HAMMING QUASI CYCLIC

HQC presents a strong argument that its decryption failure rate is low enough to obtain chosen-ciphertext security.

Encryption scheme in Hamming metric, using Quasi-Cyclic Codes

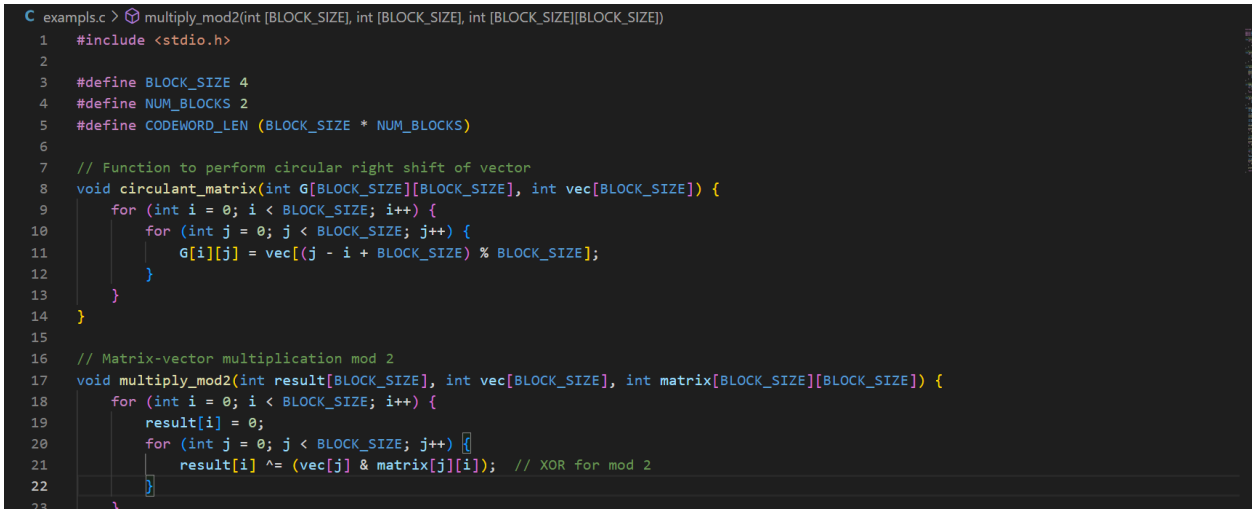
- **HAMMING METRIC :**
- The Hamming metric between two vectors is the number of positions at which the corresponding symbols are different.
- Helps define the minimum distance of a code: the smallest Hamming distance between any two distinct codewords.
- Distance determines the error-detection and correction capability.

- **QUASI CYCLIC CODES :**
- Linear block code where cyclic shift of a codeword by a fixed number of positions results in another code word.
- Representation - QC code of index ℓ and length $n=\ell \cdot p$
- Each codeword is composed of l blocks of p elements and cyclic shifting by one block gives another code word

- **USING QC CODE IN HAMMING METRIC :**
    1. Code construction
        - QC code over a finite field is described by its generator matrix, composed of circulant blocks.
        - These circulant matrices are easy to store and manipulate - they reduce complexity in encoding/decoding.
    2. Error correction in hamming metric :
        - Effective in correcting random bit flip flop error
        - minimum Hamming distance (d) of the code determines how many errors it can correct - t = [(d-1)/2).

- **APPLICATION IN CRYPTOGRAPHY:**
- In PQC, QC codes in the hamming metric are used in key encapsulation mechanisms.
- Benefits - efficient key generation and encryption/decryption , Hardness of decoding random linear codes in Hamming metric is NP-hard → strong security and Quasi-cyclic structure enables small public keys and ciphertexts, crucial for PQC.

- **SAMPLE CODE : quasi cyclic encoding and hamming metric**

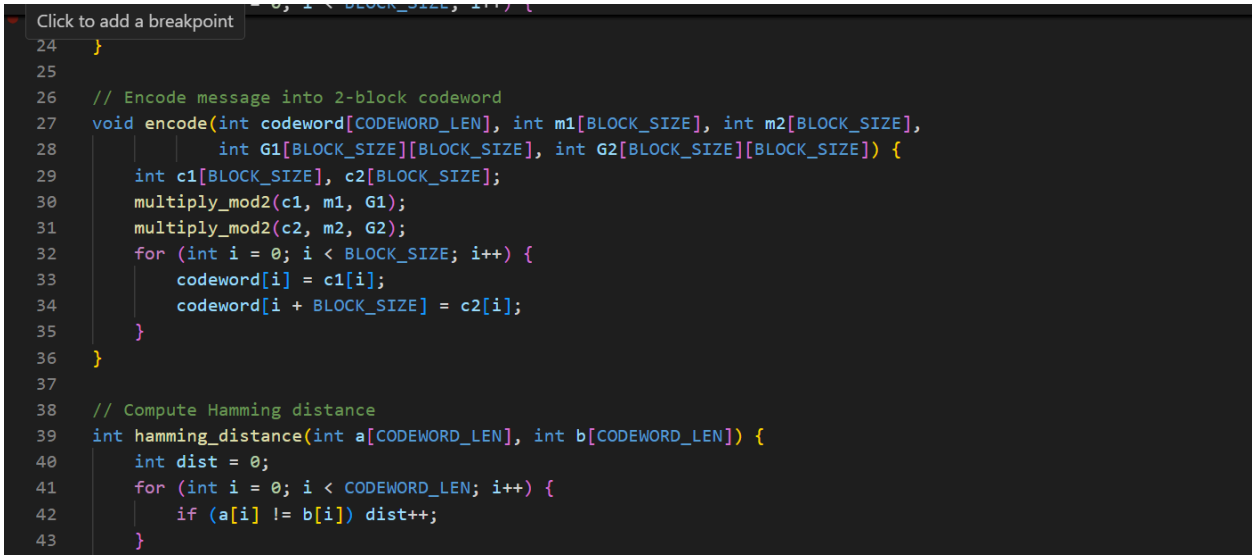


```c
examples.c > multiply_mod2(int [BLOCK_SIZE], int [BLOCK_SIZE], int [BLOCK_SIZE][BLOCK_SIZE])
#include <stdio.h>

#define BLOCK_SIZE 4
#define NUM_BLOCKS 2
#define CODEWORD_LEN (BLOCK_SIZE * NUM_BLOCKS)

// Function to perform circular right shift of vector
void circulant_matrix(int G[BLOCK_SIZE][BLOCK_SIZE], int vec[BLOCK_SIZE]) {
    for (int i = 0; i < BLOCK_SIZE; i++) {
        for (int j = 0; j < BLOCK_SIZE; j++) {
            G[i][j] = vec[(j - i + BLOCK_SIZE) % BLOCK_SIZE];
        }
    }
}

// Matrix-vector multiplication mod 2
void multiply_mod2(int result[BLOCK_SIZE], int vec[BLOCK_SIZE], int matrix[BLOCK_SIZE][BLOCK_SIZE]) {
    for (int i = 0; i < BLOCK_SIZE; i++) {
        result[i] = 0;
        for (int j = 0; j < BLOCK_SIZE; j++) {
            result[i] ^= (vec[j] & matrix[j][i]);  // XOR for mod 2
        }
    }
```



```c
Click to add a breakpoint
}

// Encode message into 2-block codeword
void encode(int codeword[CODEWORD_LEN], int m1[BLOCK_SIZE], int m2[BLOCK_SIZE],
            int G1[BLOCK_SIZE][BLOCK_SIZE], int G2[BLOCK_SIZE][BLOCK_SIZE]) {
    int c1[BLOCK_SIZE], c2[BLOCK_SIZE];
    multiply_mod2(c1, m1, G1);
    multiply_mod2(c2, m2, G2);
    for (int i = 0; i < BLOCK_SIZE; i++) {
        codeword[i] = c1[i];
        codeword[i + BLOCK_SIZE] = c2[i];
    }
}

// Compute Hamming distance
int hamming_distance(int a[CODEWORD_LEN], int b[CODEWORD_LEN]) {
    int dist = 0;
    for (int i = 0; i < CODEWORD_LEN; i++) {
        if (a[i] != b[i]) dist++;
    }
```

```c
    return dist;
}

// Perform 1-block cyclic shift of codeword
void qc_block_shift(int shifted[CODEWORD_LEN], int original[CODEWORD_LEN]) {
    for (int i = 0; i < BLOCK_SIZE; i++) {
        shifted[i] = original[i + BLOCK_SIZE];            // Move Block2 to Block1
        shifted[i + BLOCK_SIZE] = original[i];            // Move Block1 to Block2
    }
}

int main() {
    int g1[BLOCK_SIZE] = {1, 0, 1, 1};
    int g2[BLOCK_SIZE] = {0, 1, 0, 1};

    int G1[BLOCK_SIZE][BLOCK_SIZE], G2[BLOCK_SIZE][BLOCK_SIZE];
    circulant_matrix(G1, g1);
    circulant_matrix(G2, g2);

    int m1[BLOCK_SIZE] = {1, 0, 0, 1};
    int m2[BLOCK_SIZE] = {0, 1, 1, 0};
```

```c
    int codeword[CODEWORD_LEN];
    encode(codeword, m1, m2, G1, G2);

    printf("Encoded Codeword: ");
    for (int i = 0; i < CODEWORD_LEN; i++) {
        printf("%d ", codeword[i]);
    }

    // Perform QC shift
    int shifted_codeword[CODEWORD_LEN];
    qc_block_shift(shifted_codeword, codeword);

    printf("\nQC Block Shifted Codeword: ");
    for (int i = 0; i < CODEWORD_LEN; i++) {
        printf("%d ", shifted_codeword[i]);
    }

    // Introduce error
    int received[CODEWORD_LEN];
    for (int i = 0; i < CODEWORD_LEN; i++) {
        received[i] = codeword[i];
    }
```

```c
    }
    received[3] ^= 1;  // Flip 1 bit

    int dist = hamming_distance(codeword, received);
    printf("\nReceived Codeword (with error): ");
    for (int i = 0; i < CODEWORD_LEN; i++) {
        printf("%d ", received[i]);
    }

    printf("\nHamming Distance: %d\n", dist);

    return 0;
}
```

```
Loaded 'C:\WINDOWS\SysWOW64\msvcrt.dll'. Symbols loaded.
Encoded Codeword: 1 1 0 0 1 1 1 1
QC Block Shifted Codeword: 1 1 1 1 1 1 0 0
Received Codeword (with error): 1 1 0 1 1 1 1 1
Hamming Distance: 1
```

- **General Setup**
- Block size: 4 bits
- Number of blocks: 2
- Total codeword length = 4 × 2 = 8 bits
- We use 2 generator vectors to define the structure of the QC code.
- **Functions**

- circulant_matrix(...)

- Builds a circulant matrix from a generator vector.
- Each row is a right circular shift of the row above.
- Used to structure generator matrices G1 and G2.

- multiply_mod2(...)

- Multiplies a vector by a matrix mod 2 (binary arithmetic).
- Uses bitwise AND for multiplication and XOR for addition.
- Performs encoding for each message block.

- encode(...)

- Takes two message blocks (m1, m2) and generator matrices (G1, G2).
- Encodes each block using multiply_mod2.
- Concatenates encoded blocks to form the final codeword.

- hamming_distance(...)

- Computes the Hamming distance between two binary vectors.
- Loops through bits and counts how many positions differ.
- Useful for error detection.

- qc_block_shift(...)

- Implements a 1-block QC cyclic shift:
    - Moves Block 2 to Block 1's position.
    - Moves Block 1 to Block 2's position.
- Demonstrates the quasi-cyclic property of the code.

- **Main Function Logic (main())**

1. Define generator vectors g1 and g2 for two blocks.
2. Use circulant_matrix to generate matrices G1 and G2.
3. Define message blocks:
    - m1 = {1, 0, 0, 1}
    - m2 = {0, 1, 1, 0}
4. Encode the message using encode().
5. Print the original encoded codeword.
6. Apply a QC block shift and print the shifted version.
7. Simulate an error by flipping a bit in the original codeword.
8. Use hamming_distance() to calculate how far the error is from the original.
9. Print all three:
    - Original codeword
    - Shifted codeword (quasi-cyclic property)
    - Received codeword with error
    - Hamming distance

HQC SUBMISSION pdf AT NIST summary :

The document HQC_Submission.pdf" details the Hamming Quasi-Cyclic (HQC) Key Encapsulation Mechanism (KEM), a candidate for post-quantum cryptography standardization. As a formal submission, it provides a comprehensive overview of the scheme's design, security, and implementation.

1. Introduction and Overview

This section usually sets the stage by:

- Defining HQC: Explaining that HQC is a Key Encapsulation Mechanism (KEM) which is a type of public-key encryption scheme designed for key exchange. It's "Hybrid Quantum-Classical" because it aims to be secure against both classical and quantum computing attacks.
- Context: Placing HQC within the NIST Post-Quantum Cryptography Standardization process, indicating its purpose is to provide cryptographic security in an era where large-scale quantum computers could break current public-key algorithms (like RSA and ECC).
- Key Features: Highlighting its distinguishing characteristics, such as:
    - IND-CCA2 KEM: This is a strong security notion, meaning the scheme is secure against chosen-ciphertext attacks, which is crucial for practical key encapsulation.
    - Small Public Key Size: An important practical advantage for storage and transmission efficiency.
    - Efficient Implementations: Emphasizing that the scheme is not just theoretically secure but also practical for real-world use, often achieved through optimized code (e.g., using AVX2 instructions).

2. Mathematical Background and Building Blocks

This is the foundational section explaining the theoretical underpinnings. HQC likely relies on problems from coding theory, specifically on the hardness of decoding certain types of codes.

- Error-Correcting Codes:
    - Reed-Muller Codes: These are a family of linear error-correcting codes known for their algebraic structure and efficient decoding algorithms (though decoding can be complex for general cases, often simplified for specific parameters). The document notes their decoding uses AVX2 instructions for speed.
    - Reed-Solomon Codes: Another powerful class of error-correcting codes, often used for burst error correction. They involve operations in finite fields.
    - Concatenated Codes: The README mentions HQC uses concatenated codes, which means combining two or more codes (like Reed-Solomon and

Reed-Muller) to achieve better error correction capabilities than a single code alone, often with more efficient decoding.

- Finite Fields (Galois Fields - GF): Cryptography, especially coding theory-based schemes, relies heavily on arithmetic over finite fields. This section would describe the specific finite field (e.g., GF(2^m) for some m) used for computations.
- Polynomial Arithmetic: Operations like polynomial multiplication are fundamental. The document specifies an AVX2 implementation for multiplying polynomials (likely in GF(2)[x]).
- Additive Fast Fourier Transform (FFT): This is an efficient algorithm for polynomial multiplication and can be crucial for fast decoding of codes like Reed-Solomon.

3. Scheme Description (Core Cryptographic Operations)

This section formally defines the HQC-KEM, detailing the algorithms for its primary operations:

- Key Generation:
  - Input: Security parameters.
  - Output: A public key (PK) and a secret key (SK). The README mentions that to shorten keys, the public key is stored as (seed1, s) and the secret key as (seed2). This implies a "seed expander" is used to deterministically derive larger components of the key from these smaller seeds, improving key compactness.
- Encapsulation (PKE IND-CPA and KEM IND-CCA2):
  - HQC PKE IND-CPA: This is the underlying Public Key Encryption (PKE) scheme that provides chosen-plaintext attack security. A sender encrypts a message (often a random value) using the recipient's public key.
  - HQC KEM IND-CCA2: This is built on top of the PKE scheme, often using a transformation like the Fujisaki-Okamoto (FO) transform. This transform converts an IND-CPA secure PKE into an IND-CCA2 secure KEM.
    - Input: Recipient's Public Key.
    - Output: A ciphertext (CT) and a shared secret (SS). The README indicates the shared secret is the output of hashing a message m using SHAKE256.
    - The ciphertext is composed of vectors (u, v, d).
- Decapsulation:
  - Input: Ciphertext (CT) and the recipient's Secret Key (SK).
  - Output: The shared secret (SS) or an error if the ciphertext is invalid. The process involves using the secret key to "decode" the ciphertext and recover the ephemeral message that leads to the shared secret.
- Representations: The document would explain how mathematical objects (like vectors, polynomials) are represented in memory, distinguishing between their "mathematical

representations" and "bit strings," and how parsing functions facilitate conversion between them.

4. Parameter Sets

This section defines the specific numerical parameters for different security levels, typically aligned with NIST's categories (e.g., Category 1, 3, 5, corresponding to classical security equivalents of AES-128, AES-192, AES-256). For HQC, these are specified as HQC-128, HQC-192, and HQC-256. Each set defines the dimensions of vectors, sizes of fields, and error correction capabilities of the codes used.

5. Implementation Details

This part bridges the gap between theory and practice:

- Reference Implementation: A straightforward, clear, and usually unoptimized implementation intended for verifying the correctness of the scheme.
- Optimized Implementations: Focus on speed and efficiency. The README confirms the use of AVX2 instructions for vectorized operations (e.g., polynomial multiplication in gf2x.c, Reed-Muller decoding in reed_muller.c), which leverage modern CPU capabilities for significant speedups.
- Component Files: The README outlines how the scheme is modularized into different .c and .h files, each handling specific functionalities:
    - api.h, parameters.h, kem.c: Define and implement the overall KEM.
    - hqc.h, hqc.c: Define and implement the underlying PKE.
    - code.h, code.c, reed_solomon.h, reed_solomon.c, reed_muller.h, reed_muller.c, fft.h, fft.c: Implement the concatenated codes and their components.
    - gf.h, gf.c: Galois field manipulation.
    - gf2x.h, gf2x.c: Polynomial multiplication.
    - parsing.h, parsing.c: Key/ciphertext parsing.
    - shake_ds.h, shake_ds.c, domains.h: SHAKE256-based domain separation.
    - shake_prng.h, shake_prng.c: SHAKE256-based pseudo-random number generation.
    - fips202.h, fips202.c: SHA3 hash function (SHAKE is a variant of SHA3).
    - vector.o: General vector manipulation functions.
- Build Process: Explains how to compile the code (e.g., using make commands like make hqcX) and the resulting object files (.o) and executables (bin/hqcX).

6. Performance Benchmarks

This section provides quantitative data on the scheme's efficiency:

- Metrics: Execution times for key generation, encapsulation, and decapsulation operations (e.g., cycles per operation).
- Memory Footprint: How much RAM is used by the different components or during operations.
- Comparison: Often compares performance across different parameter sets and potentially against other post-quantum candidates.

7. Security Analysis

A critical part of any cryptographic submission. This section would provide:

- Formal Security Proofs: Mathematical proofs demonstrating that the scheme meets its claimed security levels (e.g., IND-CCA2).
- Hardness Assumptions: Clearly stating the computational problems upon which the scheme's security relies (e.g., decoding random linear codes, the syndrome decoding problem, or specific code-based problems). These problems are believed to be hard even for quantum computers.
- Side-Channel Analysis: Discussion of potential vulnerabilities to side-channel attacks and countermeasures.

8. Testing and Validation

- NIST KAT (Known Answer Test) Generation: Instructions for generating standard test vectors that can be used to verify the correctness of any HQC implementation. This ensures different implementations produce the same cryptographic outputs for specific inputs.
- Compliance: How the implementation adheres to the specified API and testing requirements for the NIST competition.

9. Documentation

How to generate and access the code documentation (e.g., using Doxygen).

10. Additional Information / References

Any other relevant details, such as an overview of the implementation structure, specific optimizations, and a list of all academic papers and resources cited in the document.