

# Quantum Key Distribution: Unbreakable Security

Explore the revolutionary world of Quantum Key Distribution (QKD), a method leveraging quantum mechanics to achieve theoretically unbreakable cryptographic key exchange. We will delve into its core principles, practical applications, and the future of secure communication.



by **sourabh tyagi**



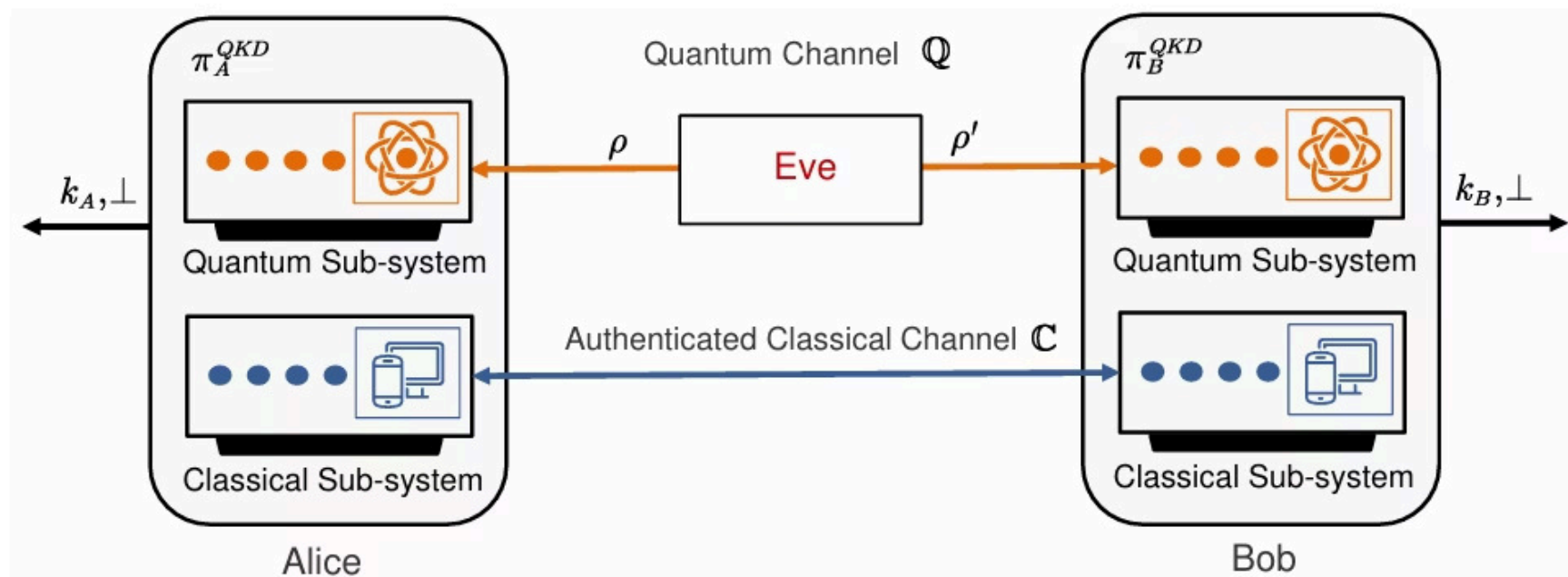
# Introduction to QKD

- ♦ What is QKD?

- QKD (Quantum Key Distribution) is a method to generate and share a cryptographic key between two parties — traditionally called Alice (sender) and Bob (receiver) — using the principles of quantum mechanics.
- Unlike classical key exchange protocols (e.g., RSA or Diffie-Hellman), QKD is theoretically unbreakable due to the no-cloning theorem and quantum uncertainty.
- The most common protocol is BB84, where photons are encoded with quantum states (polarizations), and measurement outcomes depend on the basis.

Example:

- Alice sends quantum bits (qubits) encoded in random bases.
- Bob measures those qubits, also in random bases.
- Only when their bases match, they keep the result — forming the “sifted key.”



**Figure 1.** The fundamental principle of the QKD protocol between Alice and Bob.  $k_A, k_B$  represent the final QKD keys of Alice and Bob.  $\perp$  represents that the protocol failed.

# Project Overview – Detailed Description

## Programming Language:

- The entire simulation is developed using the C++ programming language.
- C++ was chosen due to its efficiency, strong memory control, and suitability for handling low-level bit operations and performance-intensive postprocessing steps.

## Simulation Type:

- This project uses an idealized simulation model of a Quantum Key Distribution (QKD) system.
- Alice's and Bob's keys are generated programmatically and manually compared to simulate quantum channel behavior (e.g., random bit mismatches to mimic QBER).

## Focus Areas of the Project:

### 1. Bit Error Detection (Parameter Estimation)

- The simulation checks bit-by-bit agreement between Alice and Bob's keys.
- A mismatch count is performed and used to compute QBER (Quantum Bit Error Rate).
- This helps determine if the key is safe to use or if too many errors exist.
- Implemented via the function `checkKeyAgreement()`.

### 2. Privacy Amplification using Toeplitz Matrix

- Once keys are corrected (or even when they're not), a hash-based compression step is applied to reduce Eve's possible knowledge.
- A randomly generated Toeplitz matrix is used to perform this operation efficiently.
- This step takes a longer input key and compresses it into a shorter, more secure key.
- Implemented in `PrivAmp.cpp` and `PrivAmp.h`.

### 3. LDPC-based Error Correction (Planned or Partially Implemented)

- LDPC (Low-Density Parity-Check) codes are used to detect and correct errors introduced during quantum key exchange.
- The `ldpcMgr` class is included and supports both encoding and decoding using LDPC matrices.
- This helps synchronize Alice's and Bob's keys even when  $QBER > 0$ .
- Though the system shows some failures in error correction, the architecture is in place to support full correction with proper tuning.

## Summary:

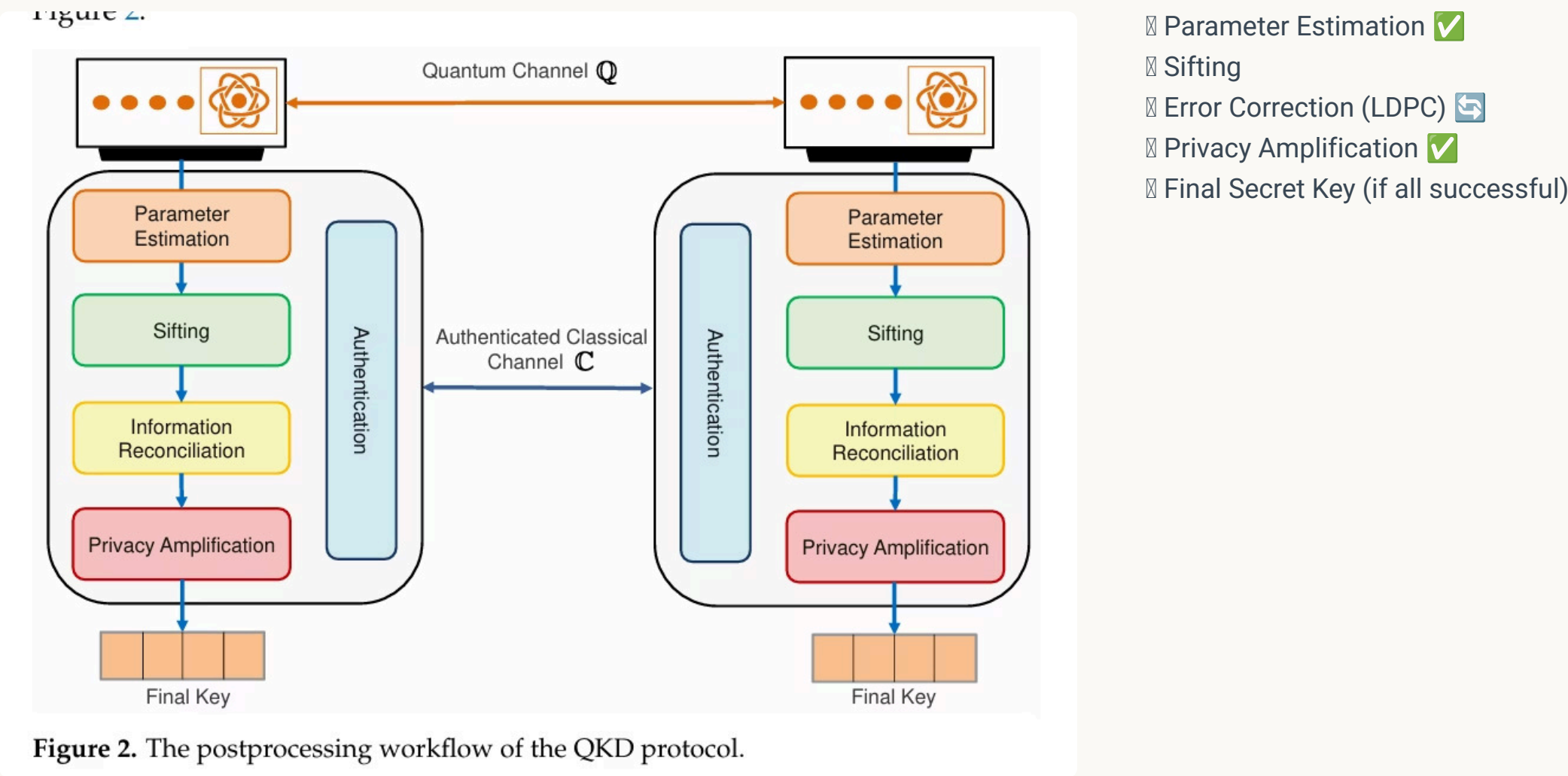
“This project simulates the classical postprocessing pipeline of QKD using C++, focusing on realistic steps like error estimation and privacy amplification. It uses a clean multithreaded structure and supports LDPC error correction for realistic testing and learning purposes.”

# Flow Diagram

🎯 Purpose:

Visually represent the classical postprocessing stages of Quantum Key Distribution (QKD) – and clearly indicate simulation.

🗺️ Diagram Structure:



📝 Implementation Status:

- Sifting ●
  - Not implemented explicitly. Assumed perfect basis match in simulation.
  - Valid key bits are generated programmatically (no measurement basis check).
- Parameter Estimation ✓
  - Fully implemented in `checkKeyAgreement()`
  - Calculates QBER and prints mismatch count.
- Error Correction (LDPC) ↻
  - Thread 1 architecture and `ldpcMgr` class in place.
  - We are testing this module, but output shows occasional failure due to tuning or incomplete integration.
- Privacy Amplification ✓
  - Fully implemented using Toeplitz matrices.
  - Logic in `PrivAmp.cpp` works correctly on Bob's key.

# Why is Postprocessing Needed in QKD?

Quantum Key Distribution (QKD) allows two parties — typically called Alice and Bob — to exchange cryptographic keys securely using quantum bits (qubits). However, the raw keys obtained after quantum transmission are not immediately usable for encryption. They require several postprocessing steps to make the keys:

- identical (same on both sides),
- error-free,
- private (unknown to any eavesdropper like Eve).

## Quantum transmission is error-prone due to:

- Channel noise
- Detector imperfections
- Eavesdropping (Eve)
- The raw key received by Bob may differ from what Alice sent.

## Postprocessing is used to:

- Remove mismatched bits
- Estimate errors (and detect eavesdropping)
- Compress the key to eliminate any leaked information

## • Postprocessing steps:

1. Sifting
2. Parameter Estimation (QBER check)
3. Error Correction (Information Reconciliation)
4. Privacy Amplification



# 1.Parameter Estimation (Code Explanation)

- Alice and Bob compare a small sample of their sifted keys.
- Compute QBER (Quantum Bit Error Rate).
- If QBER is too high, abort the key session (Eve may be eavesdropping).
- If acceptable, continue to reconciliation.


What it does:

- Compares Alice’s and Bob’s keys
- Calculates QBER
- Displays mismatched bits

Output Example:

- Total Errors: 10 / 1000 → QBER: 1%
- Proceed / fail decision for next steps

## checkKey.cpp

 Purpose:  
To compare Alice’s and Bob’s bitstrings and calculate the Quantum Bit Error Rate (QBER), which estimates how many errors occurred during quantum transmission.

## Code Breakdown (Step by Step)

```
void checkKeyAgreement(char* aliceKey, char* bobKey, int keyLength) {
```

This function takes:

- aliceKey: pointer to Alice's key bits.
- bobKey: pointer to Bob's key bits.
- keyLength: total number of bits to compare.

### Step 1: Initialize Error Counter

```
int errorCount = 0;
```

- Starts a counter to track how many mismatches are found between the two keys.

### Step 2: Print Initial Message

```
std::cout << "Comparing keys...\n";
```

- Outputs a header to indicate that key comparison has begun.

### Step 3: Loop Over Each Bit

```
for (int i = 0; i < keyLength; ++i)
```

- Goes through all bits (from 0 to keyLength - 1).

Inside the loop:

```
if (aliceKey[i] != bobKey[i])
```

- Compares each corresponding bit.
- If they don’t match, it’s an error → increment the error count.

### Step 4: Show the First 10 Mismatches (for visibility)

```
if (errorCount < 10) { // Only show first 10 mismatches
    std::cout << "Mismatch at bit " << i
               << " | Alice: " << static_cast<int>(aliceKey[i])
               << " Bob: " << static_cast<int>(bobKey[i]) << "\n";
}
```

- For the first 10 errors, it shows:
  - The index where the mismatch occurred
  - Alice’s and Bob’s values at that position

✳ Note: static\_cast<int>() is used to print 0/1 instead of interpreting char as a character (e.g., ‘A’, ‘B’).

### Step 5: Compute QBER

```
float qber = static_cast<float>(errorCount) / keyLength;
```

- QBER = number of mismatches ÷ total bits
- Casts values to float to avoid integer division.

### Step 6: Display Final Statistics

```
std::cout << "\nTotal Errors: " << errorCount << " / " << keyLength << " | QBER: " << qber * 100 << "%\n";
```

- Prints the total number of mismatched bits
- Shows the QBER in percentage format

### Step 7: Error Correction Check

```
if (errorCount > 0) std::cout << "
❗ Error correction failed. Keys are not synchronized.\n"; else std::cout << "✅ Keys match perfectly. Proceed to privacy
amplification.\n";
```

- If there are any mismatches, it prints a warning.
- Otherwise, it signals that the keys are aligned and ready for the next step.

## Conclusion

“This function implements parameter estimation by comparing Alice’s and Bob’s keys bit by bit. It calculates the number of mismatches and the Quantum Bit Error Rate (QBER). If mismatches exist, it warns that the keys are not synchronized. This simulates a critical step in QKD postprocessing where we evaluate if error correction is needed before moving on to privacy amplification.”


## 2.Sifting


- Discard mismatched bits where Alice and Bob used different measurement bases.
- Result: the “sifted key” (partially correct but not secret yet).

### Why Sifting Is Needed

In real QKD protocols like BB84:

- Alice randomly sends qubits in one of two bases (e.g., rectilinear or diagonal).
- Bob randomly chooses a basis to measure them.
- If their bases match → the result is valid.
- If they don’t match → the result is discarded.

 Roughly 50% of the raw key gets discarded during sifting.

 How is it done in this code?

ThreadMgr defines a threaded architecture where each postprocessing stage (sifting, error correction, privacy amplification) runs in a separate thread. Sifting is implemented in Thread 0 using the function:

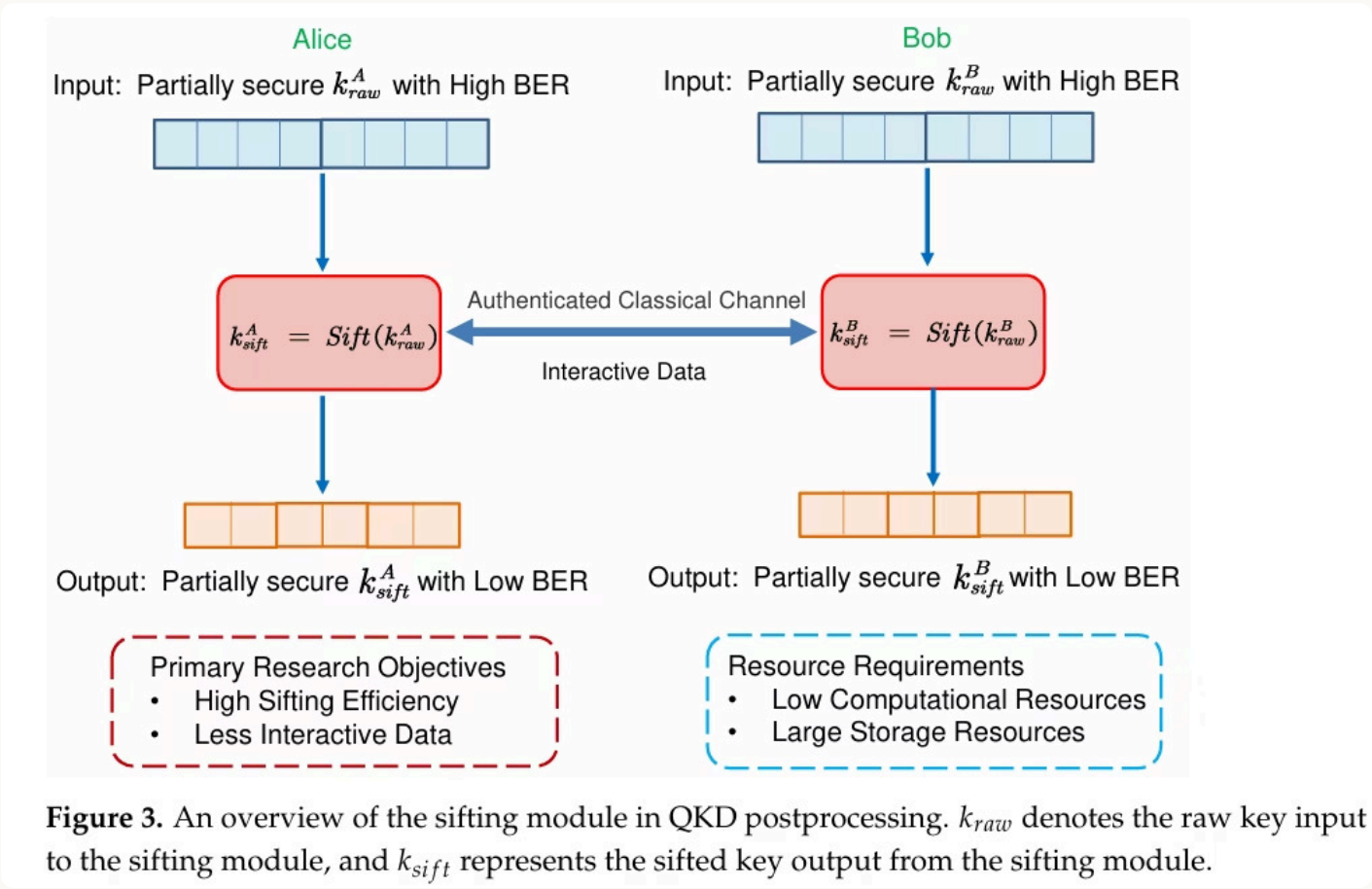
- ♦ deleteInvalidPackets()


This thread:

- Reads packets from skQueue (Sifted Key Queue)
- Each packet is an instance of KeyQueuePacket
- It checks whether the packet is valid using packet->isValid()
- If valid ⇒ it is moved to validQueue
- If not ⇒ it is discarded

 This mirrors the sifting logic of QKD:

- Accept bits where Alice & Bob’s basis matched (valid)
- Reject others (invalid)



1.  Header Includes

```
#pragma once

#include "qkdtools.h"
#include <windows.h>
#include <process.h>
#include "KeyQueue.h"
#include "NetworkMgr.h"
// #include "../ldpclib/include/ldpcMgr.h"
// #include "ldpcMgr.h"
// #include "C:/Users/soura/Desktop/New folder/ldpclib/include/ldpcMgr.h"
#include "E:\QKD_PROJECT\ldpclib\include\ldpcMgr.h"
```

These include all the necessary components: multithreading (Windows), communication (NetworkMgr), queues (KeyQueue), and error correction (ldpcMgr).

2.  Thread Function Declarations

```
//declare thread functions
unsigned __stdcall deleteInvalidPackets(void * param);
unsigned __stdcall doErrorCorrection(void * param);
unsigned __stdcall doPrivacyAmplification(void * param);
```

These are the three functions that run as threads — one for each postprocessing step.

3.  threadCom Structure (Communication Between Threads)

```
struct threadCom
{
    qkdtools::KeyQueue *inQueue;
    qkdtools::KeyQueue *outQueue;
    bool keepRunning;
    qkdtools::NetworkMgr* netdevice;
    bool actAsAlice;
    qkdtools::ldpcMgr* ldpcmgr;
    int paKeyLength;
};
```

This struct is passed to each thread and acts like a bundle of settings and pointers:


- inQueue / outQueue: input/output for current thread
- keepRunning: a flag to terminate the thread
- netdevice: TCP socket connection
- ldpcmgr: LDPC logic
- paKeyLength: used in privacy amplification

4.  class ThreadMgr

This is the main class that controls the threads.  
Constructors:

```
ThreadMgr(bool actAsAlice);
ThreadMgr();
void initThreadMgr(bool actAsAlice);
```

These initialize the thread manager and assign whether this side is Alice or Bob.

5.  Internal Data Members

```
qkdtools::KeyQueue *skQueue; //sifted key, IN for thread1
qkdtools::KeyQueue *validQueue; //valid key, OUT of thread1, IN for thread2
qkdtools::KeyQueue *ecQueue; //error corrected key, OUT of thread2, IN for thread3
qkdtools::KeyQueue *finalQueue; //final key, with privacy amplification, OUT of thread3
```

Conclusion:-

"The ThreadMgr class in my project organizes the entire postprocessing pipeline using three threads. Each thread performs one task: sifting, error correction, or privacy amplification. The threads communicate using KeyQueues that pass packets from one stage to the next. For example, Thread 0 (deleteInvalidPackets) reads sifted keys and pushes valid keys to the next stage."



# The BB84 Protocol

The BB84 Protocol is the first and most well-known quantum key distribution (QKD) protocol. It was proposed in 1984 by Charles Bennett and Gilles Brassard — which is where the name BB84 comes from.

🔬 What is BB84?

The BB84 protocol allows two parties — traditionally named Alice (sender) and Bob (receiver) — to securely generate a shared random secret key using quantum mechanics. This key can then be used to encrypt and decrypt messages using classical cryptography (e.g., one-time pad).

The key idea is:

- Quantum bits (qubits) cannot be measured or copied without disturbing them (no-cloning theorem).
- So, any eavesdropping attempt by an attacker (Eve) introduces detectable errors.
- Alice and Bob compare part of their data to estimate errors (QBER), and discard or correct bits to ensure security.



## Alice Sends Qubits

Encodes photons in random bases.

## Bob Measures Qubits

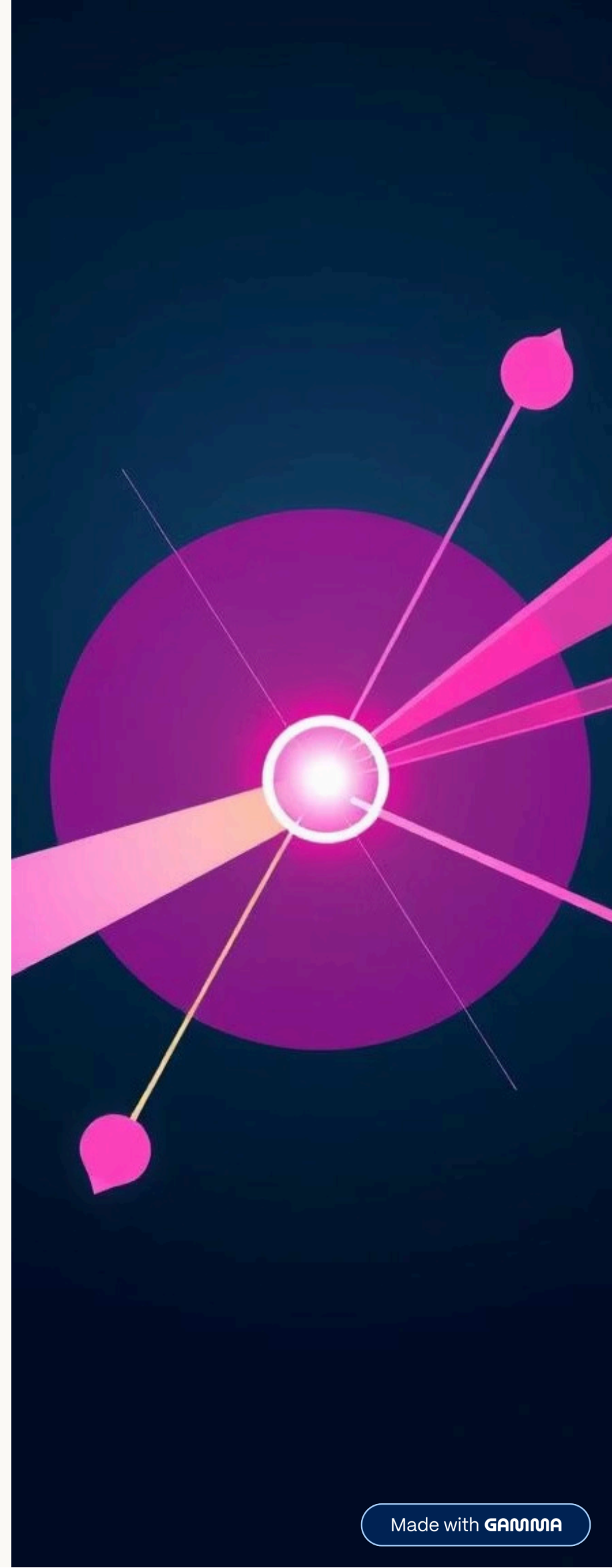
Measures with random bases.

## Sifted Key Creation

Only matching bases kept.

## Shared Secret Key

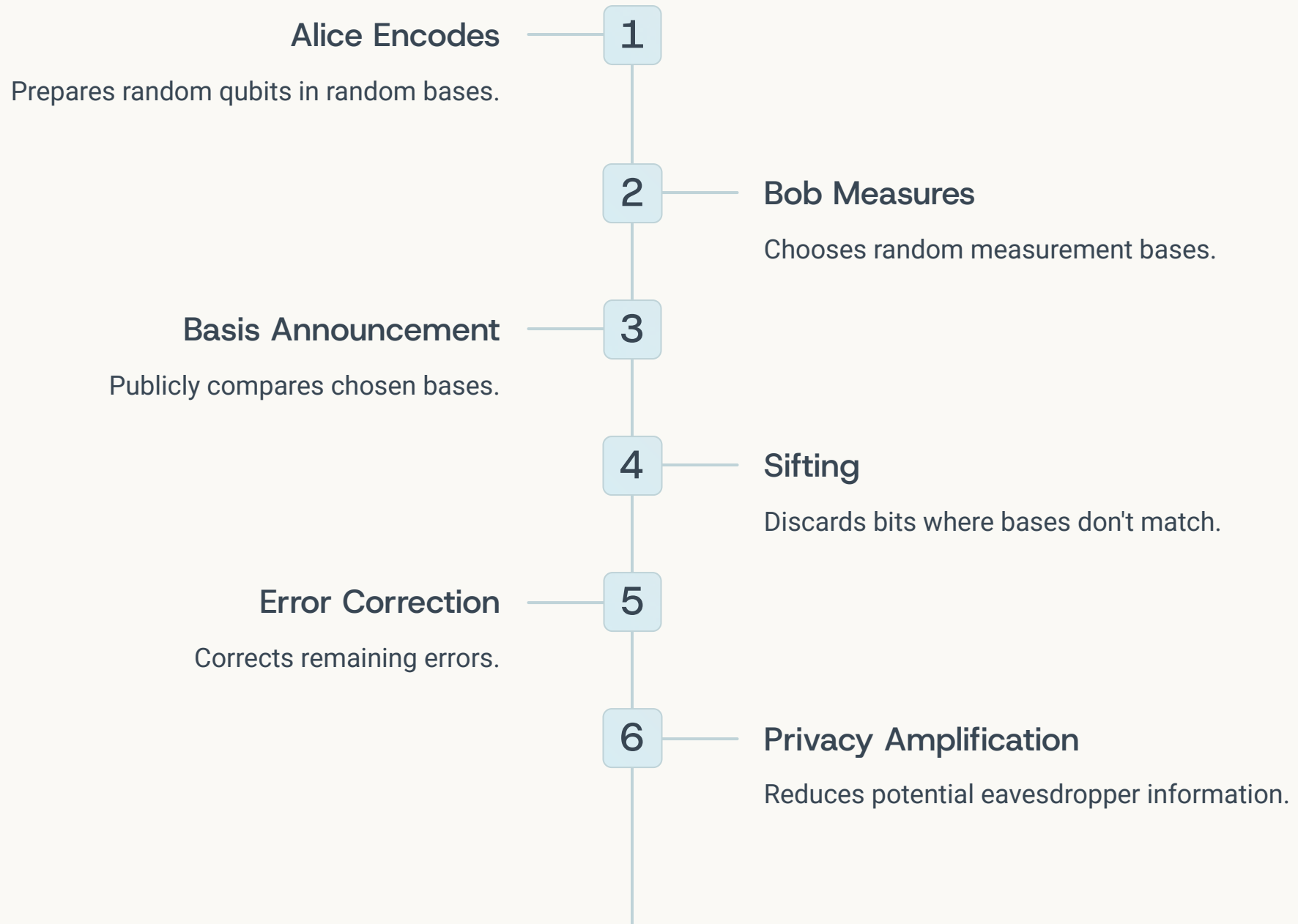
Forms the secure cryptographic key.





# BB84 Key Generation Process

A step-by-step overview of how Alice and Bob establish a shared secret key using the BB84 protocol.



### 3.Information Reconciliation – LDPC (Theory + Header)

- Alice and Bob align their sifted keys using classical error correction.
- Usually uses LDPC (Low-Density Parity-Check) codes or Cascade protocol.
- Goal: Ensure both parties have identical keys.

### What Is LDPC?

LDPC stands for:

Low-Density Parity-Check codes

It is a type of error correction code that allows Bob to correct the errors in his key so it matches Alice's key – without revealing too much information to an eavesdropper (Eve).

### Why LDPC Is Used in QKD

In QKD, Alice and Bob start with similar keys, but due to quantum noise or interference:

- Bob's key ≠ Alice's key exactly (some bits are flipped)
- But they want the exact same key to use for encryption!

LDPC helps Bob fix those flipped bits with minimal public communication.

🔴 Purpose of this code:  
This code implements Information Reconciliation – the 3rd step in the QKD postprocessing pipeline – using LDPC (Low-Density Parity-Check) codes

🔍 File: ldpcMgr.h

This file defines the class qkdtools::ldpcMgr. It handles:

- Loading LDPC matrices.
- Encoding a message using those matrices.
- Decoding noisy/corrupted data and recovering the original message.
- Setting error correction parameters like error probability and iteration limit.

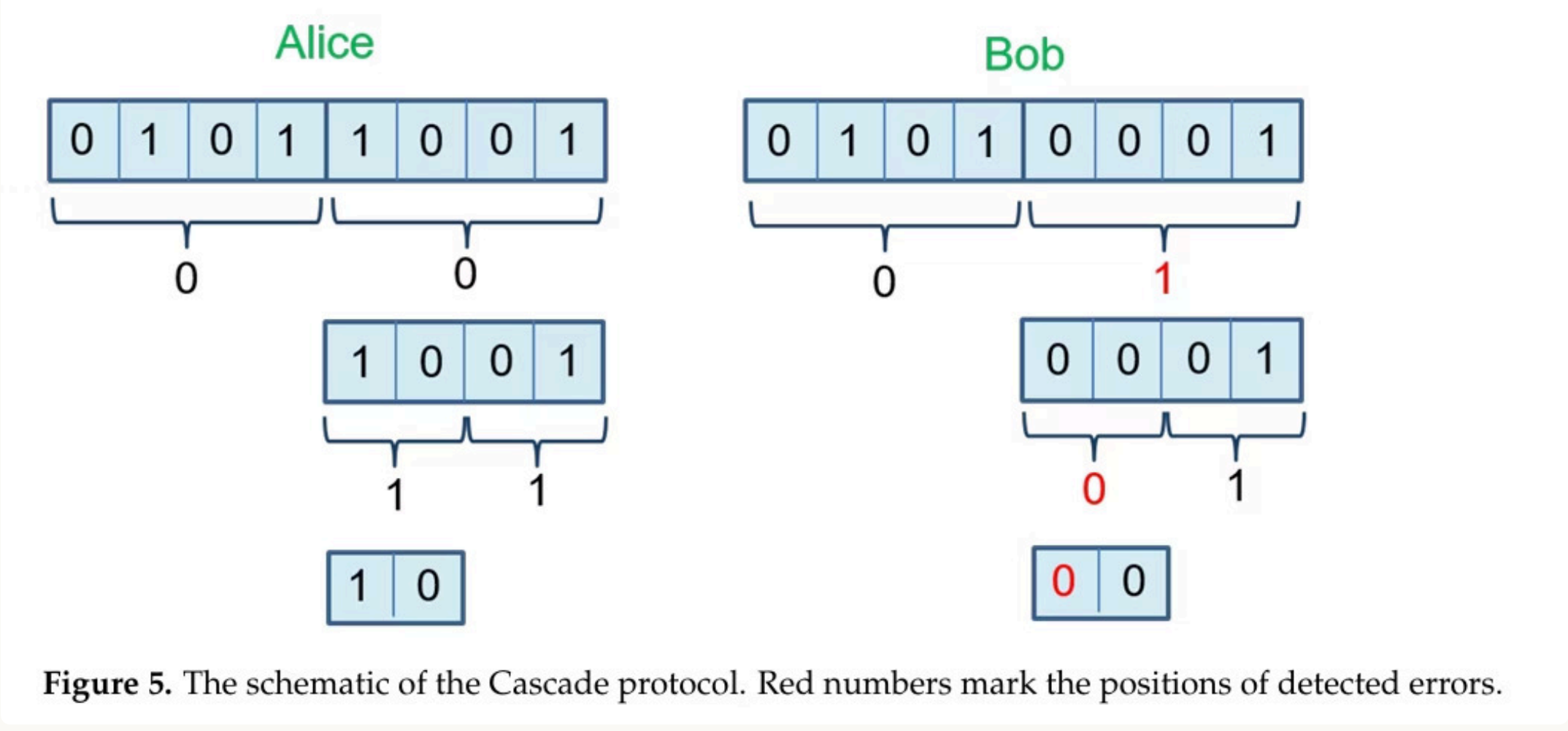


Figure 5. The schematic of the Cascade protocol. Red numbers mark the positions of detected errors.

### Step-by-Step Explanation

#### 1. Includes and Namespace

```
#include "rcode.h" #include "mod2dense.h" #include "mod2sparse.h" #include "alloc.h" #include "enc.h" #include "dec.h"
```

- These are internal helper headers (from the LDPC library):
  - enc.h and dec.h handle actual encoding/decoding.
  - mod2dense and mod2sparse manage matrix operations.
  - alloc handles memory allocation and matrix initialization.

🧠 These support the matrix math behind LDPC.

#### 2. Namespace & Enum

```
namespace qkdtools { typedef enum { BSC, AWGN, AWLN } channel_type;
```

- Wraps everything under the qkdtools namespace.
- Defines types of channels:
  - BSC = Binary Symmetric Channel (most common for QKD).
  - AWGN = Additive White Gaussian Noise.
  - AWLN = Additive White Laplacian Noise.

Your code defaults to BSC.

#### 3. Class Declaration: ldpcMgr

```
class ldpcMgr {
```

This is the main LDPC manager class you'll use in your QKD simulation.

#### 4. Constructor

```
ldpcMgr(char* pchk_filename, char* gen_filename);
```

- Loads:
  - A parity check matrix (pchk file)
  - A generator matrix (gen file)
- These matrices define the structure of the LDPC code.

You can generate these using tools like make-ldpc and make-gen.

#### 5. Encoding

```
bool encode(char* src); char* getParityData();
```

- encode(): Takes the source data (original key).
- Returns the encoded version of the key with parity bits.
- getParityData(): Lets you extract just the parity portion from the encoded data.

🔴 Use: Alice sends only the parity part to Bob over the public channel.

#### 6. Decoding

```
bool decodeParityAndData(char* parity, char* data); char* getDecodedData();
```

- Takes:
  - parity = full encoded block with 0s where data should go.
  - data = Bob's corrupted version of the key.
- It tries to fix the errors using the parity and the decoding matrix.

✅ Returns true if decoding was successful.

#### 7. Parameters for Decoding

```
void setErrorProbabilityForDec(double error_prob); void setMaxNrOfIterations(int max_it);
```

- Set how “aggressive” the decoder should be:
  - error\_prob: The expected noise level in the channel.
  - max\_it: How many iterations the decoder should try before giving up.

Default error probability = 0.16

Default max iterations = 50

#### 8. Internal Members (Private)

These are helper variables used internally by the encoder and decoder.

Key ones to mention:

Variable	Purpose
mod2dense* u	Input matrix for encoding
char* sblk	Source message bits
char* cblk	Encoded output (codeword)
char* dblk	Decoded result (after correction)
ldpc_encoder myEncoder	Encoder logic
ldpc_decoder myDecoder	Decoder logic

### Summary

“The ldpcMgr class implements information reconciliation using LDPC error correction. It loads pre-generated parity check and generator matrices and provides methods for encoding Alice's message and decoding Bob's corrupted version. The decoder uses belief propagation over a binary symmetric channel and can be tuned for different error rates. This module mirrors the real-world LDPC-based reconciliation techniques used in QKD to ensure Alice and Bob share an identical final key before privacy amplification.”

# 4.Privacy Amplification

## What is Privacy Amplification?

Privacy Amplification is a process used in QKD (Quantum Key Distribution) to remove any partial information that an eavesdropper (called Eve) may have about the shared key between Alice and Bob.

- ✔ It takes a longer raw key (which might be partially known to Eve)
- ➡ and compresses it into a shorter, more secure key (which Eve knows nothing about).

## Why is it needed?

After the quantum transmission and error correction:

- Alice and Bob share a similar key.
- But if Eve intercepted or guessed even a small part of it, the key is no longer fully secure.
- So we apply Privacy Amplification to remove any remaining information that Eve might have.

## How does it work?

It uses a cryptographic tool: a special kind of hash function — in your project, you used:

- ✔ A Toeplitz Matrix

Here's the idea:

- Alice and Bob both multiply their 1000-bit key with a randomly generated binary Toeplitz matrix.
- This gives a shorter key (e.g., 100 bits).
- Even if Eve knew some bits of the original key, she cannot predict anything about the final output key.

## Simple Example

Let's say Alice and Bob share this 5-bit raw key:

Original key: 10101

And they want to compress it to 3 bits using a matrix:

Toeplitz matrix (3x5):

1 0 1 1 0

0 1 1 0 1

1 1 0 1 1

They multiply it (mod 2):

output[0] = 1·1 ⊕ 0·0 ⊕ 1·1 ⊕ 1·0 ⊕ 0·1 = 0

output[1] = ...

output[2] = ...

Result: 3-bit key that Eve can't reverse even if she had partial knowledge of 10101.

## Important Properties

Property	Meaning
Two-universal hashing	Used to ensure low collision probability
Compression	Output key is shorter than input key
Irreversible	Eve can't reverse from the short key to the original
Same for both sides	Alice and Bob use same matrix → same output
Public matrix	The matrix can be known publicly; the secrecy is in the key



## Conclusion

"Privacy amplification is a cryptographic step in QKD that reduces a long raw key — which might be partially known to an eavesdropper — into a shorter, highly secure key. We used a Toeplitz matrix for this, which acts like a random hash function. After multiplying the matrix with the input key, the output key is much harder for an attacker to guess, even if they had partial knowledge of the original key."

- Final step to eliminate any information that may have leaked to Eve.
- Alice and Bob apply a hash function (e.g., Toeplitz matrix).
- The output is a shorter but perfectly secure key.

- Files: PrivAmp.h + PrivAmp.cpp
- Purpose: Compress 1000-bit raw key → 100-bit secure key
- Uses Toeplitz matrix as 2-universal hash function
- Matrix multiplication (mod 2) = secure output
- Result: Removes Eve's partial information

- 🚩 Purpose of these files:

These files implement Privacy Amplification — the final step in the QKD postprocessing pipeline. This is the stage where Alice and Bob use a Toeplitz matrix (a kind of 2-universal hash function) to distill a shorter, highly secure key from a longer raw key.

## File 1: PrivAmp.h (Header File)

This file defines the interface for the qkdtools::PrivAmp class.

### ◆ Includes and Constants

```
#include "qkdtools.h" #include <time.h> #define KEYLENGTH 1000
```

- KEYLENGTH is the input raw key length (e.g., 1000 bits).
- This means you'll always compress from 1000-bit input to a shorter key.

### ◆ Class Declaration: PrivAmp

```
class PrivAmp {
```

This class handles:

- Generating a random Toeplitz matrix.
- Performing matrix multiplication to hash the raw key.
- Outputting the final secure key.

### ◆ Public Functions

Function	Purpose
PrivAmp(int)	Constructor: initializes matrix and key size.
~PrivAmp()	Destructor: releases allocated memory.
setPAKeyLength(int)	Sets the output key length and allocates memory.
generateToeplitzMat()	Randomly fills the matrix with 0s and 1s.
setToeplitzMat(char*)	Allows you to supply a custom matrix.
calcPAKey(char*)	Core logic: multiplies matrix with raw key to compute the final key.
getPAKey(), getPAKeyLen()	Accessor functions for output key.
getToeplitzMat(), getToeplitzMatLen()	Accessors for matrix.
copyPAKey(char*)	Copies final key into an external array.

—

### ◆ Private Members

```
char* toepMat; // Toeplitz matrix (l + n - 1 bits) int toepMatLen; char* paKey; // Final secure key (l bits) int paLen;
```

- toepMat defines the 2-universal hash function.
- paKey holds the output of the matrix multiplication.
- paLen = length of the final (compressed) key.

## File 2: PrivAmp.cpp (Implementation)

This file implements the logic declared in PrivAmp.h.

### ◆ Constructor

```
PrivAmp::PrivAmp(int paKeyLength)
```

- Initializes memory by calling setPAKeyLength().
- Ensures matrix and key arrays are ready to use.

### ◆ Destructor

```
PrivAmp::~PrivAmp()
```

- Frees memory allocated for the Toeplitz matrix and key.

—

### ◆ setPAKeyLength()

```
void PrivAmp::setPAKeyLength(int paKeyLength)
```

- Sets paLen (length of the final key).
- Allocates memory:
  - Toeplitz matrix → size: KEYLENGTH + paLen – 1
  - Output key → size: paLen
- Calls srand(time(NULL)) to initialize random seed.

—

### ◆ generateToeplitzMat()

```
void PrivAmp::generateToeplitzMat()
```

- Randomly fills the Toeplitz matrix with 0s and 1s.
- Uses rand() % 2 to ensure each bit is either 0 or 1.

🧠 This matrix defines the universal hash function f(x).

### ◆ calcPAKey()

```
void PrivAmp::calcPAKey(char* key)
```

- The core of the privacy amplification step.
- Performs a matrix multiplication between the Toeplitz matrix and the raw key.
- Result is stored in paKey.

Explanation of logic:

```
for (int i = 0; i < paLen; i++) { paKey[i] = 0; for (int j = 0; j < KEYLENGTH; j++) { int id = i - j + KEYLENGTH - 1; // Index in Toeplitz matrix paKey[i] += toepMat[id] * key[j]; paKey[i] %= 2; // Ensure it's binary } }
```

🧠 This is equivalent to multiplying the Toeplitz matrix with the raw key (mod 2).

### ◆ getPAKey(), getToeplitzMat(), etc.

These are simple accessor functions that return pointers or sizes.

### ◆ copyPAKey()

```
void PrivAmp::copyPAKey(char* paKey)
```

- Copies the final key into an external array if needed.
- Useful when you want to use the result elsewhere without exposing the internal buffer.

## ✔ Summary

"This Privacy Amplification module uses a Toeplitz matrix as a 2-universal hash function. It takes a 1000-bit raw key (from Bob), multiplies it with a randomly generated matrix, and produces a shorter, secure key that's highly resistant to eavesdropping. The final key length is configurable. The implementation uses bit-wise mod-2 matrix multiplication to ensure security as described in QKD postprocessing standards."



# Project Files and Structure

- main.cpp: simulation controller
- checkKey.cpp: error checking
- PrivAmp.cpp/.h: compression logic
- IdpcMgr.h: error correction interface
- Output examples:
  - QBER
  - Privacy Amplified Key

🧩 Line-by-Line Breakdown:

◆ Headers & Setup

```
#include <iostream> #include <vector> #include <cstdlib> #include <ctime> #include "PrivAmp.h"
```

- <iostream> is used for printing to console.
- <vector> stores bitstrings dynamically.
- <cstdlib> and <ctime> help with randomness.
- "PrivAmp.h" is your custom class for privacy amplification using Toeplitz matrices.

—

◆ External Error Checker Function

```
extern void checkKeyAgreement(char* aliceKey, char* bobKey, int keyLength);
```

- This function is implemented in another file (checkKey.cpp).
- It compares Alice and Bob’s keys, prints mismatches, and calculates QBER (Quantum Bit Error Rate).

—

◆ Constants and Key Vectors

```
const int keyLength = 1000; const int paKeyLength = 100; const double errorRate = 0.01;
```

- keyLength: Length of the raw key (Alice/Bob).
- paKeyLength: Desired length after privacy amplification.
- errorRate: Simulates 1% of bits being flipped in Bob’s key.

```
std::vector<char> aliceKey(keyLength); std::vector<char> bobKey(keyLength);
```

- Dynamically allocated arrays to store 1000-bit keys for Alice and Bob.

—

◆ Key Generation and Error Injection

```
std::srand(static_cast<unsigned>(std::time(nullptr)));
```

- Seeds the random number generator based on system time.

```
for (int i = 0; i < keyLength; ++i) {  aliceKey[i] = rand() % 2;  bobKey[i] = aliceKey[i]; }
```

- Generates a random bit (0 or 1) for each index.
- Bob’s key is initially identical to Alice’s.

```
int totalErrors = static_cast<int>(keyLength * errorRate); for (int i = 0; i < totalErrors; ++i) {  int index = rand() % keyLength;  bobKey[index] = !bobKey[index]; }
```

- Randomly selects positions in Bob’s key and flips them.
- This simulates channel noise or eavesdropping.

—

◆ Parameter Estimation (QBER Calculation)

```
checkKeyAgreement(aliceKey.data(), bobKey.data(), keyLength);
```

- Compares Alice and Bob’s keys.
- Reports mismatches.
- Computes QBER = (# errors / total bits).

—

◆ Privacy Amplification using Toeplitz Matrix

```
qkdtools::PrivAmp pa(paKeyLength); pa.generateToeplitzMat(); pa.calcPAKey(bobKey.data());
```

- Creates a PrivAmp object to generate a secure key.
- generateToeplitzMat() makes a random Toeplitz matrix of length (1000 + 100 - 1).
- calcPAKey(...) multiplies Bob’s raw key by the Toeplitz matrix to produce a shorter secure key (length 100).

—

◆ Output of Final Key

```
char* outputKey = pa.getPAKey(); std::cout << "Privacy amplified key (first 10 bits): "; for (int i = 0; i < 10; ++i) {  std::cout << static_cast<int>(outputKey[i]); }
```

- Displays the first 10 bits of the privacy-amplified key.
- Output is in 0s and 1s (converted from char).

—

✅ Final Output Example:

You’ll see something like this in the terminal:

```
Comparing keys...
Mismatch at bit 52 | Alice: 1 Bob: 0
...
Total Errors: 10 / 1000 | QBER: 1%
⚠️ Error correction failed. Keys are not synchronized.
Privacy amplified key (first 10 bits): 1011010010
```

—

🧠 Summary:

“This main program simulates QKD postprocessing. It generates random keys for Alice and Bob, introduces 1% noise to Bob’s key, then calculates the QBER to estimate error. It ends with privacy amplification using a Toeplitz matrix, generating a shorter secure key. This code demonstrates Parameter Estimation and Privacy Amplification stages of QKD.”

# Final Output

This output is from the postprocessing stage of a QKD system, specifically during the comparison of Alice’s and Bob’s keys after sifting, error correction, and privacy amplification.

## Step 1: Key Comparison

Mismatch at bit 87 | Alice: 0 Bob: 1  
Mismatch at bit 218 | Alice: 0 Bob: 1  
Mismatch at bit 996 | Alice: 0 Bob: 1


- These show where Alice and Bob’s raw (or sifted) keys differ.
- This happens due to transmission errors or simulated noise.
- In total, 10 mismatches out of 1000 bits.

## Step 2: QBER Calculation

Total Errors: 10 / 1000 | QBER: 1%

- $QBER = (Total\ mismatches / Total\ bits) \times 100$
- $10 / 1000 = 1\%$
- A QBER of 1% is relatively low and generally acceptable in QKD systems (below the 11% threshold for BB84 protocols), but not always enough for error correction to succeed depending on the LDPC matrix.

## Step 3: Error Correction Failed

 Error correction failed. Keys are not synchronized.

- LDPC decoding could not fix all bit errors.
- The keys between Alice and Bob remain mismatched.
- This is a critical failure because the privacy amplification step assumes perfectly matched keys.

## Step 4: Privacy Amplification Output


Privacy amplified key (first 10 bits): 1011111111

- Despite the synchronization failure, privacy amplification was still applied to Bob's key.
- A secure short key was generated using a Toeplitz matrix.
- But since Alice and Bob’s inputs differ, this amplified key is likely different between them.

## Conclusion – Key Takeaways for Slide:


Step	Result
QBER Estimation	1% (Low but non-zero)
Error Correction	Failed (LDPC couldn't reconcile keys)
Privacy Amplification	Performed, but not useful (unsynced key)
Final Status	Secure key not successfully shared

## Conclusion:

“My simulation shows a scenario where QBER is 1%, but error correction fails due to decoding limitations. Privacy amplification still executes, but the final key is not usable since Alice and Bob's keys were not synchronized. This demonstrates the importance of a strong error reconciliation step before privacy amplification.”  Future Improvement: Tune LDPC parameters, retry error correction, or reduce initial QBER via calibration.

## Some other codes

### Communication – NetworkMgr

 Theory:


- In a real QKD system, Alice and Bob must exchange classical information during postprocessing (e.g., basis info, parity bits, syndromes).
- This is done using classical channels such as TCP/IP, which must be reliable but not secure (security is ensured by quantum rules).

 Code: NetworkMgr.h

- Class: NetworkMgr
- Functionality:
  - Handles socket creation, data transmission, and connection setup.
  - sendData() and receiveData() methods support sending arbitrary byte arrays.
- Roles:
  - Alice acts as server → binds to a port using bindSocket()
  - Bob acts as client → connects using connectSocket()
- Used to exchange:
  - LDPC parity data
  - Control messages (e.g., "start", "sync")
  - Toeplitz matrices (optional in some setups)

—

### Thread Management – ThreadMgr

 Theory:


- Each QKD postprocessing stage can run in parallel for better performance.
- Threading allows efficient processing without blocking.

 Code: ThreadMgr.h

- Class: ThreadMgr
- Launches 3 threads, each responsible for a postprocessing step:
  - a. Thread 0: Sifting (delete invalid packets from skQueue)
  - b. Thread 1: Error Correction (uses LDPC)
  - c. Thread 2: Privacy Amplification (Toeplitz matrix)
- Data transfer between threads is done using queues (KeyQueue).
- Threads communicate using threadCom struct, which includes flags, queues, and parameters.
- No need for mutex due to the use of separate in/out queues per thread.

—

### Central Controller – QkdMgr

 Theory:


- A controller is needed to initialize, configure, and coordinate all QKD components (delays, queues, threads, network).

 Code: QkdMgr.h

- Class: QkdMgr
- Responsibilities:
  - setDelays(): Configures FPGA delays (real hardware use).
  - setActAsAlice(): Assigns role.
  - setNetworkData(): IP and port configuration.
  - setPAKeyLength(), setLDPCMatFiles(): Configures security parameters.
  - start(), stop(), reset(): Starts or stops the QKD engine.
- Uses:
  - ThreadMgr: for postprocessing flow.
  - NetworkMgr: for classical communication.
  - DevMgr: for FPGA device access (not simulated here).

—

### Queue Architecture – KeyQueue

 Theory:

- Each stage in the QKD pipeline needs to pass data (keys) to the next stage.
- To support asynchronous and parallel operation, FIFO queues are used.

 Code: KeyQueue.h

- Classes:
  - KeyQueue: the FIFO container.
  - KeyQueuePacket: stores key block + metadata.
- KeyQueuePacket holds:
  - bool\* data: raw or processed key bits.
  - int length(): size of data.
  - float QBER: error rate of this packet.
  - bool valid: marks if packet passed checks.
- Pipeline:
  - skQueue (raw sifted key)  
→ validQueue (after sifting)  
→ ecQueue (after error correction)  
→ finalQueue (after privacy amplification)

—

 Summary :

"Each component of the system – communication, threading, control, and queuing – is modular and implemented using clean C++ classes. They work together to simulate the full classical postprocessing pipeline of QKD."