantum computers.
  - **KEM (Key Encapsulation Mechanism):** An algorithm for securely sharing a symmetric key using asymmetric (public/private) keys.
  - **QRNG (Quantum Random Number Generator):** A device/process that uses quantum mechanics for true, unpredictable randomness.
  - **AES-GCM:** A modern, authenticated symmetric encryption standard.

  - **Hybrid Encryption:** The technique of using fast symmetric encryption (AES) for the data and slower asymmetric encryption (Kyber) to protect the key.

---

6. Real-World Application & Company Use

This project is a powerful prototype for a real-world security product.

Important Distinction: Quantum-Resistant vs. Quantum Advantage

- **Quantum Advantage** is when a quantum computer *outperforms* any classical computer (e.g., breaking encryption). This is an *offensive* capability.
- **Quantum-Resistant (PQC)** is a *defensive* capability. It's a classical algorithm (like Kyber) that runs on classical hardware (your PC, your phone) but is safe from quantum attacks.

Pitch is **not** "We have a quantum computer." It is: **"We use post-quantum cryptography to secure your data from future quantum threats, and we use quantum-based randomness to ensure the highest level of security available today."**

Scalable Architecture

To turn this into a product, you would not have users manage keys in scripts. You would build a secure API.

1. **Backend Server (API):** Manages user accounts and stores all **public keys** in a database. All **private keys** are stored in a secure **Hardware Security Module (HSM)**. This server also connects via API to a **commercial Hardware QRNG** (e.g., from AWS, Quantinuum) instead of using the Qiskit simulator.
2. **Generator App (Web/Mobile):** A user types a message and selects a receiver. The app sends this to the API, which does all the encryption and sends back the QR code image.
3. **Decoder App (Mobile):** A user scans the QR code. The app sends the ciphertext_kem to the API. The API uses the HSM-stored private key to run decap, sending back *only* the recovered aes_key. The app then decrypts the message *locally*.

Use Cases:

- **Secure Offline Data Transfer:** Sending passwords, credentials, or configurations to air-gapped systems.
- **Anti-Counterfeiting:** Placing a secure QR code on a luxury item or pharmaceutical, which a customer can scan with a special app to verify its authenticity.
- **Verifiable Supply Chain:** Tracking high-value assets with a cryptographically secure and unforgeable digital signature.