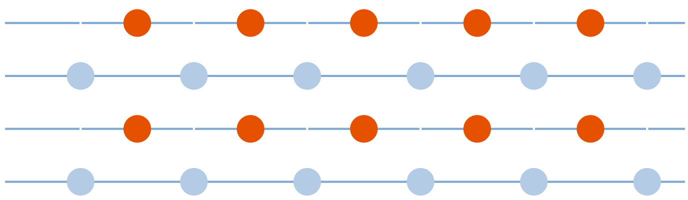

A Note to the Reader

This report is written for someone who has *zero* background in quantum physics, cryptography, or networking. By the time you finish reading, you will understand:

- What quantum mechanics is, with everyday analogies
- How the BB84 protocol works, step by step
- Every component of QSTCS and how they connect
- The actual Python code that makes it work
- Why this matters for future secure communication

No maths degree or coding experience needed.

Think of this as a **digital vault that uses the laws of physics as its lock**. Traditional locks can be broken. Quantum locks — the very act of picking them *permanently breaks them*, leaving evidence. That is the core idea.

Contents

Chapter 1

The Big Picture — What Are We Building?

1.1 The Problem: Why Normal Encryption Is Not Enough

Every time you send a message or do online banking, your data is encrypted using **mathematical problems** that are hard to solve — like factoring a huge number.

Concept: The Padlock Analogy

You put a letter in a locked box. Only your friend with the key can open it. Classical encryption works the same way — the lock is a maths problem, the key is its solution. Security assumption: *it would take millions of years to crack without the key.*

1.1.1 The Quantum Threat

Quantum computers can try many solutions at once (via **superposition**). In 1994, Peter Shor proved a powerful quantum computer could crack today's encryption in *hours*. Adversaries could capture data today and decrypt it later once quantum computers are ready — called **"Harvest Now, Decrypt Later."**

The timeline for cryptographically relevant quantum computers is estimated at 5–15 years. Sensitive data that must stay secret for decades is already at risk.

1.1.2 Our Solution

Instead of harder maths (a losing arms race), we use **the laws of physics**:

1. **BB84 QKD**: keys generated from quantum mechanics
2. **AES-256-GCM**: military-grade message encryption

3. **QBER monitoring**: automatic eavesdropper detection
4. **Router enforcement**: physical network block on attack

1.2 Project Overview

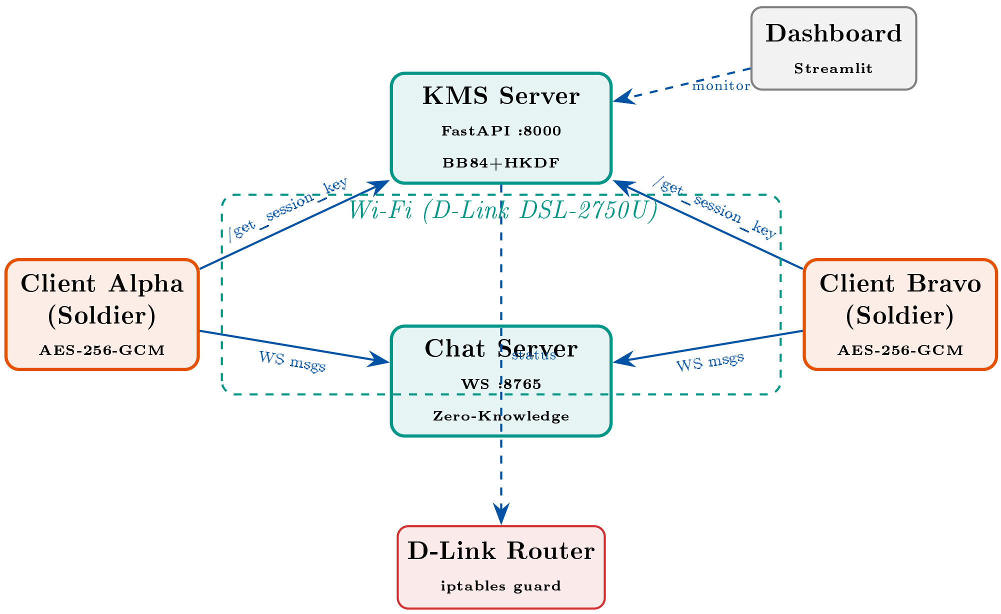

Figure 1.1: High-level architecture of QSTCS v3

Component	File	Purpose
BB84 Simulator	`bb84_simulator.py`	Generates keys using quantum principles, detects eavesdropping
KMS Server	`kms_server.py`	Key safe — manages and issues AES keys
Chat Server	`chat_server.py`	Relay: forwards encrypted messages, never decrypts
Chat Client	`client_app.py`	Soldier app — encrypts/decrypts messages
Router Guard	`router_guard.sh`	Physically blocks network if attack detected
Dashboard	`dashboard_ui.py`	Real-time SOC monitoring

Chapter 2

Quantum Mechanics Crash Course

Three concepts. Each explained with a simple analogy, then the technical detail. No physics degree needed.

2.1 Concept 1: Superposition — Two States at Once

In everyday life, a coin is heads or tails. A switch is on or off. Quantum particles do not follow this rule.

Concept: The Spinning Coin

While a coin spins, it is neither heads nor tails — it is in a **superposition** of both. It only “chooses” when it lands (when observed). A photon works the same way: it exists in superposition of polarization states until measured.

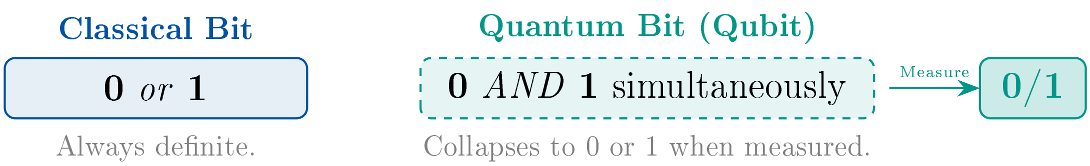

Figure 2.1: Classical bit vs. quantum qubit

2.2 Concept 2: Observer Effect — Measurement Changes Reality

The act of observing a quantum particle changes its state permanently and irreversibly.

Concept: The Magic Camera

Imagine a creature that exists in an undefined state until observed. The act of looking forces it to take one specific form. A quantum photon behaves the same way — when an eavesdropper intercepts and measures it, the photon collapses and the disturbance is detectable.

An eavesdropper **cannot** listen to a quantum channel without leaving detectable traces. This is the fundamental security guarantee.

2.3 Concept 3: No-Cloning Theorem

It is **physically impossible** to copy an unknown quantum state.

A classical attacker can copy encrypted data and crack it later. A quantum attacker **cannot copy** the quantum signal. They must measure it (disturbing it), which is detectable. There is no quantum photocopier.

2.3.1 Photon Polarization

BB84 encodes information in photon polarization across two measurement bases:

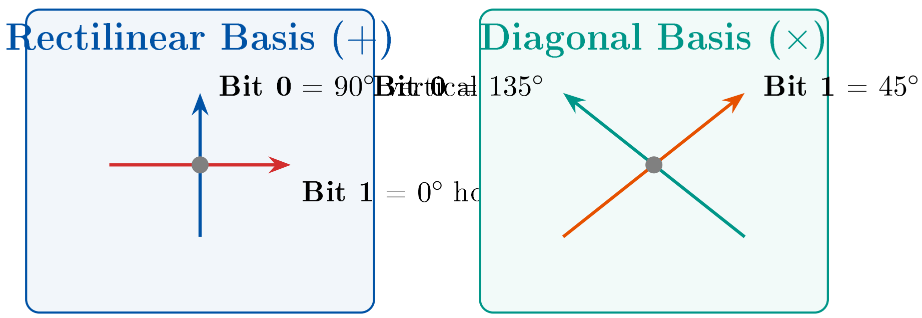

Figure 2.2: Two polarization bases in BB84

Chapter 3

The BB84 Protocol

3.1 Background

BB84 was invented in 1984 by Charles Bennett and Gilles Brassard. It allows two parties to share a secret key over an insecure channel, with security guaranteed by physics not maths. In our system, **Alice** = KMS server, **Bob** = client.

3.2 Step-by-Step

3.2.1 Step 1 — Alice Generates Random Bits and Bases

Position	1	2	3	4	5	6	7	8	...
Alice bits	0	1	1	0	0	1	0	1	...
Alice bases	$+$	$\times$	$+$	$\times$	$+$	$\times$	$+$	$\times$	...
Photon	$\uparrow$	$\searrow$	$\rightarrow$	$\nearrow$	$\uparrow$	$\searrow$	$\rightarrow$	$\searrow$	...

3.2.2 Step 2 — Bob Measures with Random Bases

Position	1	2	3	4	5	6	7	8	...
Bob bases	$+$	$+$	$+$	$\times$	$\times$	$\times$	$+$	$\times$	...
Bob result	0	?	1	0	?	1	0	1	...
Correct?	✓	✗	✓	✓	✗	✓	✓	✓	...

3.2.3 Step 3 — Basis Reconciliation

Alice announces her bases publicly. They keep only bits where both used the *same* basis. About 50% survive. These form the **sifted key**.

3.2.4 Step 4 — QBER Check

They sacrifice 20% of sifted bits to measure the Quantum Bit Error Rate.

Concept: Why Does QBER Reveal Eavesdroppers?

Without Eve: same-basis measurements always agree. QBER $\approx$ 0%.

With Eve: she must measure each photon (observer effect). Wrong-basis guesses (50% of the time) disturb photons. Bob then gets wrong results. This creates **≈25% errors** — detectable!

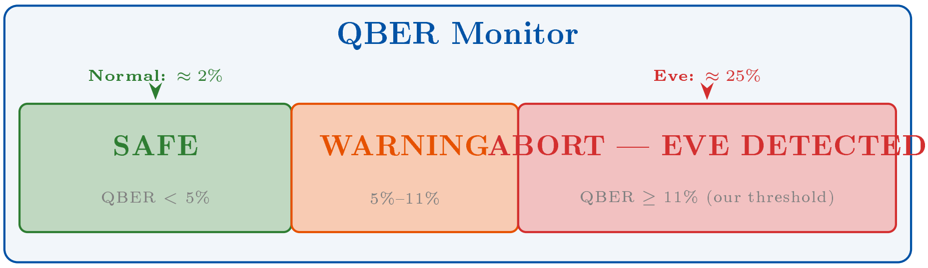

Figure 3.1: QBER threshold monitor

3.2.5 Step 5 — HKDF Key Derivation

Alice and Bob apply HKDF-SHA256 to compress the sifted bits into exactly 256 bits, eliminate any partial information Eve may have, and apply session-specific separation.

3.3 Complete BB84 Flowchart

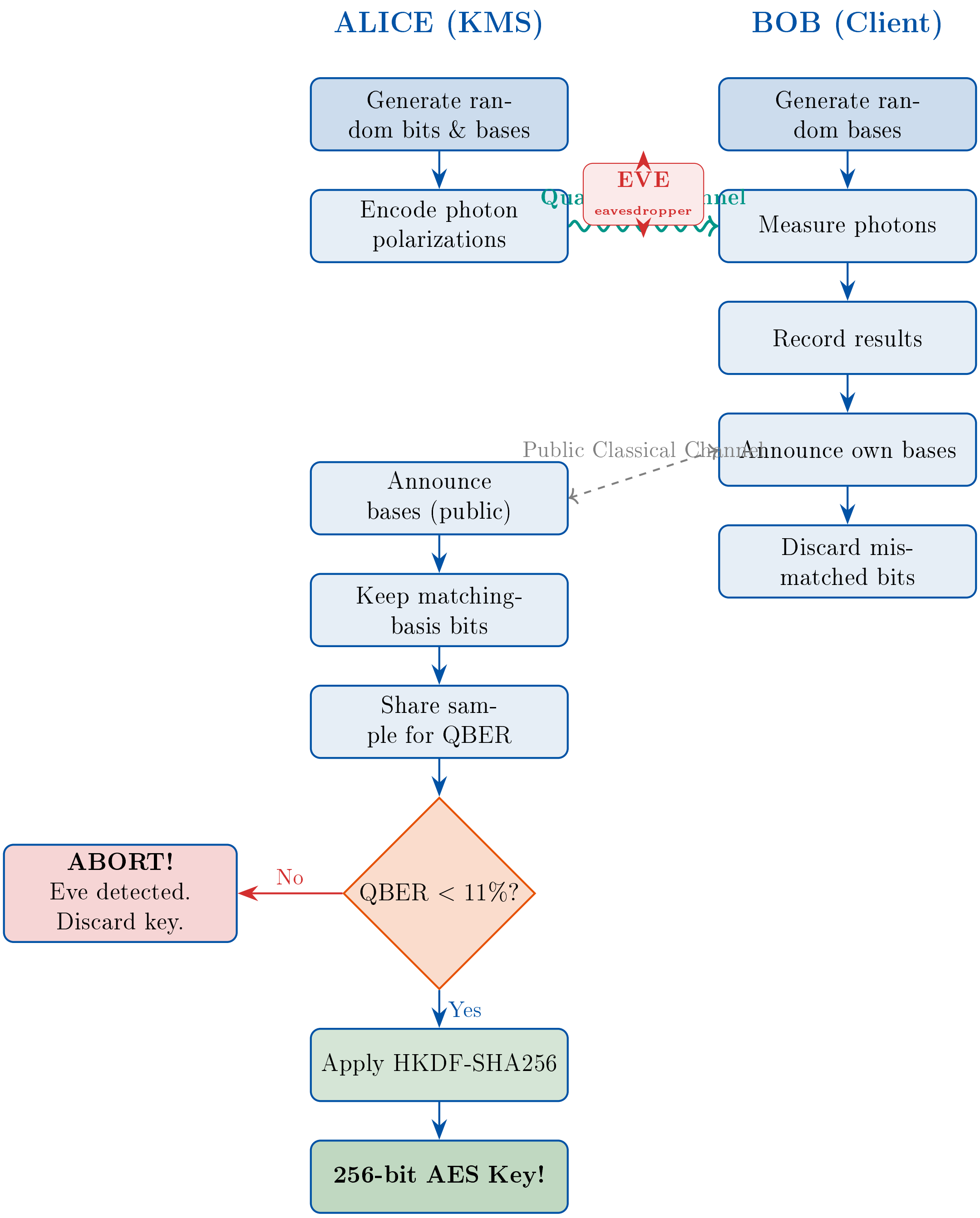

Figure 3.2: Complete BB84 protocol flowchart

Chapter 4

System Architecture

4.1 Component Diagram

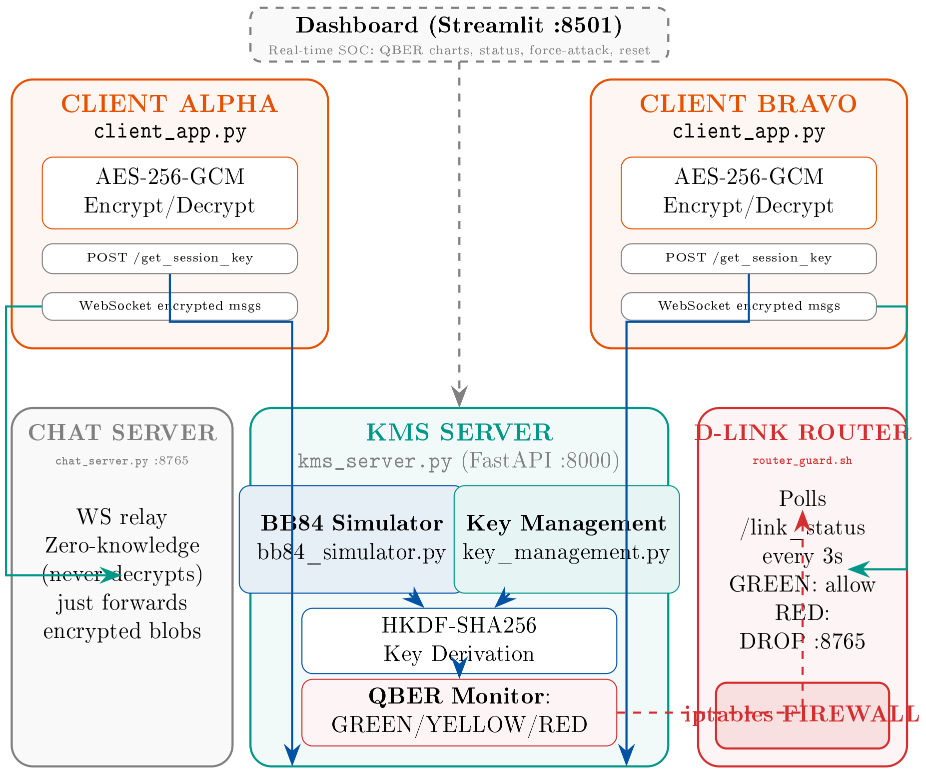

Figure 4.1: Detailed system architecture

4.2 Message Flow Sequence

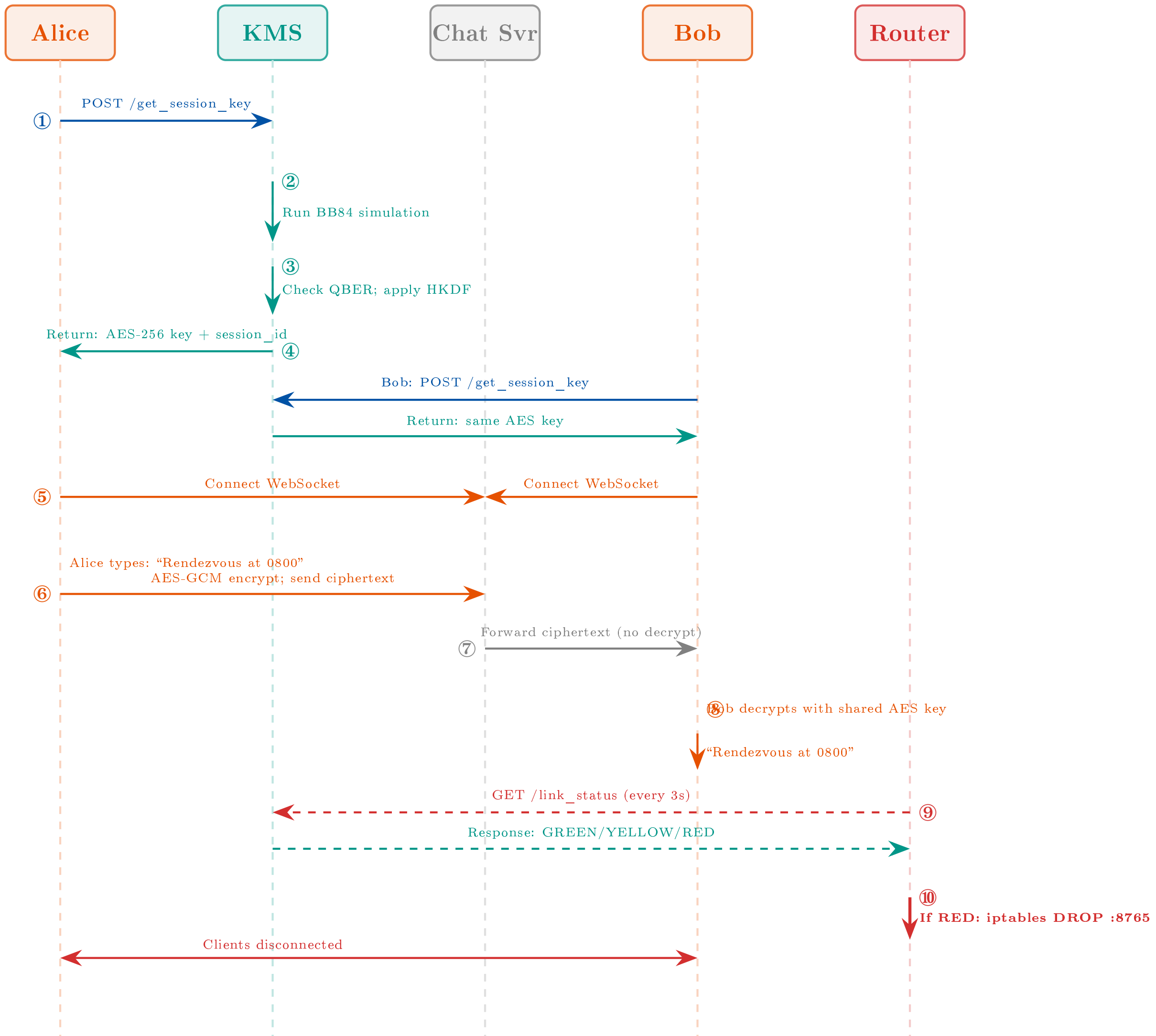

Figure 4.2: Sequence diagram: from key request to message delivery and threat response

Chapter 5

The Code — Every File Explained

5.1 bb84_simulator.py — The Quantum Engine

```python
import random

class BB84Simulator:
    """
    Simulates BB84 Quantum Key Distribution.
    Alice = KMS Server,  Bob = Client,  Eve = optional attacker
    """
    def __init__(self, n_bits=256):
        self.n_bits = n_bits

    def generate_key(self, eve_present=False):
        # STEP 1: Alice's random bits and bases
        alice_bits  = [random.randint(0,1) for _ in range(self.n_bits)]
        alice_bases = [random.randint(0,1) for _ in range(self.n_bits)]
        photons     = list(zip(alice_bits, alice_bases))

        # STEP 2: Optional Eve intercepts
        if eve_present:
            eve_bases = [random.randint(0,1) for _ in range(self.n_bits)]
            photons   = self._eve_intercept(photons, eve_bases)

        # STEP 3: Bob measures with random bases
        bob_bases = [random.randint(0,1) for _ in range(self.n_bits)]
        bob_bits  = self._measure_photons(photons, bob_bases)

```

```python
        # STEP 4: Basis sifting -- keep matching-basis positions only
        sifted_alice, sifted_bob = [], []
        for i in range(self.n_bits):
            if alice_bases[i] == bob_bases[i]:
                sifted_alice.append(alice_bits[i])
                sifted_bob.append(bob_bits[i])

        # STEP 5: QBER calculation on 20% sample
        n      = max(10, len(sifted_alice) // 5)
        sample = random.sample(range(len(sifted_alice)), n)
        errors = sum(sifted_alice[i] != sifted_bob[i] for i in sample)
        qber   = errors / n

        final_key = [sifted_alice[i]
                     for i in range(len(sifted_alice)) if i not in sample]
        return {'sifted_key': final_key, 'qber': qber,
                'key_length': len(final_key), 'eve_detected': qber > 0.11}

    def _measure_photons(self, photons, bob_bases):
        results = []
        for (alice_bit, alice_basis), bob_basis in zip(photons, bob_bases):
            if alice_basis == bob_basis:
                results.append(alice_bit)      # correct basis = perfect
            else:
                results.append(random.randint(0, 1))  # wrong = random
        return results

    def _eve_intercept(self, photons, eve_bases):
        """Wrong-basis interceptions disturb photon -> elevated QBER."""
        out = []
        for (alice_bit, alice_basis), eve_basis in zip(photons, eve_bases):
            if alice_basis == eve_basis:
                out.append((alice_bit, alice_basis))
            else:
                out.append((random.randint(0,1), eve_basis))  # disturbed!
```

```python
        return out
```

Listing 5.1: BB84 Simulator — Core Logic

Concept: Why Eve Creates ≈25% QBER

Eve guesses right 50% of the time. When wrong (50%), she re-sends a disturbed photon. Of those, Bob gets the wrong answer 50% of the time. Result: $0.5 \times 0.5 =$ 25% errors — well above our 11% abort threshold.

5.2 kms/key_management_service.py

```python
import os, hashlib
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend

class KeyManagementService:
    def __init__(self):
        self.sessions, self.qber_history = {}, []
        self.link_status, self.attack_active = 'GREEN', False

    def get_session_key(self, client_id, peer_id, eve_present=False):
        from quantum_engine.bb84_simulator import BB84Simulator
        result = BB84Simulator(n_bits=512).generate_key(eve_present)
        qber   = result['qber']
        self._update_link_status(qber)
        if self.link_status == 'RED':
            raise SecurityException(f"Compromised! QBER={qber:.1%}")
        raw     = self._bits_to_bytes(result['sifted_key'])
        aes_key = self._hkdf_derive(raw, client_id, peer_id)
        sid     = os.urandom(16).hex()
        self.sessions[sid] = {'key': aes_key, 'qber': qber}
        return {'session_id': sid, 'aes_key': aes_key.hex(),
                'qber': qber, 'link_status': self.link_status}

    def _hkdf_derive(self, raw_key, client_id, peer_id):
        """HKDF-SHA256: variable-length quantum bits -> exactly 32 bytes."""
        salt = hashlib.sha256(f"qstcs-{client_id}-{peer_id}".encode()).digest()
```

```python
        info = f"qstcs-session-{client_id}-{peer_id}".encode()
        return HKDF(algorithm=hashes.SHA256(), length=32,
                    salt=salt, info=info,
                    backend=default_backend()).derive(raw_key)

    def _update_link_status(self, qber):
        self.qber_history.append(qber)
        avg = sum(self.qber_history[-5:]) / len(self.qber_history
           [-5:])
        self.link_status = ('RED' if avg>=0.11 else 'YELLOW' if avg
           >=0.05 else 'GREEN')

    def _bits_to_bytes(self, bits):
        while len(bits) % 8: bits.append(0)
        return bytes(int(''.join(str(b) for b in bits[i:i+8]),2)
                     for i in range(0,len(bits),8))

class SecurityException(Exception): pass
```

Listing 5.2: Key Management Service

Concept: What is HKDF?

HKDF is a "key refinery." Raw quantum bits might not be exactly 256 bits and could carry subtle patterns. HKDF runs them through SHA-256 with a salt (**extract**) then stretches/compresses to exactly 32 bytes (**expand**). Perfect key out, regardless of what went in.

5.3 kms_server.py — REST API

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from kms.key_management_service import KeyManagementService,
   SecurityException

app = FastAPI(title="QSTCS Key Management Server")
kms = KeyManagementService()

class KeyRequest(BaseModel):
    client_id: str
    peer_id: str

@app.post("/get_session_key")
async def get_session_key(request: KeyRequest):
```

```python
    try:
        return kms.get_session_key(request.client_id, request.peer_id,
                                   kms.attack_active)
    except SecurityException as e:
        raise HTTPException(status_code=503, detail=str(e))

@app.get("/link_status")
async def link_status():
    """Router polls this every 3 seconds. Returns GREEN/YELLOW/RED."""
    return {"status": kms.link_status,
            "qber":   kms.qber_history[-1] if kms.qber_history else 0,
            "history": kms.qber_history[-20:]}

@app.post("/force_attack")
async def force_attack():
    """Demo: trigger Eve simulation. QBER spikes to ~25%."""
    kms.attack_active = True
    return {"message": "Eve attack activated"}

@app.post("/reset")
async def reset():
    kms.attack_active, kms.qber_history = False, []
    kms.link_status, kms.sessions = 'GREEN', {}
    return {"message": "KMS reset"}
```

Listing 5.3: KMS FastAPI Server

5.4 `chat_server.py` — Zero-Knowledge Relay

```python
import asyncio, json, websockets

connected_clients = {}   # name -> WebSocket

async def handle_client(websocket, path):
    """Relay handler. Never decrypts anything -- pure forwarding."""
    client_name = None
    try:
        data = json.loads(await websocket.recv())
        client_name = data['from']
        connected_clients[client_name] = websocket
```

```
        async for message in websocket:
            msg = json.loads(message)
            target = msg.get('to')
            if target and target in connected_clients:
                await connected_clients[target].send(message)
    except websockets.ConnectionClosed:
        pass
    finally:
        if client_name: connected_clients.pop(client_name, None)

asyncio.get_event_loop().run_until_complete(
    websockets.serve(handle_client, "0.0.0.0", 8765))
asyncio.get_event_loop().run_forever()
```

Listing 5.4: WebSocket Chat Server

5.5 client_app.py — The Soldier's App

```
import asyncio, json, os, requests, websockets
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

class QuantumSecureClient:
    def __init__(self, my_name, peer_name, kms_url, ws_url):
        self.my_name, self.peer_name = my_name, peer_name
        self.kms_url, self.ws_url = kms_url, ws_url
        self.aes_key = None

    def get_quantum_key(self):
        data = requests.post(f"{self.kms_url}/get_session_key",
            json={"client_id": self.my_name, "peer_id": self.
               peer_name}).json()
        self.aes_key = bytes.fromhex(data['aes_key'])
        self.session_id = data['session_id']
        print(f"[+] Key received. QBER:{data['qber']:.1%} | {data['
           link_status']}")

    def encrypt_message(self, plaintext):
        """AES-256-GCM: confidentiality + authentication in one
           step."""
        aesgcm = AESGCM(self.aes_key)
        nonce  = os.urandom(12)          # unique per message
        ct_tag = aesgcm.encrypt(nonce, plaintext.encode(), None)
        return {"from": self.my_name, "to": self.peer_name,
```

```
                "ciphertext": ct_tag[:-16].hex(), "nonce": nonce.hex(),
                "tag": ct_tag[-16:].hex(), "session_id": self.session_id}

    def decrypt_message(self, msg):
        """Raises InvalidTag if message tampered -- rejected automatically."""
        aesgcm = AESGCM(self.aes_key)
        nonce  = bytes.fromhex(msg['nonce'])
        ct_tag = bytes.fromhex(msg['ciphertext']) + bytes.fromhex(msg['tag'])
        return aesgcm.decrypt(nonce, ct_tag, None).decode()

    async def chat_session(self):
        self.get_quantum_key()
        async with websockets.connect(self.ws_url) as ws:
            await ws.send(json.dumps({"from": self.my_name}))
            await asyncio.gather(self._sender(ws), self._receiver(ws))

    async def _sender(self, ws):
        loop = asyncio.get_event_loop()
        while True:
            text = await loop.run_in_executor(None, input, f"{self.my_name}> ")
            if text.lower() == '/quit': break
            await ws.send(json.dumps(self.encrypt_message(text)))

    async def _receiver(self, ws):
        async for raw in ws:
            data = json.loads(raw)
            if data.get('from') == self.peer_name:
                try:    print(f"\n{self.peer_name}> {self.decrypt_message(data)}")
                except: print("[!] Verification failed -- possible tampering!")
```

Listing 5.5: Chat Client with AES-256-GCM

5.5.1 AES-256-GCM Encryption Flow

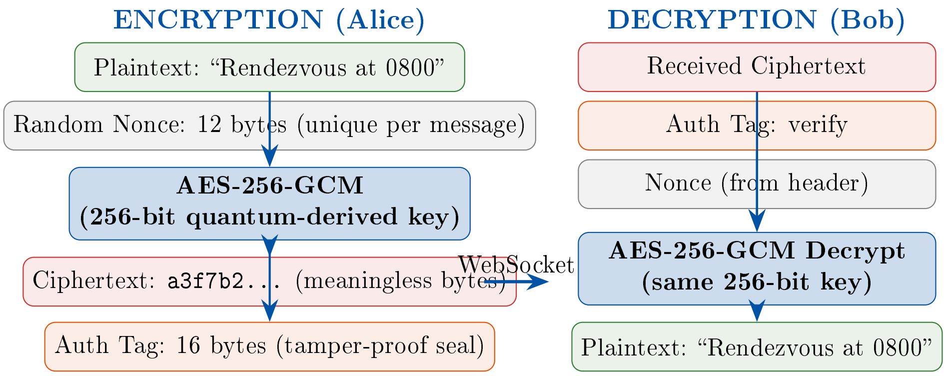

Figure 5.1: AES-256-GCM encryption and decryption flow

5.6 `router_guard.sh` — Network Enforcement

Listing 5.6: Router Guard (D-Link / OpenWrt)

```
#!/bin/sh
KMS_HOST="192.168.1.100"
CHAT_PORT="8765"
CURRENT_STATUS="GREEN"

allow_traffic() { iptables -D FORWARD -p tcp --dport $CHAT_PORT -j DROP 2>/dev/null; }
block_traffic()  { iptables -I FORWARD -p tcp --dport $CHAT_PORT -j DROP; }

while true; do
    STATUS=$(wget -qO- "http://$KMS_HOST:8000/link_status" 2>/dev/null |
             grep -o '"status":"[^"]*"' | cut -d'"' -f4)
    if   [ "$STATUS" = "RED"   ] && [ "$CURRENT_STATUS" != "RED" ]; then
        block_traffic;  CURRENT_STATUS="RED"
    elif [ "$STATUS" = "GREEN" ] && [ "$CURRENT_STATUS"  = "RED" ]; then
        allow_traffic;  CURRENT_STATUS="GREEN"
    fi
    sleep 3
done
```

Chapter 6

Security Analysis

6.1 Attack Protection

Attack	**Classical Defence**	**QSTCS Defence**
Passive eavesdropping	Hard maths	Physics — impossible without detection
Man-in-the-middle	Certificate authorities	QBER spike + router block
Message tampering	HMAC/signatures	AES-GCM authentication tag
Replay attacks	Timestamps/sequence nos.	Unique nonce per message
Future quantum computer	Broken by Shor's algorithm	Unaffected (physics-based)
Compromised relay	Single point of failure	Chat server is zero-knowledge

6.2 QBER Decision Flowchart

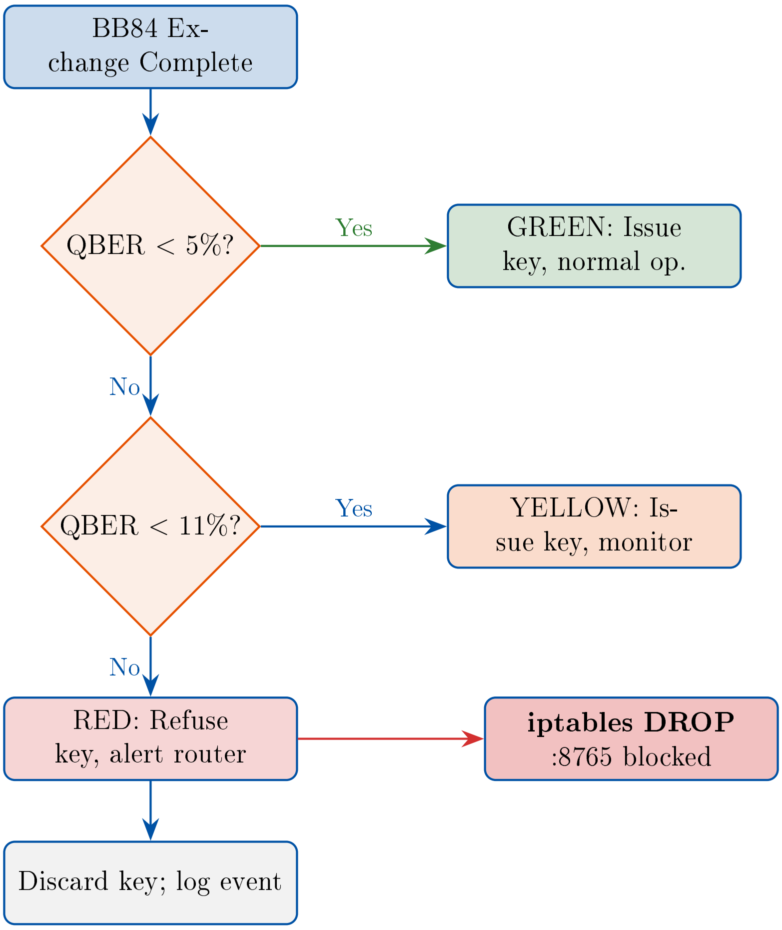

Figure 6.1: QBER decision flowchart

1. **Classical simulation, not real quantum hardware.** Real QKD requires actual photon hardware. The simulation correctly models all protocol logic.
2. **AES-256-GCM is real.** Production-grade cryptographic library — genuine security.
3. **Router enforcement is real.** iptables rules run on a real D-Link router.
4. **Eve is simulated.** The "force_attack" button triggers elevated QBER, mimicking what real hardware detects automatically from physics.

Chapter 7

Deployment Guide

7.1 Installation

Listing 7.1: Installation

```
git clone https://github.com/Sansyuh06/Comms.git && cd Comms
pip install -r requirements.txt
# Installs: fastapi, uvicorn, websockets, cryptography,
#           streamlit, plotly, requests, numpy
```

7.2 Local Demo — 5 Terminals

Listing 7.2: Local Demo

```
# Terminal 1
python kms_server.py          # KMS: http://localhost:8000
# Terminal 2
python chat_server.py         # WS relay: ws://localhost:8765
# Terminal 3
python client_app.py          # Soldier_Alpha / http://localhost:8000
    / ws://localhost:8765
# Terminal 4
python client_app.py          # Soldier_Bravo / http://localhost:8000
    / ws://localhost:8765
# Terminal 5
streamlit run dashboard/dashboard_ui.py    # http://localhost:8501
```

7.3 Live Demo Steps

1. **Normal:** Both clients chat. Dashboard shows QBER $\approx 2\%$, **GREEN**.
2. **Attack:** Click "Force Eve Attack." QBER spikes to $\approx 25\%$. Status $\rightarrow$ **RED**.

3. **Block:** Router executes `iptables DROP :8765` within 3 seconds. Clients disconnect.
4. **Recovery:** Click "Reset." QBER returns to $\approx 2\%$. Router allows traffic. Reconnect.

Chapter 8

Summary

What We Built

1. **BB84 QKD** — keys secured by physics, not maths
2. **QBER monitoring** — automatic eavesdropper detection
3. **HKDF-SHA256** — perfect 256-bit key derivation
4. **AES-256-GCM** — military-grade encryption with tamper detection
5. **Zero-knowledge relay** — server that cannot betray message contents
6. **Router enforcement** — physical network-level security response
7. **Real-time dashboard** — SOC monitoring with attack simulation

8.1 The Five Defence Layers

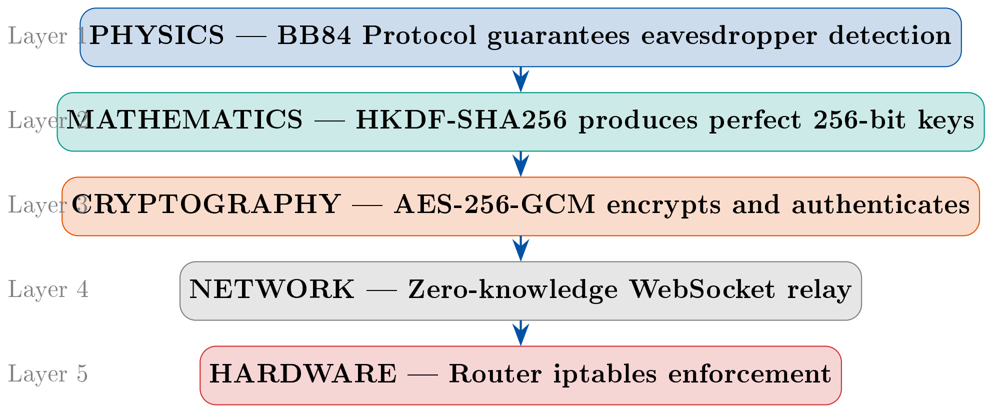

Figure 8.1: The five defence layers of QSTCS

8.2 Glossary

Term	**Definition**
BB84	First quantum key distribution protocol, invented 1984 by Bennett and Brassard
QBER	Quantum Bit Error Rate — fraction of sifted bits where Alice and Bob disagree; high QBER = eavesdropper
Qubit	Quantum bit — superposition of 0 and 1 until measured
Superposition	Quantum particle in multiple states simultaneously until observed
Observer Effect	Unavoidable disturbance caused by measuring a quantum state
No-Cloning Theorem	Physical law preventing copying of an unknown quantum state
Sifting	Discarding quantum key bits where Alice and Bob used different bases
HKDF	Hash-based Key Derivation Function — converts raw quantum bits into strong 256-bit key
AES-256-GCM	Advanced Encryption Standard, 256-bit, Galois/-Counter Mode — military-grade encryption
Nonce	Number Used Once — random value ensuring same plaintext encrypts differently each time
Auth Tag	16-byte value from AES-GCM that detects any tampering
KMS	Key Management Service — server running BB84 and distributing AES keys
WebSocket	Full-duplex TCP protocol for real-time bidirectional messaging
Zero-Knowledge Relay	Server that forwards data without ever reading it
iptables	Linux kernel firewall controlling network traffic at packet level
FastAPI	Python framework for building REST APIs
PQC	Post-Quantum Cryptography — math algorithms resistant to quantum attacks

Every layer of this system is a real technology:

- BB84 is the *same protocol* running in commercial QKD hardware today
- AES-256-GCM is the *same encryption* protecting your bank account
- HKDF-SHA256 is in the *TLS 1.3 standard* used by every modern website
- iptables is *real firewall technology* used in data centres worldwide

The gap between this simulation and real quantum hardware is engineering, not concept. This architecture describes systems that will protect communications in the quantum era.

That is what we built. And now — you understand all of it.