QKD - BB84 protocol Demonstration

Anipriyo Mandal

May 2025

1 BB84 Protocol Overview according to the code implementation

1. **Key Generation:** The sender (Alice) either inputs or generates a random binary key intended for secure transmission to the receiver (Bob).

2. **Privacy Amplification and Encoding:** To enhance security and reduce any partial information an eavesdropper may have gained, privacy amplification is applied to the key. The resultant key is then encoded using the Steane $[[7, 1, 3]]$ CSS quantum error-correcting code, which allows the detection and correction of single-qubit errors during transmission.

3. **Quantum State Preparation:** For each bit of the encoded key, Alice randomly selects a basis (either computational(Z) or Hadamard(X)) to encode the bit into a quantum state. This basis selection is independent of Bob's choice and is essential for the security of the BB84 protocol.

4. **Quantum Transmission:** The encoded qubits are transmitted over a quantum communication channel. During this process, qubits may be disturbed due to environmental noise or active eavesdropping (e.g., measurement by an adversary).

5. **Measurement and Basis Guessing:** Upon receiving the qubits, Bob measures each one using a randomly chosen basis (computational(Z) or Hadamard(X)). Since his basis choices may not match Alice's, not all measured bits are guaranteed to match the original.

6. **Error Correction:** Bob applies quantum error correction decoding (using the Steane CSS code) to the measured qubits. This step corrects errors introduced by both channel noise and limited eavesdropping, helping to recover the original privacy amplified key.

7. **Repeat:(Currently not in implementation)** The steps from Quantum State Preparation to here can be repeated multiple times to get the total privacy amplified key since CSS error correction can't gurrantee a 100% key recovery

8. **Key Recovery and De-hashing:** After decoding, Bob applies the inverse of the privacy amplification procedure (e.g., de-hashing) to retrieve the final corrected version of the original key shared by Alice.

Algorithm 1: BB84 Protocol for Quantum Key Distribution

```
Function BB84QuantumKeyDistribution(n):
    /* Step 1:  Initial Key Generation                                   */
    rawKey ← GenerateRandomBits(n)    // Generate random bits
     for the key
    /* Step 2:  Privacy Amplification and Error Correction
       Encoding                                                          */
    amplifiedKey ← PrivacyAmplification(rawKey)    // Apply n
     to n-1 XOR hash function
    encodedKey ← SteaneEncode(amplifiedKey)    // Encode with
     Steane [[7,1,3]] CSS Code
    /* Step 3:  Quantum State Preparation                                */
    senderBasis ← GenerateRandomBases(length(encodedKey))
     // Alice chooses random basis
    quantumState ← QuantumEncoding(encodedKey, senderBasis)
     // Encode qubits according to basis
    /* Step 4:  Quantum Channel Transmission                             */
    transmittedState ← QuantumChannel(quantumState)
       // During transmission, some qubits may be intercepted
     (eavesdropping)
       // Channel noise adds additional errors to the quantum
     state
    /* Step 5:  Measurement at Receiver                                  */
    receiverBasis ← GenerateRandomBases(length(transmittedState))
     // Bob chooses random basis
    measuredBits ←
     QuantumDecoding(transmittedState, receiverBasis)  // Measure
     in chosen basis

    /* Step 6:Basis Reconciliation(NOT IN IMPLEMENTATION)  */
    basisComparison ← CompareBases(senderBasis, receiverBasis)
     // Public classical channel
    siftedKey ← SiftBits(measuredBits, basisComparison)   // Keep
     bits with matching basis

    /* Step 7:  Error Detection and Correction                           */
    errorPositions ← DetectErrors(measuredBits)          // Find
     positions with errors
    correctedKey ← SteaneDecode(measuredBits)  // Apply Steane
     code error correction
    /* Step 8:  Final Key Recovery                                       */
    finalKey ← KeyRecovery(correctedKey)   // Apply de-hashing
     to recover original key
    return finalKey
```

Algorithm 2: Encoding with Random Basis of Sender and Decoding with Guessed Basis of Reciever functions

```
Function QuantumEncoding(bits, basis):
    quantumState ← []
    for i ← 0 to length(bits) − 1 do
        if basis[i] = 0 then
            /* Z-basis encoding (Computational basis)        */
            if bits[i] = 0 then
                qubit ← |0⟩                      // Qubit in state |0⟩
            else
                qubit ← |1⟩                      // Qubit in state |1⟩
            end
        else
            /* X-basis encoding (Hadamard basis)             */
            if bits[i] = 0 then
                qubit ← |+⟩    // Qubit in state |+⟩ = (|0⟩+|1⟩)/2
            else
                qubit ← |−⟩    // Qubit in state |-⟩ = (|0⟩-|1⟩)/2
            end
        end
        quantumState.append(qubit)
    end
    return quantumState
Function QuantumDecoding(quantumState, basis):
    measurements ← []
    for i ← 0 to length(quantumState) − 1 do
        if basis[i] = 0 then
            /* Measure in Z-basis (Computational basis)      */
            result ← MeasureInZBasis(quantumState[i])
        else
            /* Measure in X-basis (Hadamard basis)           */
            result ← MeasureInXBasis(quantumState[i])
        end
        measurements.append(result)
    end
    return measurements
```

Algorithm 3: Privacy amplification and De-Hashing functions

```
Function PrivacyAmplification(key):
    /* Apply n to n-1 hash function for privacy amplification
       */
    amplifiedKey ← []
    for i ← 1 to length(key) − 1 do
        newBit ← key[i − 1] ⊕ key[i]                // XOR adjacent bits
        amplifiedKey.append(newBit)
    end
    return amplifiedKey    // Return the privacy amplified key
Function KeyRecovery(key):
    /* Reverse the privacy amplification (generalized)     */
    recoveredKey ← [0]               // Initialize with a seed bit
    for i ← 0 to length(key) − 1 do
        originalBit ← recoveredKey[i] ⊕ key[i]      // XOR to recover
          original bit
        recoveredKey.append(originalBit)
    end
    return recoveredKey               // Return the de-hashed key
```

1.1 Output

=== BB84 Quantum Key Distribution Protocol Demonstration ===

Step 1: Initial Key Generation

Initial non privacy amplified key: [1 0 0 1 0 0 0 1 1 0 0 1 0 1 1 0 1 0 1 1 0]

Privacy Amplified key length: 20 bits Initial privacy amplified key: [1 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 1 1 0 1]

Step 2: Quantum State Preparation

Encoded(for css error correction) quantum state length: 70 qubits Alice's basis choice: [1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 0 1 1 1 0 1 0 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0]

Step 3: Quantum Channel Transmission

Eve intercepts 3 qubits at positions: [4 32 16] Simulated channel conditions: - Eavesdropping rate: 5- Channel noise rate: 5

Step 4: Measurement and Key Recovery

Bob's basis choice: [1 0 1 0 1 0 1 0 0 1 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 0 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 0 0 0 0 1 1 0 0 0 1 1 1 0 0 1 0 1 0 0] Raw decoded key: [0 1 0 1 0 0 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0 1 0 0 1 1]

Step 5: Error Detection and Correction

Detected errors at positions: [5 7 12 15 26 30 40 42 54 61 67] Total errors detected: 11

Step 6: Final Key Analysis

Final key length: 20 bits Bit match rate: 70.00Successfully matched bits: 14 out of 20 Final key: [1 1 1 0 0 0 1 1 1 0 1 0 1 0 1 0 1 0 0 1] Init key : [1 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 1 1 0 1]

1.2 My Comments on the Output and Implementation

I have run the simulation many times but only 50-85% key can be recovered for a single iteration ...

So the sender(Alice) needs to send the key via this protocol implementation multiple times (At least 3) so that we can deduce the full key from the receiver(Bob's) side.

Therefore for every bit of the received error corrected but non hashed key, the actual bit value = majority occurrence bit for multiple transmission of the same bit.

Now once the actual value of the error corrected but non hashed key is obtained we use the de hashing function recieve the original key.

2 CSS Code Error Correction Algorithm: Steane [[7,1,3]] Code

2.1 Introduction

The Steane [[7,1,3]] code is a quantum CSS (Calderbank-Shor-Steane) error-correcting code that encodes 1 logical qubit into 7 physical qubits with a minimum distance of 3, allowing it to correct any single-qubit error. In our binary implementation for BB84, we use it to correct bit errors in classical bit strings.

2.2 CSS Construction

The CSS code construction uses two classical linear codes C_1 and C_2 such that $C_1 \subset C_2$. For the Steane code: - C_1 is a $[7, 3, 4]$ classical code - C_2 is a $[7, 4, 3]$ classical code (the Hamming code)

2.3 Generator and Parity Check Matrices

The Steane code uses the following matrices:
Generator Matrix for C_1:

$$G_1 = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Generator Matrix for C_2:

$$G_2 = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Parity Check Matrix for C_1:

$$H_1 = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Parity Check Matrix for C_2:

$$H_2 = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

2.4 Detailed Algorithms

2.4.1 CSS Encoding Process

Algorithm 4: CSS Encoding

Data: Input data bits d of length $k_2 - k_1$ (2 bits for Steane code)
Result: Encoded codeword of length n (7 bits for Steane code)

Create logical word w of length k_2 (4 bits for Steane);
Set the last $k_2 - k_1$ bits of w to the input data d;
`// First` k_1 `bits remain 0`
Compute encoded codeword $c = wG_2 \mod 2$;
return c;

2.4.2 Algorithm 2: Syndrome Table Construction

Algorithm 5: Build Syndrome Tables

```
Result: Syndrome lookup tables for X and Z errors
Initialize empty X-syndrome table T_X and Z-syndrome table T_Z;
// Handle single-bit errors
for i ∈ {0, 1, ..., n − 1} do
    Create error pattern e with a single 1 at position i;
    Compute X-syndrome s_X = e · H_2^T mod 2;
    Compute Z-syndrome s_Z = e · H_1^T mod 2;
    Store mapping s_X ↦ e in T_X;
    Store mapping s_Z ↦ e in T_Z;
end
// Add no-error case
Store mapping 0 ↦ 0 in both T_X and T_Z;
// Handle two-bit errors
for i ∈ {0, 1, ..., n − 2} do
    for j ∈ {i + 1, ..., n − 1} do
        Create error pattern e with 1s at positions i and j;
        Compute X-syndrome s_X = e · H_2^T mod 2;
        Compute Z-syndrome s_Z = e · H_1^T mod 2;
        if s_X not already in T_X then
            Store mapping s_X ↦ e in T_X;
        end
        if s_Z not already in T_Z then
            Store mapping s_Z ↦ e in T_Z;
        end
    end
end
return T_X, T_Z;
```

2.4.3 Algorithm 3: CSS Decoding Process

Algorithm 6: CSS Decoding

Data: Received noisy codeword r of length n (7 bits for Steane code)
Result: Decoded data bits of length $k_2 - k_1$ (2 bits for Steane code)
`// Error correction`
Compute X-syndrome $s_X = r \cdot H_2^T \mod 2$;
if *s_X is in X-syndrome table T_X* **then**
    Retrieve error pattern e from T_X for syndrome s_X;
else
    Find minimum weight error pattern e that satisfies $e \cdot H_2^T = s_X \mod 2$;
end
Correct the received word: $c = r \oplus e$;
`// Extract logical bits (specific to Steane code)`
$d_0 = c_3$; `// Bit at position 3`
$d_1 = c_5$; `// Bit at position 5`
return $[d_0, d_1]$;

2.4.4 Algorithm 4: Minimum Weight Error Correction

Algorithm 7: Minimum Weight Error Correction

Data: Received word r and parity check matrix H
Result: Minimum weight error pattern
Compute syndrome $s = r \cdot H^T \mod 2$;
`// Try weight-1 error patterns`
for $i \in \{0, 1, \ldots, n-1\}$ **do**
    Create error pattern e with a single 1 at position i;
    if $e \cdot H^T = s \mod 2$ **then**
        return e;
    end
end
`// Try weight-2 error patterns`
for $i \in \{0, 1, \ldots, n-2\}$ **do**
    for $j \in \{i+1, \ldots, n-1\}$ **do**
        Create error pattern e with 1s at positions i and j;
        if $e \cdot H^T = s \mod 2$ **then**
            return e;
        end
    end
end
`// If no match found`
return *Zero error pattern*;

2.4.5 Algorithm 5: Key Encoding

Algorithm 8: Key Encoding

Data: Original key k of arbitrary length
Result: Encoded key with error correction
Pad key k to length divisible by $(k_2 - k_1)$;
Split padded key into blocks of size $(k_2 - k_1)$;
for *each block* b **do**
    Encode b using CSS encoding (Algorithm 1);
    Append encoded block to result;
end
return *Concatenated encoded blocks*;

2.4.6 Algorithm 6: Key Decoding with Error Correction

Algorithm 9: Key Decoding

Data: Received key r with possible errors
Result: Decoded and error-corrected key
Pad received key r to length divisible by n;
Split padded key into blocks of size n;
for *each block* b **do**
    Decode and correct b using CSS decoding (Algorithm 3);
    Append decoded block to result;
end
Truncate result to original key length;
return *Decoded key*;

2.4.7 Algorithm 7: Error Identification

Algorithm 10: Error Identification

Data: Received key r with possible errors
Result: List of positions where errors were detected
Initialize empty list of error positions;
Pad received key r to length divisible by n;
Split padded key into blocks of size n;
for *each block* b *at index* i **do**
    Compute X-syndrome $s_X = b \cdot H_2^T \mod 2$;
    if $s_X \neq 0$ **then**
        if s_X *is in X-syndrome table* **then**
            Retrieve error pattern e from table;
        else
            Compute error pattern e using minimum weight correction;
        end
        for *each position* j *where* $e_j = 1$ **do**
            Append global position $(i \cdot n + j)$ to error positions list;
        end
    end
end
return *Sorted list of error positions*;