Secure Image Sharing Using Threshold Cryptography and Visual Secret Sharing

S.Azhagesh (23BCE1205)

Giridharen Goguladhevan (23BCE5043)

Rohith Ganesh Kanchi (23BCE5049)

S.D. Madhumitha (23BCE5058)

BCSE204L (Design and analysis of Algorithm Theory)

1.Aim:

The primary aim of this project is to design, implement, and evaluate a robust and efficient system for secure image sharing based on a threshold mechanism. The system should divide a secret image into multiple 'shares' such that the original image can only be reconstructed when a predetermined minimum number (threshold, k) of shares are combined. Possessing fewer than k shares should reveal no discernible information about the original image content. The system must prioritize security against unauthorized access and computational efficiency suitable for potentially large images.(SSS vs CRT).

2. Proposed Solution:

The proposed solution leverages Shamir's Secret Sharing (SSS) algorithm, applied independently to each pixel of a grayscale image. The core idea involves representing the intensity value of each pixel as a secret and generating polynomial shares based on this secret.

The evolution of the solution across the provided code variations is as follows:

1. **Basic SSS (Code 1):** Implements the fundamental (k, n) threshold SSS scheme. Each pixel value (modulo a prime P=257 for lossless 8-bit representation) is encoded as the constant term (y-intercept) of a random polynomial of degree k-1.

n points on this polynomial (at x=1, 2, ..., n) are calculated, forming the share values for that pixel across n shares. Reconstruction uses Lagrange interpolation with any k shares. Integrity is maintained using SHA-256 hashing of the original image and individual numerical shares.

2. **SSS with N+1 Share (Code 2):** Extends the basic scheme by creating an additional (n+1)-th share. This share is computed as the pixel-wise sum (modulo P) of the initial n shares. This variation, often cited in participant-increasing SSS schemes, allows reconstruction from any k shares out of the *total* n+1 shares. When the (n+1)-th share is included in the reconstruction set, a system of linear equations is solved (using modular arithmetic) instead of Lagrange interpolation. This version uses P=251, potentially introducing minor information loss for pixel values 251-255, matching a specific scheme potentially referenced.
3. **Parallelized SSS with N+1 Share (Code 3):** Addresses the computational efficiency objective by introducing parallelism using Python's concurrent.futures.ProcessPoolExecutor. Both the share creation (polynomial evaluation across all pixels) and the reconstruction (Lagrange/Linear Algebra across all pixels) phases are parallelized by dividing the image into row-based chunks processed by multiple worker processes. This significantly speeds up processing for larger images. It retains the n+1 share mechanism and P=251. Timings for parallel vs. sequential execution are measured.
4. **Encrypted & Parallelized SSS with N+1 Share (Code 4):** Enhances security, particularly for shares at rest. While SSS provides information-theoretic security against reconstruction with < k shares, the stored numerical shares could still be vulnerable if the storage medium is compromised. This version adds AES-GCM authenticated encryption to the *numerical* share data before saving it to disk. A strong key is derived from a user-provided password using PBKDF2HMAC. This ensures that even if an attacker obtains the share files, they cannot access the numerical data without the correct password. Hashing for integrity is performed on the *plaintext* numerical data *before* encryption. Parallel processing, the n+1 mechanism, and P=251 are retained.

3. Definition of the terms:

In the context of this project, 'P' refers to the **Prime Modulus** used in the Shamir's Secret Sharing algorithm. SSS operates within a finite mathematical field, specifically a Galois Field GF(P), where P is a prime number. All polynomial coefficients, evaluations, and reconstruction calculations (including modular inverses) are performed modulo P.

- **Choice of P:**
    - For grayscale images with pixel values ranging from 0 to 255, choosing a prime P > 255 is necessary for **lossless** reconstruction. P = 257 (used in Code 1) is the smallest prime greater than 255 and is suitable.
    - Codes 2, 3, and 4 use P = 251. This choice might be based on specific research papers or constraints. Using P = 251 means that original pixel values from 251 to 255 will be reduced modulo 251 during the sharing process (e.g., 251 becomes 0, 255 becomes 4). This introduces a minor, usually imperceptible, **loss** of information for the brightest pixels upon reconstruction. The system correctly handles this by hashing the *original image modulo P* for integrity checks.
- **Role:** The prime P defines the field size, ensuring that operations like addition, subtraction, multiplication, and division (via modular inverse) are well-defined and closed within the field, which is essential for the mathematical properties of SSS to hold.

4. Procedure:

The overall procedure implemented, particularly in the most advanced version (Code 4), involves the following steps:

1. **Initialization:**
    - Load the input image.
    - Prompt user for SSS parameters: initial number of shares (n), threshold (k).
    - Preprocess the image: Convert to grayscale, obtain NumPy array representation.
    - Define the prime modulus P (e.g., 251 or 257).
    - Calculate the SHA-256 hash of the original image data (pixel values taken modulo P) and store it.
    - Create the output directory if it doesn't exist.
2. **Share Generation (Encryption Phase):**
    - **(Parallel Execution - Code 3 & 4):** Divide the image pixel data into chunks (e.g., rows).
    - **(Parallel/Sequential):** For each pixel (r, c):
        - Determine the secret s = pixel_value(r, c) % P.

        - Generate a random polynomial poly(x) of degree k-1 such that poly(0) = s. Coefficients are chosen randomly within [0, P-1].
        - Calculate n share points for this pixel: share_value_i = poly(i) % P for x = i = 1, 2, ..., n.
        - Store these n values in the corresponding positions (r, c) of the n numerical share arrays.
    - **(Codes 2, 3, 4):** Calculate the (n+1)-th numerical share array by summing the first n numerical share arrays, pixel-wise, modulo P.
    - Combine all numerical shares (1 to n or 1 to n+1) into a list.
3. **Share Storage (Securing Shares):**
    - **(Code 4):** Prompt the user to set and confirm a password for encrypting the numerical shares. Derive an AES key using PBKDF2HMAC.
    - For each numerical share array (index i = 1 to n or n+1):
        - Calculate the SHA-256 hash of the **plaintext** numerical share array. Save this hash to share_{i}_hash.txt.
        - **(Code 4):** Encrypt the numerical share array bytes using AES-GCM with the derived key, nonce, and salt. Save the resulting bundle (salt + nonce + ciphertext) to share_{i}_numerical.enc.
        - **(Codes 1, 2, 3):** Save the plaintext numerical share array to share_{i}_numerical.npy.
        - Generate a visual representation (modulo 256, uint8) of the numerical share. Save this as share_{i}_visual.png.
4. **Reconstruction Simulation (Decryption Phase):**
    - Prompt the user to enter the indices of k shares they wish to use for reconstruction.
    - **(Code 4):** Prompt the user for the decryption password if not already provided in the session.
    - For each selected share index i:
        - Load the corresponding hash from share_{i}_hash.txt.
        - **(Code 4):** Load the encrypted data from share_{i}_numerical.enc. Decrypt it using the provided password (deriving the key again). Reshape the resulting bytes into a NumPy array (shape inferred from the visual share PNG).
        - **(Codes 1, 2, 3):** Load the numerical share from share_{i}_numerical.npy.

        - Calculate the SHA-256 hash of the loaded (and potentially decrypted) numerical share.
        - **Integrity Check:** Compare the calculated hash with the loaded hash. If they don't match, flag the share as invalid/corrupt and abort reconstruction if needed.
    - If all k required shares are loaded, decrypted, and verified successfully:
        - **(Parallel Execution - Code 3 & 4):** Divide the reconstruction task into chunks.
        - **(Parallel/Sequential):** For each pixel (r, c):
            - Gather the k points (share_index, share_value_at_pixel) from the loaded shares.
            - Determine if the (n+1)-th share (if applicable) is among the chosen shares.
            - **Reconstruct Secret:**
                - If using standard SSS (Code 1) or the (n+1)-th share is *not* present (Codes 2, 3, 4): Use Lagrange interpolation on the k points to find the value at x=0 (the original secret s).
                - If the (n+1)-th share *is* present (Codes 2, 3, 4): Construct the k x k matrix and k x 1 vector for the linear system of equations. Solve the system M * A = Y modulo P to find the coefficient vector A. The secret s is the first element A[0].
            - Store the reconstructed secret s in the corresponding position (r, c) of the reconstructed image array.
    - Assemble the final reconstructed image array. Convert to uint8 (clipping if necessary, though unlikely if P > 255).
5. **Final Verification:**
    - Calculate the SHA-256 hash of the reconstructed image data (pixel values taken modulo P).
    - Compare the reconstructed hash with the originally stored hash of the source image. Report PASS or FAIL.
6. **Display:**

- Show the original image, selected visual shares (e.g., Share 1 and Share n+1), and the final reconstructed image using matplotlib. Include timing results if measured.

A. The sharing phase:

Suppose that the secret image S was already encrypted into n shadows. And S cannot be revealed without k or less shadow images. We present the proposed participants increasing method based on the input n original shadow images SC1SC2, ⋯ SCn generated from Shamir's (k, n) threshold scheme. The steps are described in Algorithm 1, whose diagrammatic design concept is shown in Fig. 1. From Fig. 1 we can see that the pixel SCn+1(i, j) is got by adding up the input k pixels.

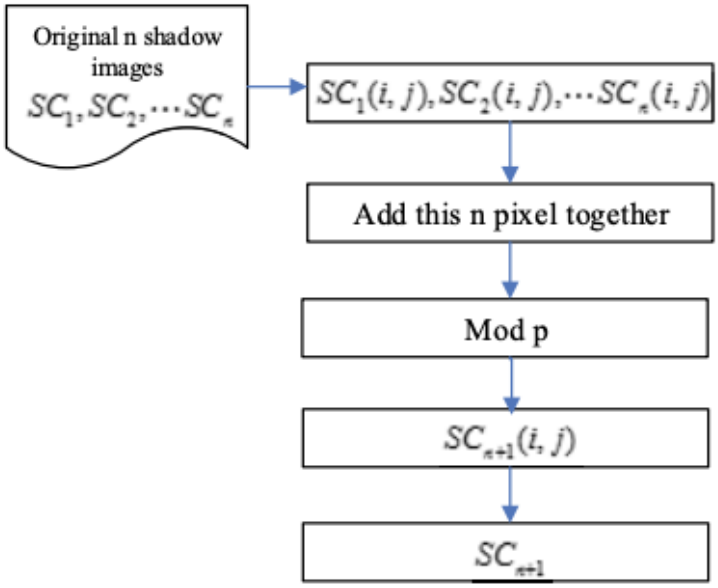


Figure 1. Design concept of the proposed method

The input k original secret shadow images SC1,SC2, ⋯ SCn are generated from Shamir's polynomial-based scheme. In step 2 of our Algorithm 1, we generate a new shadow pixel by adding together the n shadow images pixels at position (i, j). As a result, at position (i, j), (n + 1,SCn+1(i, j)) will construct the shadow image pixels pairs which may play the same role as SCx(i, j) for the recovery, where x = 1, 2, ⋯ , n. We note that: we select the prime number p be 251 which is the greatest prime number not larger than 255. The thing is, since the gray pixel value of an image range from 0 to 256, our scheme is not totally lossless. But, compared to exited VCS schemes, it is still a high resolution.

Algorithm 1. The proposed participant increasing method in sharing phase
Input: k original secret shadow images $SC_1, SC_2, \cdots SC_n$ **Output**: 1 new shadow image SC_{n+1}
Step 1: For each position$(i,j) \in \{(i,j)\|1 \leq i \leq M, 1 \leq j \leq N\}$, repeat Step 2. **Step 2:** Set $SC_{n+1}(i,j) = (SC_1(i,j) + SC_2(i,j) + \cdots + SC_n(i,j)) \bmod p$ **Step 3:** Output the new shadow image SC_{n+1}

B. The recovery phase:

In above sharing phase, when we generate the new shadow pixel, we actually generate the new shadow pixel by linear combination of polynomials. The new polynomial is SCn+1(i, j)=(f(1) + f(2) + ⋯ + f(k)) mod p = (a0 × k + a1 × (x1 + x2 + ⋯ + xk)+ ⋯ + ak × (xk−1 1 + xk−1 2 + ⋯ + xk−1 k)) mod p (3) Thus, we can reveal the secret image S with k or more shadows selected from the n + 1 shadow images. The recovery steps can be described in Algorithm 2. In steps 1−3

Algorithm 2. The proposed participants increasing method in recovery phase
Input: The $k(> k)$ shadow images which are randomly selected from $n+1$ shadow images $SC_1, SC_2, \cdots, SC_n, SC_{n+1}$ **Output**:The original secret image S
Step 1: Select $k(> k)$ shadows, and note down their lables x. **Step 2:** For each position$(i,j) \in \{(i,j)\|1 \leq i \leq M, 1 \leq j \leq N\}$, repeat Step 3-4. **Step 3:** According to the k or more lables, Eq(1) and Eq(3), construct the k or more corresponding polynomials. **Step 4:** Get the coefficient a_0 of $f(x)$ by solving polynomials, and set the pixel $S(i,j)$ as the value a_0. **Step 5:** Output the secret image S

of Algorithm 2, we construct the k or more polynomials. In detail, if the n + 1th shadow image SCn+1 is selected, we get the corresponding polynomial according to equation (3). And other polynomials are constructed in the form of equation (1). While Step 4 of Algorithm 2 aims to solve polynomials and get the secret pixel value a0. We note that: if the n + 1th new shadow is not selected, we can solve the equations by the Largrange's interpolation. And, if the n+1th new shadow is selected, some conversion calculation need to be performed. For example, we want to extend a (2, 2) threshold to a (2, 3) threshold scheme, if the shadow images (1,SC1) and (2,SC2) are selected, we can reveal the secret image by Largrange's interpolation directly. If (1,SC1) and (3,SC3) are selected, we subtracts SC3(i, j) from SC1(i, j), then we can reveal the secret image by Largrange's interpolation.

We have also implemented a basic visual cryptography, and also explored a new CRT(chinese remainder theorem) Ashmuth Bloom and tested how the entropy analysis for both the SSS and the CRT, the results have been provided below.

Maxflow Algorithm has been been implemented for the CRT.

5. Algorithms:

Algorithm 1: Basic SSS (Code 1)

Function ShareImage(imageData, k, n, P=257):

```
Initialize n empty share arrays (height, width)
originalHash = Hash(imageData % P)
Save originalHash
For each pixel (r, c) in imageData:
  secret = imageData[r, c] % P
  coeffs = GeneratePolynomial(degree=k-1, intercept=secret, P)
  For i from 1 to n:
    shareValue = EvaluatePolynomial(coeffs, x=i, P)
    shareArrays[i-1][r, c] = shareValue
For i from 1 to n:
  shareHash = Hash(shareArrays[i-1])
  Save shareArrays[i-1] as NumericalShare_i.npy
  Save shareHash as ShareHash_i.txt
  Save VisualShare(shareArrays[i-1]) as VisualShare_i.png
Return success
```

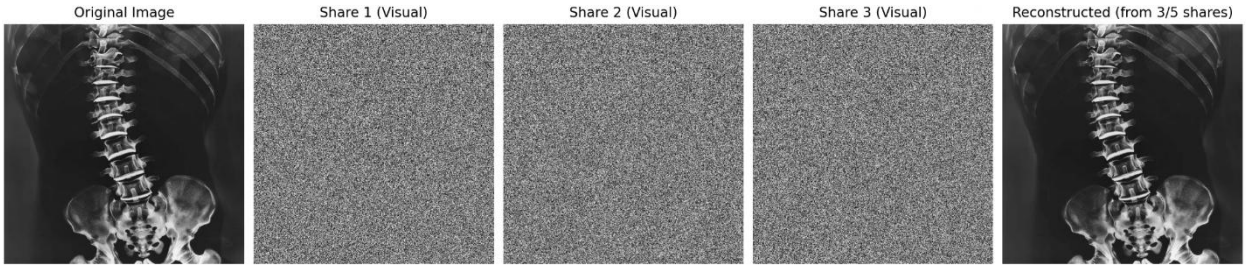
Function ReconstructImage(shareIndices, k, P=257):

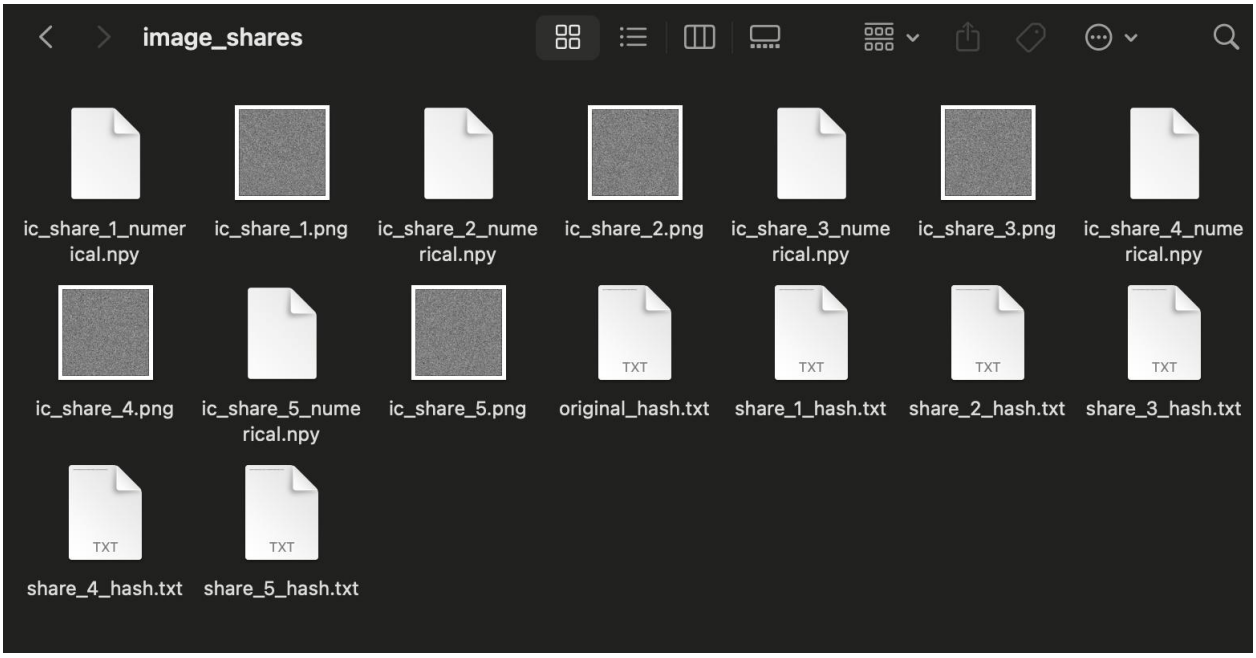
```
Load k numerical shares specified by shareIndices
Verify hashes of loaded shares
If verification fails, return Error
Initialize reconstructedImage array (height, width)
For each pixel (r, c):
  points = []
  For each loaded share i with index idx:
    points.append((idx, share_i[r, c]))
  secret = LagrangeInterpolate(points, x_target=0, P)
  reconstructedImage[r, c] = secret
finalImage = ClipAndConvertToUint8(reconstructedImage)
Verify Hash(finalImage % P) against originalHash
Return finalImage
```

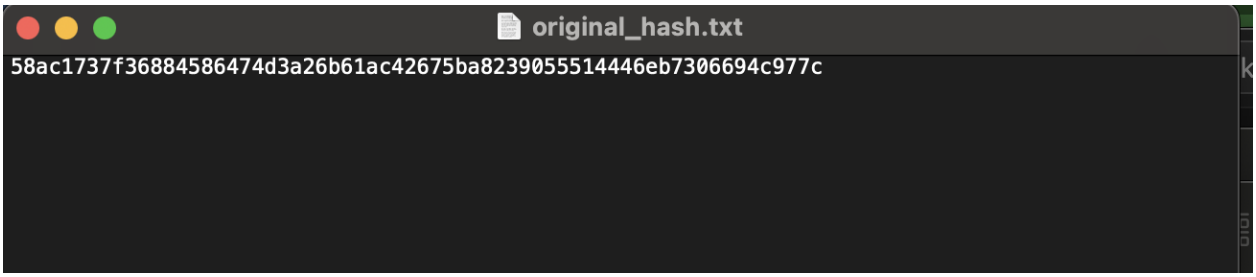
Output:

```
(base) Rohiths-MacBook-Pro:DAA2 rgk$ python test.py
Enter the path to the image file: ./ic.png
Enter the total number of shares to create (n): 5
Enter the threshold number of shares needed to reconstruct (k <= 5): 3
------------------------------
1. Preprocessing Image...
Image loaded successfully (512x512).
Original image hash (58ac1737...) saved to image_shares/original_hash.txt
------------------------------
2. Creating Shares...
Creating 5 shares (k=3) for image of size 512x512...
  Processed row 51/512
  Processed row 102/512
  Processed row 153/512
  Processed row 204/512
  Processed row 255/512
  Processed row 306/512
  Processed row 357/512
  Processed row 408/512
  Processed row 459/512
  Processed row 510/512
  Processed row 512/512
Share creation complete.
------------------------------
3. Saving Shares and Hashes...
Saved visual share 1 to image_shares/ic_share_1.png
Saved numerical share 1 data to image_shares/ic_share_1_numerical.npy
Saved hash for share 1 to image_shares/share_1_hash.txt
Saved visual share 2 to image_shares/ic_share_2.png
Saved numerical share 2 data to image_shares/ic_share_2_numerical.npy
Saved hash for share 2 to image_shares/share_2_hash.txt
Saved visual share 3 to image_shares/ic_share_3.png
Saved numerical share 3 data to image_shares/ic_share_3_numerical.npy
Saved hash for share 3 to image_shares/share_3_hash.txt
Saved visual share 4 to image_shares/ic_share_4.png
Saved numerical share 4 data to image_shares/ic_share_4_numerical.npy
Saved hash for share 4 to image_shares/share_4_hash.txt
Saved visual share 5 to image_shares/ic_share_5.png
Saved numerical share 5 data to image_shares/ic_share_5_numerical.npy
Saved hash for share 5 to image_shares/share_5_hash.txt
Shares and hashes saved in directory: image_shares
------------------------------
4. Simulating Reconstruction...
Enter 3 share indices (e.g., '1 3 5') to use for reconstruction: 1 2 3
Attempting reconstruction using shares: [1, 2, 3]
Loaded and verified numerical share 1 from image_shares/ic_share_1_numerical.npy
Loaded and verified numerical share 2 from image_shares/ic_share_2_numerical.npy
Loaded and verified numerical share 3 from image_shares/ic_share_3_numerical.npy
Reconstructing image from 3 shares (k=3)...
  Reconstructed row 51/512
  Reconstructed row 102/512
  Reconstructed row 153/512
  Reconstructed row 204/512
  Reconstructed row 255/512
  Reconstructed row 306/512
  Reconstructed row 357/512
  Reconstructed row 408/512
  Reconstructed row 459/512
  Reconstructed row 510/512
  Reconstructed row 512/512
Image reconstruction complete.
------------------------------
5. Verifying Integrity...
Reconstructed image hash: 58ac1737...
Original image hash:      58ac1737...
Integrity Check PASSED: Reconstructed image matches the original.
------------------------------
6. Displaying Results...
```

Output:


Original Image
Share 1 (Visual)
Share 2 (Visual)
Share 3 (Visual)
Reconstructed (from 3/5 shares)


image_shares
ic_share_1_numerical.npy
ic_share_1.png
ic_share_2_numerical.npy
ic_share_2.png
ic_share_3_numerical.npy
ic_share_3.png
ic_share_4_numerical.npy
ic_share_4.png
ic_share_5_numerical.npy
ic_share_5.png
original_hash.txt
share_1_hash.txt
share_2_hash.txt
share_3_hash.txt
share_4_hash.txt
share_5_hash.txt


original_hash.txt
58ac1737f36884586474d3a26b61ac42675ba8239055514446eb7306694c977c


share_1_hash.txt
3ee98b2f747ad20f621583c81246ee17e954635d8833f4070533e5c3a0e7646a

Algorithm 2: SSS with N+1 Share (Code 2)

Function ShareImage(imageData, k, n, P=251):

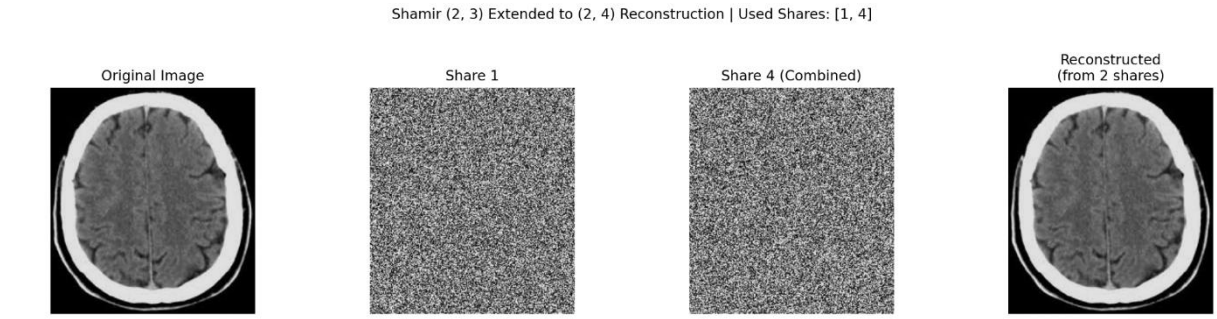
```
// Step 1: Create n shares as in Algorithm 1 (using P=251)
shareArrays_n = BasicShareCreation(imageData, k, n, P)
// Step 2: Create n+1 share
share_nplus1 = Sum(shareArrays_n) % P
allShares = shareArrays_n + [share_nplus1]
// Step 3: Save all n+1 shares (numerical, visual, hashes) as in Algorithm 1
Return success
```

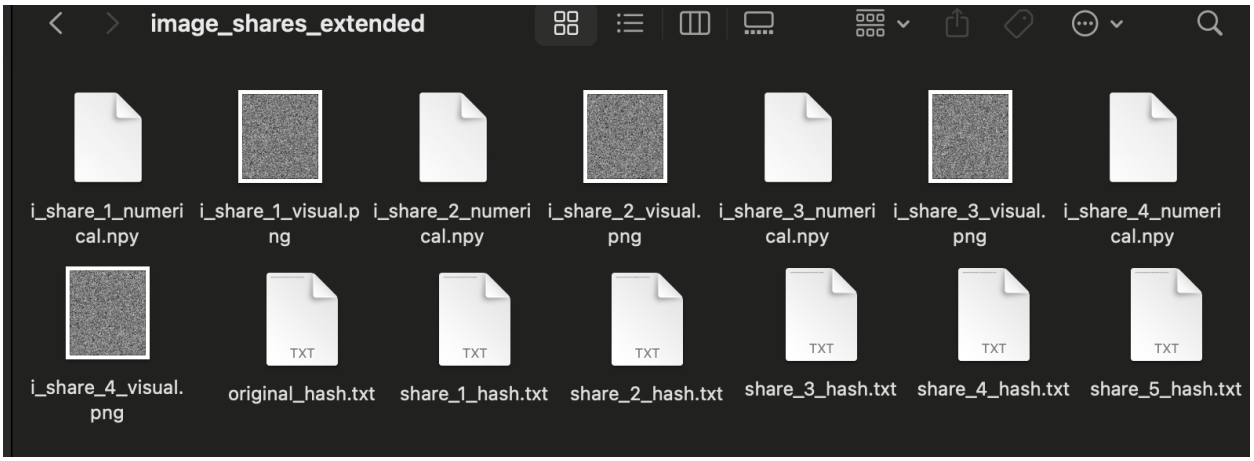
Function ReconstructImage(shareIndices, k, n_orig, P=251):

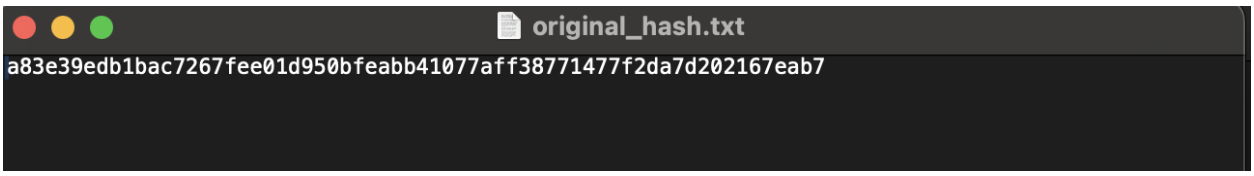
```
Load k numerical shares specified by shareIndices
Verify hashes of loaded shares
If verification fails, return Error
Initialize reconstructedImage array
n_plus_1_present = (n_orig + 1) in shareIndices
If n_plus_1_present:
  powerSums = PrecomputePowerSums(n_orig, k-1, P)
For each pixel (r, c):
  If n_plus_1_present:
    matrixM, vectorY = BuildLinearSystem(shares, shareIndices, r, c, powerSums, k, P)
    coeffs = SolveLinearSystemModP(matrixM, vectorY, P)
    secret = coeffs[0]
  Else:
    points = BuildLagrangePoints(shares, shareIndices, r, c)
    secret = LagrangeInterpolate(points, x_target=0, P)
  reconstructedImage[r, c] = secret
finalImage = ClipAndConvertToUint8(reconstructedImage)
Verify Hash(finalImage % P) against originalHash
Return finalImage
```

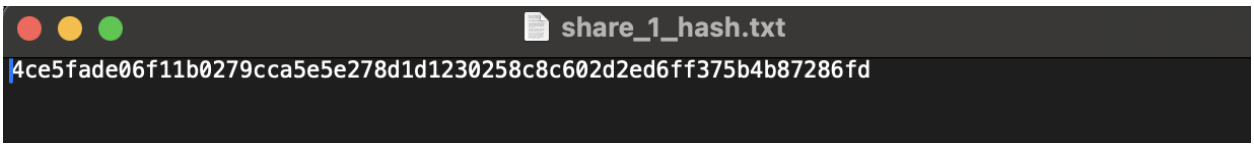
Output:

```
(base) Rohiths-MacBook-Pro:DAA2 rgk$ python test.py
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
Enter the path to the image file: ./i.jpeg
Enter the initial number of shares (n >= 2): 3
Enter the threshold (k, where 2 <= k <= 3): 2
------------------------------
1. Preprocessing Image...
Image loaded successfully (213x236).
Original image hash (using mod 251, a83e39ed...) saved.
------------------------------
2. Creating Initial 3 Shares (k=2)...
Creating 3 shares (k=2) for image of size 213x236 using PRIME=251...
  Processed row 23/236
  Processed row 46/236
  Processed row 69/236
  Processed row 92/236
  Processed row 115/236
  Processed row 138/236
  Processed row 161/236
  Processed row 184/236
  Processed row 207/236
  Processed row 230/236
  Processed row 236/236
Share creation complete.
------------------------------
3. Creating Share 4 using Participant Increasing Method...
Creating the 4-th share by summing shares 1 to 3 (mod 251)...
Share 4 created.
------------------------------
4. Saving All Shares and Hashes...
Saved 4 shares (visual+numerical) and hashes.
All 4 shares and hashes saved in directory: image_shares_extended
------------------------------
5. Simulating Reconstruction (Requires k=2 shares from 4 total)...

Total available shares: 1 to 4
Enter exactly 2 unique share indices (e.g., '1 3 4') or 'q' to quit: 1 3 4
Error: Please provide exactly 2 unique indices. You provided 3.

Total available shares: 1 to 4
Enter exactly 2 unique share indices (e.g., '1 3 4') or 'q' to quit: 1 4

Attempting reconstruction using shares: [1, 4]
Attempting to load numerical shares for indices: [1, 4]
 -> Loaded and verified numerical share 1
 -> Loaded and verified numerical share 4
Reconstructing image from 2 shares (k=2, n_orig=3). Share 4 present: True
 -> Using linear system solving method.
  Reconstructed row 23/236
  Reconstructed row 46/236
  Reconstructed row 69/236
  Reconstructed row 92/236
  Reconstructed row 115/236
  Reconstructed row 138/236
  Reconstructed row 161/236
  Reconstructed row 184/236
  Reconstructed row 207/236
  Reconstructed row 230/236
  Reconstructed row 236/236
Image reconstruction computation complete.
------------------------------
6. Verifying Integrity...
Reconstructed image hash (using mod 251): a83e39ed...
Original image hash      (using mod 251): a83e39ed...
Integrity Check PASSED: Reconstructed image matches the original (mod {PRIME}).
Note: Minor visual differences possible if original had pixels > 250.
```


Shamir (2, 3) Extended to (2, 4) Reconstruction | Used Shares: [1, 4]
Original Image
Share 1
Share 4 (Combined)
Reconstructed
(from 2 shares)


image_shares_extended
i_share_1_numerical.npy
i_share_1_visual.png
i_share_2_numerical.npy
i_share_2_visual.png
i_share_3_numerical.npy
i_share_3_visual.png
i_share_4_numerical.npy
i_share_4_visual.png
original_hash.txt
share_1_hash.txt
share_2_hash.txt
share_3_hash.txt
share_4_hash.txt
share_5_hash.txt


original_hash.txt
a83e39edb1bac7267fee01d950bfeabb41077aff38771477f2da7d202167eab7


share_1_hash.txt
4ce5fade06f11b0279cca5e5e278d1d1230258c8c602d2ed6ff375b4b87286fd

Algorithm 3: Parallel SSS with N+1 Share (Code 3)

Function ShareImageParallel(imageData, k, n, P=251):

```
Divide imageData into row chunks
Create tasks for each chunk
Use ProcessPoolExecutor to run _create_shares_chunk_processor on tasks:
  Worker(_create_shares_chunk_processor):
    Processes its chunk pixel by pixel (like Algo 1)
    Returns n share chunks
Assemble results from workers into n full shareArrays_n
share_nplus1 = Sum(shareArrays_n) % P
allShares = shareArrays_n + [share_nplus1]
Save all n+1 shares (numerical, visual, hashes)
Return success
```

Function ReconstructImageParallel(shareIndices, k, n_orig, P=251):

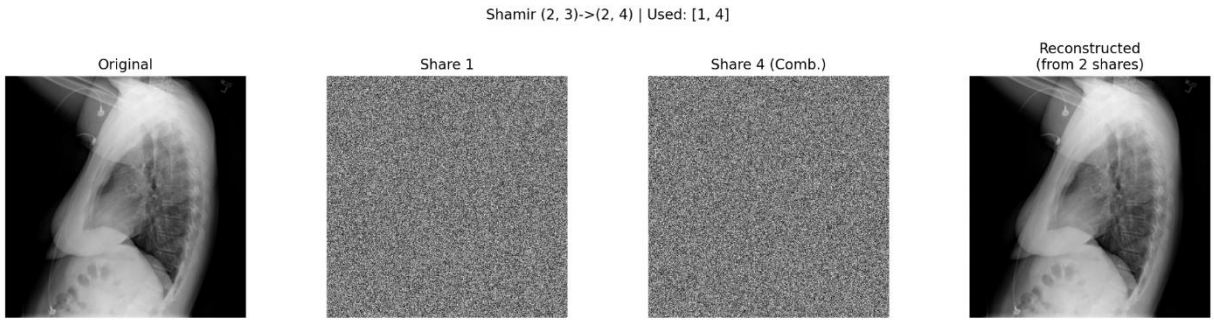
```
Load and Verify k shares
If verification fails, return Error
Initialize reconstructedImage array
n_plus_1_present = (n_orig + 1) in shareIndices
If n_plus_1_present: powerSums = PrecomputePowerSums(...)
Divide reconstruction task into chunks (based on rows)
Create tasks for each chunk (passing relevant share data slices)
Use ProcessPoolExecutor to run _reconstruct_chunk_processor on tasks:
  Worker(_reconstruct_chunk_processor):
    Processes its chunk pixel by pixel (like Algo 2 Recon)
    Returns reconstructed chunk
Assemble results from workers into full reconstructedImage
finalImage = ClipAndConvertToUint8(reconstructedImage)
Verify Hash(finalImage % P) against originalHash
Return finalImage
```

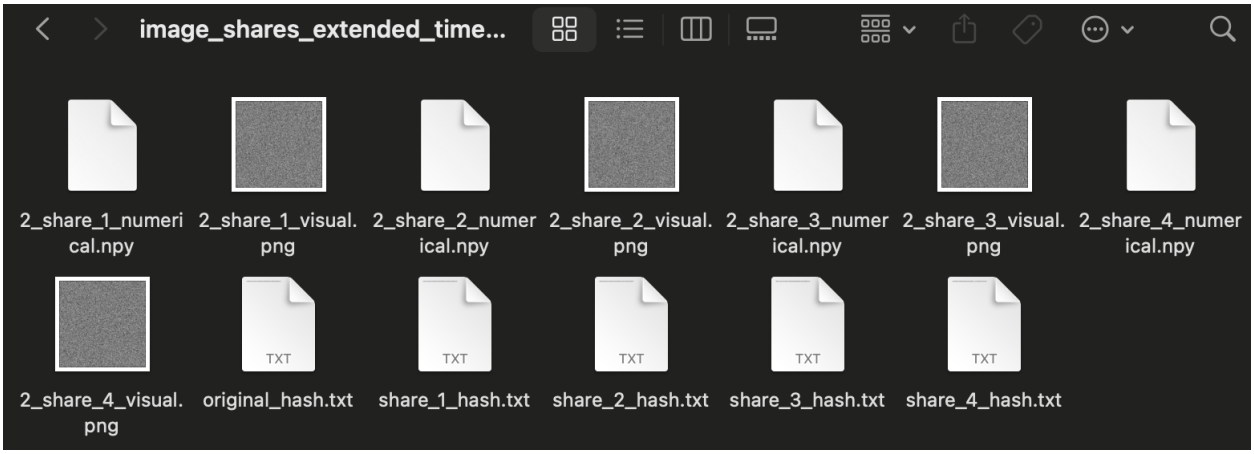
Output:

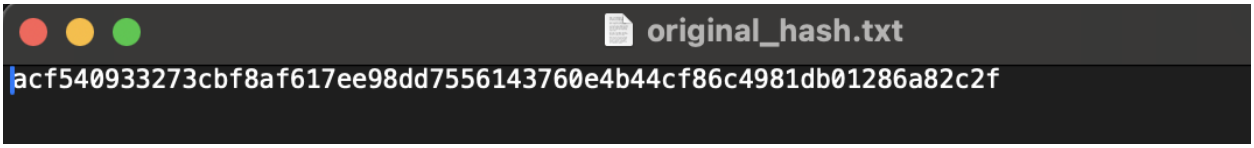
```
[(base) Rohiths-MacBook-Pro:DAA2 rgk$ python test.py
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
Enter the path to the image file: ./2.jpeg
Enter initial number of shares (n >= 2): 3
Enter threshold (k, 2 <= k <= 3): 2
------------------------------
1. Preprocessing Image...
Image loaded (512x512).
Original hash (mod 251, acf54093...) saved.
------------------------------
2/3. Creating 4 Total Shares (k=2)...
Creating 3 shares (k=2) in PARALLEL using 16 workers...
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
  Worker finished chunk starting at row 0
  Worker finished chunk starting at row 128
  Worker finished chunk starting at row 224
  Worker finished chunk starting at row 32
  Worker finished chunk starting at row 160
  Worker finished chunk starting at row 64
  Worker finished chunk starting at row 256
  Worker finished chunk starting at row 192
  Worker finished chunk starting at row 96
  Worker finished chunk starting at row 320
  Worker finished chunk starting at row 288
  Worker finished chunk starting at row 352
  Worker finished chunk starting at row 384
  Worker finished chunk starting at row 416
  Worker finished chunk starting at row 448
  Worker finished chunk starting at row 480
Assembling results...
Parallel share creation complete.
Creating the 4-th share by summing shares 1 to 3 (mod 251)...
Share 4 created.
Parallel Encryption Time: 1.9937 seconds
Run sequential encryption for comparison? (y/n): y
Creating 3 shares (k=2) SEQUENTIALLY...
```

```
Creating 3 shares (k=2) SEQUENTIALLY...
  Processed row 51/512
  Processed row 102/512
  Processed row 153/512
  Processed row 204/512
  Processed row 255/512
  Processed row 306/512
  Processed row 357/512
  Processed row 408/512
  Processed row 459/512
  Processed row 510/512
  Processed row 512/512
Sequential share creation complete.
Creating the 4-th share by summing shares 1 to 3 (mod 251)...
Share 4 created.
Sequential Encryption Time: 0.8439 seconds
Note: Verification between parallel/sequential shares not implemented.
------------------------------
4. Saving All Shares and Hashes...
Saved 4 shares (visual+numerical) and hashes.
------------------------------
5. Simulating Reconstruction (Requires k=2 shares)...

Total available shares: 1 to 4
Enter 2 unique share indices (e.g., '1 3 4') or 'q' to quit: 1 3 4
Error: Need exactly 2 indices.

Total available shares: 1 to 4
Enter 2 unique share indices (e.g., '1 3 4') or 'q' to quit: 1 4

Attempting reconstruction using shares: [1, 4]
Attempting to load numerical shares for indices: [1, 4]
 -> Loaded and verified numerical share 1
 -> Loaded and verified numerical share 4
Reconstructing image PARALLEL (k=2, n_orig=3). Share 4 present: True
 -> Using linear system solving method.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
```

```
INFO: Using 16 workers for parallel processing.
  Worker finished recon chunk starting at row 128
  Worker finished recon chunk starting at row 192
  Worker finished recon chunk starting at row 224
  Worker finished recon chunk starting at row 64
  Worker finished recon chunk starting at row 96
  Worker finished recon chunk starting at row 32
  Worker finished recon chunk starting at row 160
  Worker finished recon chunk starting at row 288
  Worker finished recon chunk starting at row 0
  Worker finished recon chunk starting at row 256
  Worker finished recon chunk starting at row 480
  Worker finished recon chunk starting at row 352
  Worker finished recon chunk starting at row 416
  Worker finished recon chunk starting at row 320
  Worker finished recon chunk starting at row 448
  Worker finished recon chunk starting at row 384
Assembling reconstruction results...
Parallel image reconstruction complete.
Parallel Decryption Time: 5.0492 seconds
Run sequential decryption for comparison? (y/n): y
Reconstructing image SEQUENTIAL (k=2, n_orig=3). Share 4 present: True
 -> Using linear system solving method.
  Reconstructed row 51/512
  Reconstructed row 102/512
  Reconstructed row 153/512
  Reconstructed row 204/512
  Reconstructed row 255/512
  Reconstructed row 306/512
  Reconstructed row 357/512
  Reconstructed row 408/512
  Reconstructed row 459/512
  Reconstructed row 510/512
  Reconstructed row 512/512
Sequential image reconstruction complete.
Sequential Decryption Time: 23.0016 seconds
 -> Sequential and Parallel reconstructions match.
------------------------------
6. Verifying Integrity...
Recon hash (mod 251): acf54093...
Orig hash  (mod 251): acf54093...
Integrity Check PASSED.
------------------------------
7. Displaying Results (using parallel reconstruction)...

--- Timing Summary ---
Parallel Encryption:   1.9937 sec
Sequential Encryption: 0.8439 sec
Parallel Decryption:   5.0492 sec
Sequential Decryption: 23.0016 sec
 -> Encryption Speedup: 0.42x
 -> Decryption Speedup: 4.56x
(base) Rohiths-MacBook-Pro:DAA2 rgk$
```


Shamir (2, 3)->(2, 4) | Used: [1, 4]
Original
Share 1
Share 4 (Comb.)
Reconstructed
(from 2 shares)


image_shares_extended_time...
2_share_1_numerical.npy
2_share_1_visual.png
2_share_2_numerical.npy
2_share_2_visual.png
2_share_3_numerical.npy
2_share_3_visual.png
2_share_4_numerical.npy
2_share_4_visual.png
original_hash.txt
share_1_hash.txt
share_2_hash.txt
share_3_hash.txt
share_4_hash.txt
TXT


original_hash.txt
acf540933273cbf8af617ee98dd7556143760e4b44cf86c4981db01286a82c2f


share_2_hash.txt
091d03201a9b4a2ea7432267b9f219ea46a902c2e911647f210c458c5dfb84ff

Algorithm 4: Encrypted Parallel SSS with N+1 Share (Code 4)

Function ShareImageEncryptedParallel(imageData, k, n, P=251):

```
  // Generate shares as in Algorithm 3
  allShares = CreateSharesParallelAndNPlus1(...)
  Get encryption password from user
  For i from 1 to n+1:
    shareHash = Hash(allShares[i-1]) // Hash plaintext
    Save shareHash as ShareHash_i.txt
    encryptedData = EncryptData(allShares[i-1].tobytes(), password) // Encrypt numerical
    Save encryptedData as NumericalShare_i.enc
    Save VisualShare(allShares[i-1]) as VisualShare_i.png
  Return success
```

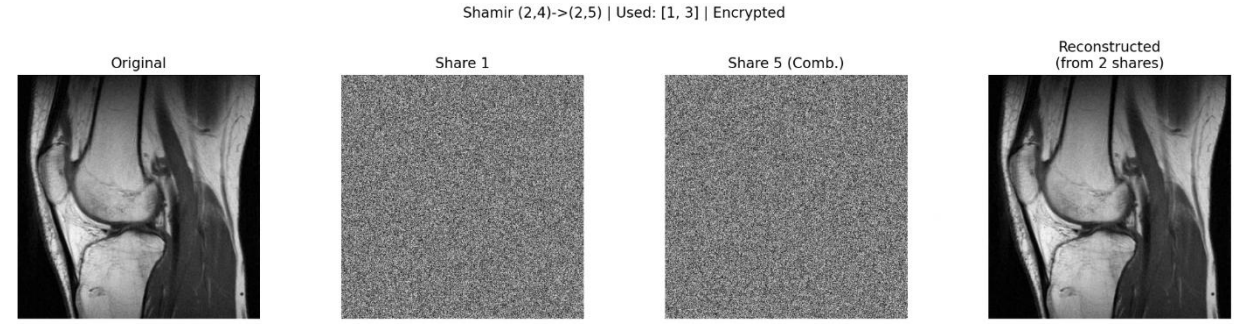
Function ReconstructImageEncryptedParallel(shareIndices, k, n_orig, P=251):

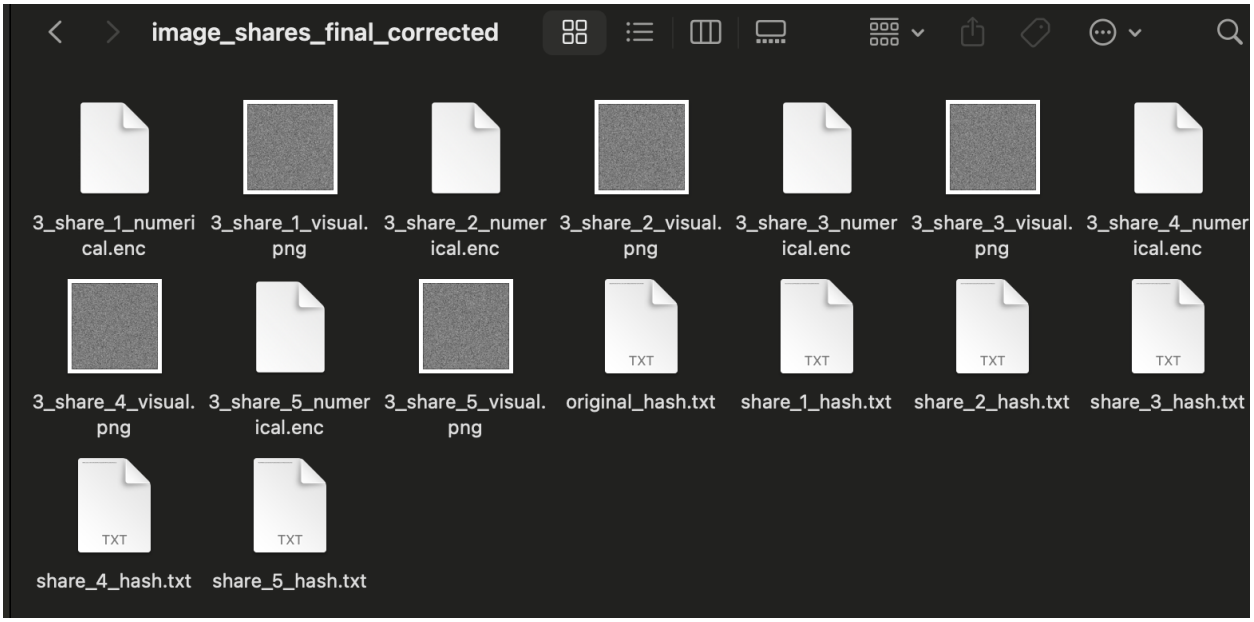
```
  Get decryption password from user
  Load k shares:
    For each index idx in shareIndices:
      Load hash_idx
      Load encryptedData_idx
      Try:
        plainBytes = DecryptData(encryptedData_idx, password)
        numericalShare = ReshapeBytesToNumpy(plainBytes, idx) # Infer shape
        calculatedHash = Hash(numericalShare)
        If calculatedHash != hash_idx: Raise IntegrityError
        Store numericalShare
      Catch DecryptionError/IntegrityError: Return Error
  // Reconstruct using loaded plaintext shares as in Algorithm 3
  reconstructedImage = ReconstructParallelUsingPlainShares(...)
  finalImage = ClipAndConvertToUint8(reconstructedImage)
  Verify Hash(finalImage % P) against originalHash
  Return finalImage
```

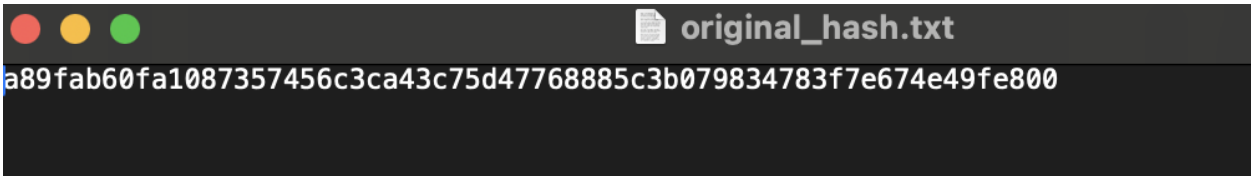
Output:

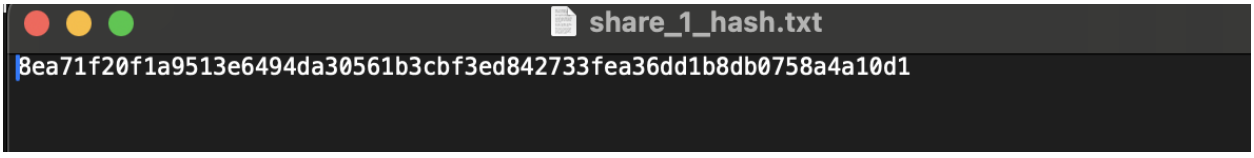
```
[(base) Rohiths-MacBook-Pro:DAA2 rgk$ python test.py
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 Enter path to image file: ./3.jpeg
 Enter initial number of shares (n >= 2): 4
 Enter threshold (k, 2 <= k <= 4): 2
 ------------------------------
 1. Preprocessing Image...
 Image loaded (512x512).
 Original hash (mod 251) saved.
 ------------------------------
 2/3. Creating 5 Shares (k=2)...
 Creating 4 shares (k=2) PARALLEL (16 workers)...
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using 16 workers for parallel processing.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
 INFO: Using 16 workers for parallel processing.
 Assembling created shares...
 Parallel share creation complete.
 Parallel Share Generation Time: 1.8202 sec
 Run sequential generation for comparison? (y/n): y
 Creating 4 shares (k=2) SEQUENTIALLY...
 Sequential share creation complete.
 Sequential Gen Time: 0.9498 sec
 ------------------------------
 4. Saving Shares (Encrypting Numerical)...

 --- Encryption ---
[Set password to ENCRYPT numerical shares:
[Confirm password:
 Password set for saving.
 Saving shares (Numerical data ENCRYPTED)...
 Saved 5 shares (Encrypted Num + Visual) and Hashes.
 ------------------------------
 5. Simulating Reconstruction (k=2)...

 Available shares: 1 to 5
 Enter 2 unique share indices (e.g., '1 3 5') or 'q' to quit: 1 3

 Attempting load/decrypt for shares: [1, 3]
```

```
Attempting load/decrypt for shares: [1, 3]
Loading/Decrypting shares: [1, 3]
 -> Decrypted and verified share 1
 -> Decrypted and verified share 3
Reconstructing PARALLEL (k=2, n=5). N+1 Present: False
 -> Lagrange method.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
INFO: Using PRIME = 251. Pixel values > 250 will be reduced modulo 251.
INFO: Using 16 workers for parallel processing.
Assembling reconstructed image...
Parallel image reconstruction complete.
Parallel Decrypt+Recon Time: 1.7437 sec
Run sequential decrypt/recon for comparison? (y/n): y
Reconstructing SEQUENTIAL (k=2, n=5). N+1 Present: False
 -> Lagrange method.
Processing pixels sequentially...
Sequential image reconstruction complete.
Sequential Decrypt+Recon Time: 1.0427 sec
------------------------------
6. Verifying Integrity...
Recon hash (mod 251): a89fab60...
Orig hash  (mod 251): a89fab60...
Integrity Check PASSED.
------------------------------
7. Displaying Results...
^Z
[8]+  Stopped                 python test.py
(base) Rohiths-MacBook-Pro:DAA2 rgk$
```


Shamir (2,4)->(2,5) | Used: [1, 3] | Encrypted
Original
Share 1
Share 5 (Comb.)
Reconstructed
(from 2 shares)


image_shares_final_corrected
3_share_1_numerical.enc
3_share_1_visual.png
3_share_2_numerical.enc
3_share_2_visual.png
3_share_3_numerical.enc
3_share_3_visual.png
3_share_4_numerical.enc
3_share_4_visual.png
3_share_5_numerical.enc
3_share_5_visual.png
original_hash.txt
share_1_hash.txt
share_2_hash.txt
share_3_hash.txt
share_4_hash.txt
share_5_hash.txt
TXT


original_hash.txt
a89fab60fa1087357456c3ca43c75d47768885c3b079834783f7e674e49fe800


share_1_hash.txt
8ea71f20f1a9513e6494da30561b3cbf3ed842733fea36dd1b8db0758a4a10d1

5.Visual cryptography:

```
DEFINE BLACK_VAL = 0       // Value representing Black pixels

DEFINE WHITE_VAL = 255     // Value representing White pixels

// Define the standard complementary 2x2 subpixel patterns (0=Internal Black, 1=Internal White)

DEFINE PATTERN_A = [[0, 1], [1, 0]]

DEFINE PATTERN_B = [[1, 0], [0, 1]]
```

Function PreprocessImageVC(imagePath, threshold)

```
    Input: imagePath (String), threshold (Integer, e.g., 128)

    Output: binaryImage (2D Array of BLACK_VAL or WHITE_VAL) OR ErrorStatus

    // Load the image specified by imagePath

    imageData = LoadImageFromFile(imagePath)

    If imageData is Error: Return ErrorStatus("File load failed")

    // Convert to grayscale

    grayscaleImage = ConvertToGrayscale(imageData)

    If grayscaleImage is Error: Return ErrorStatus("Grayscale conversion failed")

    // Threshold to create binary image

    binaryImage = CreateArray(dimensions=grayscaleImage.dimensions, dtype=uint8)

    For each pixel (r, c) in grayscaleImage:

        If grayscaleImage[r, c] >= threshold:

            binaryImage[r, c] = WHITE_VAL

        Else:

            binaryImage[r, c] = BLACK_VAL
```

```
        End If
    End For

    Return binaryImage
End Function
```

Function GenerateVCShares(binaryImage)

```
    Input: binaryImage (2D Array of BLACK_VAL/WHITE_VAL)
    Output: (share1, share2) (Tuple of 2D Arrays) OR ErrorStatus
    If binaryImage is Null or Error: Return ErrorStatus("Invalid binary image input")
    height, width = GetDimensions(binaryImage)
    shareHeight = height * 2
    shareWidth = width * 2
    // Create blank share canvases (initialized to White)
    share1 = CreateArray(dimensions=(shareHeight, shareWidth), dtype=uint8,
initialValue=WHITE_VAL)
    share2 = CreateArray(dimensions=(shareHeight, shareWidth), dtype=uint8,
initialValue=WHITE_VAL)
    // Convert internal 0/1 patterns to actual pixel values (0/255)
    patternA_img = PATTERN_A * WHITE_VAL
    patternB_img = PATTERN_B * WHITE_VAL
    // Process each pixel of the original binary image
    For r from 0 to height-1:
        For c from 0 to width-1:
```

```
        pixelValue = binaryImage[r, c]
        blockRowStart = r * 2
        blockColStart = c * 2
        // Randomly choose which of the complementary patterns goes to which share first
        // This ensures shares look random even if the pattern choice wasn't random
        usePattern1, usePattern2 = RandomlySelectOneOf([(patternA_img, patternB_img), (patternB_img, patternA_img)])

        If pixelValue == WHITE_VAL:
            // WHITE original pixel: Both shares get the SAME randomly chosen pattern
            chosenPattern = RandomlySelectOneOf([usePattern1, usePattern2])
            // Assign block to Share 1
            share1[blockRowStart : blockRowStart+2, blockColStart : blockColStart+2] = chosenPattern
            // Assign the SAME block to Share 2
            share2[blockRowStart : blockRowStart+2, blockColStart : blockColStart+2] = chosenPattern
        Else If pixelValue == BLACK_VAL:
            // BLACK original pixel: Shares get COMPLEMENTARY patterns
            // Assign the first chosen pattern to Share 1
            share1[blockRowStart : blockRowStart+2, blockColStart : blockColStart+2] = usePattern1
                    share2[blockRowStart : blockRowStart+2, blockColStart : blockColStart+2] = usePattern2
```

```
            End If
        End For
    End For
    Return (share1, share2)
End Function
```

Function ReconstructVCFromShares(share1, share2)

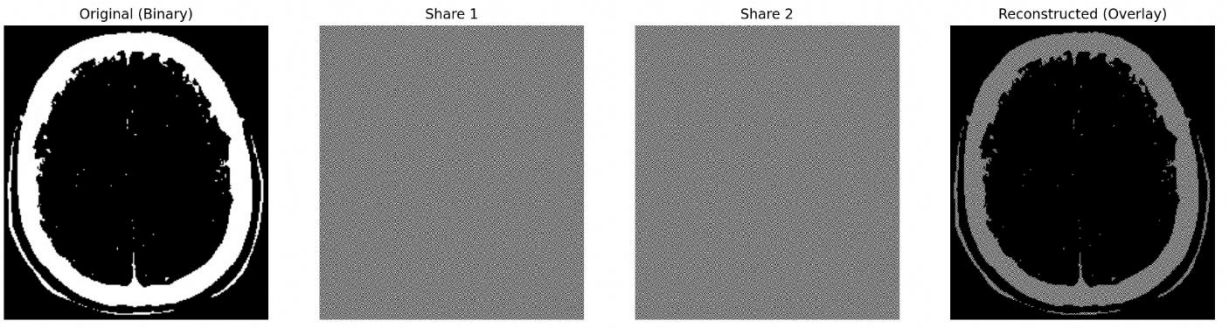
```
    Input: share1 (2D Array), share2 (2D Array)
    Output: reconstructedImage (2D Array) OR ErrorStatus
    If share1 is Null or share2 is Null: Return ErrorStatus("Invalid share input")
    If GetDimensions(share1) != GetDimensions(share2):
        Return ErrorStatus("Share dimensions must match")
    // Simulate physical overlay by taking the element-wise minimum
    // Assumes BLACK_VAL (0) < WHITE_VAL (255). Black + Anything = Black.
    reconstructedImage = ElementWiseMinimum(share1, share2)
    Return reconstructedImage
End Function
```

Algorithm MainVCWorkflow

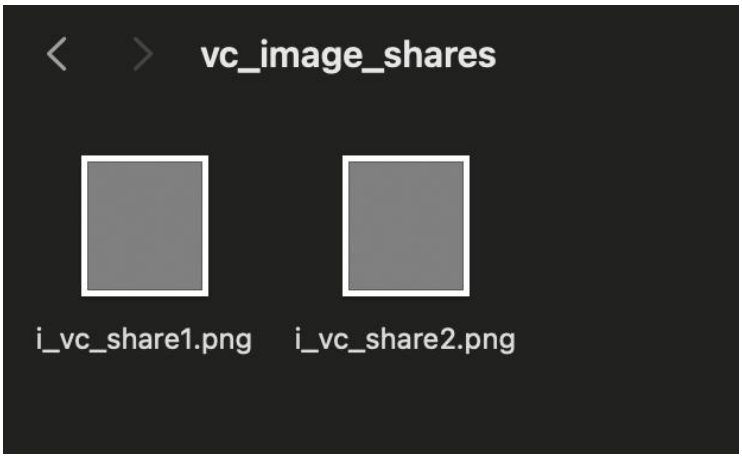
```
    imageFile = GetUserInput("Path to image")
    threshold = 128 // Example threshold

    binaryImg = PreprocessImageVC(imageFile, threshold)
```

```
    If binaryImg is Error: ExitWithError
    sharesResult = GenerateVCShares(binaryImg)
    If sharesResult is Error: ExitWithError
    shareImg1, shareImg2 = sharesResult
    // Optional: Save shares
    SaveImageToFile(shareImg1, "share1.png")
    SaveImageToFile(shareImg2, "share2.png")
    // Reconstruct
    reconImg = ReconstructVCFromShares(shareImg1, shareImg2)
    If reconImg is Error: ExitWithError
    // Optional: Display results
    DisplayImages(binaryImg, shareImg1, shareImg2, reconImg)
    Return Success
End Algorithm
```

Output:



```
(base) Rohiths-MacBook-Pro:DAA2 rgk$ python vc.py
Enter the path to the image file: ./i.jpeg
------------------------------
1. Preprocessing Image for Visual Cryptography...
Image loaded and binarized (213x236).
------------------------------
2. Creating Visual Cryptography Shares...
Creating 2 VC shares for binary image of size 213x236...
  Processed row 23/236
  Processed row 46/236
  Processed row 69/236
  Processed row 92/236
  Processed row 115/236
  Processed row 138/236
  Processed row 161/236
  Processed row 184/236
  Processed row 207/236
  Processed row 230/236
  Processed row 236/236
VC Share creation complete.
------------------------------
3. Saving VC Shares...
Created directory: vc_image_shares
Saved VC Share 1 to vc_image_shares/i_vc_share1.png
Saved VC Share 2 to vc_image_shares/i_vc_share2.png
VC Shares saved in directory: vc_image_shares
------------------------------
4. Simulating Reconstruction by Overlay...
Reconstructing by overlaying shares...
Reconstruction complete.
------------------------------
5. Displaying Results...
```



6.Chinese Remainder Theorem(CRT), Ashmuth bloom:

Function SelectAndCheckModuliCRT(n, k, p0, CANDIDATE_MODULI)

Input: n (Integer, total shares), k (Integer, threshold), p0 (Integer), CANDIDATE_MODULI (List)

Output: selectedModuli (List of n integers) OR ErrorStatus

// 1. Select Moduli

If n > length(CANDIDATE_MODULI):

Return ErrorStatus("Not enough candidate moduli defined for n shares")

selectedModuli = GetFirstNItems(CANDIDATE_MODULI, n)

// 2. Verify Coprimality (Optional if candidates are pre-verified)

// For m in selectedModuli: If GCD(p0, m) != 1: Return ErrorStatus(...)

// For i, j pairs in selectedModuli: If GCD(moduli[i], moduli[j]) != 1: Return ErrorStatus(...)

// 3. Check Asmuth-Bloom Condition

sortedModuli = Sort(selectedModuli)

prodSmallestK = CalculateProduct(sortedModuli[0]...sortedModuli[k-1])

prodLargestKminus1 = 1

If k > 1:

prodLargestKminus1 = CalculateProduct(sortedModuli[n-k+1]...sortedModuli[n-1])

End If

If prodSmallestK <= p0 * prodLargestKminus1:

```
        Print Warning("Asmuth-Bloom condition NOT satisfied! Security/Reconstruction risk.")
        // Optionally: Return ErrorStatus or require confirmation to proceed
    Else:
        Print "Asmuth-Bloom condition satisfied."
    Return selectedModuli
End Function
```

Function CreateCRTShares(imageData, n, k, p0, moduli)

```
    Input: imageData (2D Array, uint8), n, k, p0 (Integers), moduli (List of n integers)
    Output: sharesData (List of n 2D Arrays, int64) OR ErrorStatus
    height, width = GetDimensions(imageData)
    sharesData = CreateListOfArrays(n, dimensions=(height, width), dtype=int64)
    // 1. Calculate A_max (Range for random component)
    sortedModuli = Sort(moduli)
    prodSmallestK = CalculateProduct(sortedModuli[0]...sortedModuli[k-1])
    A_max = floor(prodSmallestK / p0) - 1
    If A_max < 0:
        Return ErrorStatus("Invalid A_max calculated. Check moduli/k/p0.")
    // 2. Generate Shares for Each Pixel
    For r from 0 to height-1:
        For c from 0 to width-1:
            secret S = ConvertToInt(imageData[r, c])
            // a. Generate random component A
```

```
            A = GenerateRandomInteger(0, A_max)
            // b. Calculate encoded value y
            y = S + A * p0
            // c. Calculate share values as remainders
            For i from 0 to n-1:
                sharesData[i][r, c] = y MOD moduli[i]
            End For
        End For
        // Optional: Print Progress
    End For
    Return sharesData
End Function
```

Function ReconstructCRTImage(kSharesData, kShareIndices, k, p0, allModuli)

```
    Input: kSharesData (List of k 2D Arrays, int64), kShareIndices (List of k integers, 1..n), k, p0 (Integers), allModuli (List of n integers)
    Output: reconstructedImage (2D Array, uint8) OR ErrorStatus
    // 1. Validate Input
    If length(kSharesData) != k OR length(kShareIndices) != k:
        Return ErrorStatus("Mismatch between k and number of shares/indices provided")
    // Check uniqueness and range of kShareIndices
    height, width = GetDimensions(kSharesData[0])
    reconstructedData = CreateArray(dimensions=(height, width), dtype=int64)
    // 2. Identify Moduli for Reconstruction
```

```
moduliForRecon = []
Try:
    For idx in kShareIndices:
        moduliForRecon.Add(allModuli[idx - 1]) // Assumes allModuli is 0-indexed
    End For
Catch IndexError:
    Return ErrorStatus("Invalid share index used")
// 3. Reconstruct Each Pixel
For r from 0 to height-1:
    For c from 0 to width-1:
        // a. Gather remainders for this pixel
        remainders = []
        For i from 0 to k-1:
            remainders.Add(ConvertToInt(kSharesData[i][r, c]))
        End For
        // b. Solve the system using CRT (Assumes CRT_Solve function exists)
        y = CRT_Solve(remainders, moduliForRecon)
        If y is Error:
            reconstructedData[r, c] = 0 // Error handling: set pixel to black
            Print Error("CRT failed at pixel", r, c)
        Else:
            // c. Recover the original secret
```

```
            secret S = y MOD p0
            reconstructedData[r, c] = S
        End If
    End For
    // Optional: Print Progress
End For
// 4. Finalize Image
reconstructedImage = ClipValues(reconstructedData, 0, 255)
reconstructedImage = ConvertToUint8(reconstructedImage)
Return reconstructedImage
End Function
```

Algorithm MainCRTWorkflow

```
// 1. Input & Setup
imageFile = GetUserInput(...)
n = GetUserInput(...)
k = GetUserInput(...)
moduli = SelectAndCheckModuliCRT(n, k, p0, CANDIDATE_MODULI)
If moduli is Error: Exit
// 2. Preprocessing
imageData = PreprocessImage(imageFile) // Load, Grayscale
If imageData is Error: Exit
```

// 3. Share Creation

allShares = CreateCRTShares(imageData, n, k, p0, moduli)

If allShares is Error: Exit

SaveCRTRelatedData(allShares, p0, moduli, baseFilename)

// 5. Reconstruction Request

indicesToUse = GetUserInput("Indices for reconstruction", k) // Get k valid indices

// 6. Loading

loadedShares, loaded_p0, loaded_moduli = LoadCRTRelatedData(indicesToUse, baseFilename)

If loadedShares is Error: Exit

// 7. Reconstruction

reconImage = ReconstructCRTImage(loadedShares, indicesToUse, k, loaded_p0, loaded_moduli)

If reconImage is Error: Exit

originalHash = Hash(imageData) // Calculate or load saved hash

reconHash = Hash(reconImage)

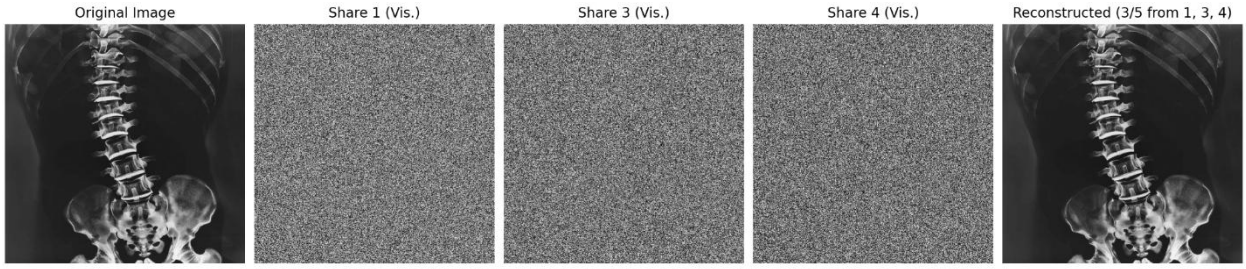
If originalHash == reconHash: Print "Integrity PASSED" Else: Print "Integrity FAILED"

DisplayImage(reconImage)

Return Success

End Algorithm

Output:

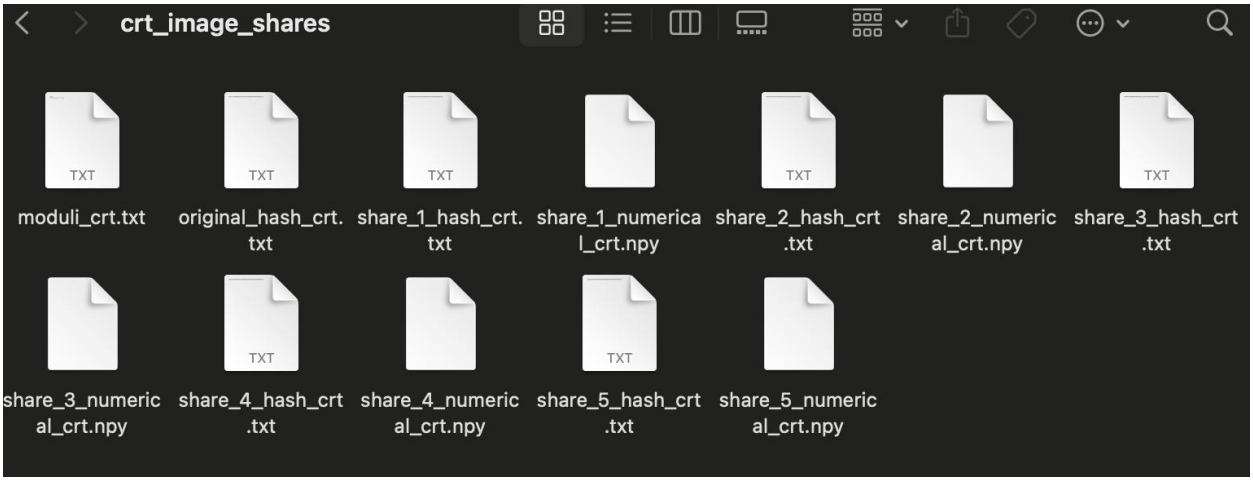


original_hash_crt.txt

```
58ac1737f36884586474d3a26b61ac42675ba8239055514446eb7306694c977c
```

share_1_hash_crt.txt

```
4ea3e6d8e989c70b71bd5a429669640b4e209f4dd0535ba24d2478482f7ebff0
```



```
(base) Rohiths-MacBook-Pro:DAA2 rgk$ python crt.py
Enter the path to the image file: ./ic.png
Enter total number of shares (n <= 45): 5
Enter threshold (k, 2 <= k <= 5): 3
------------------------------
1. Selecting Moduli...
Selected moduli m_1...m_5: [263, 269, 271, 277, 281]
------------------------------
2. Checking Asmuth-Bloom Condition...
  Condition check: Prod(smallest 3) = 19172437
  Condition check: p0 * Prod(largest 2) = 20004109
  WARNING: Asmuth-Bloom condition NOT satisfied for the chosen k, n, and moduli.
  Reconstruction might fail or leak information.
Condition not met. Proceed anyway? (yes/no): yes
------------------------------
3. Preprocessing Image...
Image loaded (512x512).
Original hash (58ac1737...) saved to crt_image_shares/original_hash_crt.txt
------------------------------
4. Creating CRT Shares...
Range for random factor A: [0, 74599]
Creating 5 CRT shares (k=3, p0=257) for image 512x512...
  Processed row 51/512
  Processed row 102/512
  Processed row 153/512
  Processed row 204/512
  Processed row 255/512
  Processed row 306/512
  Processed row 357/512
  Processed row 408/512
  Processed row 459/512
  Processed row 510/512
  Processed row 512/512
CRT Share creation complete.
------------------------------
5. Saving Shares, Hashes, and Moduli...
Saved moduli information to crt_image_shares/moduli_crt.txt
Saved numerical share 1 data to crt_image_shares/share_1_numerical_crt.npy
Saved hash for share 1 to crt_image_shares/share_1_hash_crt.txt
Saved numerical share 2 data to crt_image_shares/share_2_numerical_crt.npy
Saved hash for share 2 to crt_image_shares/share_2_hash_crt.txt
Saved numerical share 3 data to crt_image_shares/share_3_numerical_crt.npy
Saved hash for share 3 to crt_image_shares/share_3_hash_crt.txt
Saved numerical share 4 data to crt_image_shares/share_4_numerical_crt.npy
Saved hash for share 4 to crt_image_shares/share_4_hash_crt.txt
Saved numerical share 5 data to crt_image_shares/share_5_numerical_crt.npy
Saved hash for share 5 to crt_image_shares/share_5_hash_crt.txt
Shares and info saved in directory: crt_image_shares
------------------------------
6. Simulating Reconstruction...
Enter exactly 3 unique share indices (1 to 5, e.g., '1 3 5'): 1 3 5
Attempting reconstruction using shares: [1, 3, 5]
Loaded p0=257 and 5 moduli from crt_image_shares/moduli_crt.txt
Loaded and verified numerical share 1 from crt_image_shares/share_1_numerical_crt.npy
Loaded and verified numerical share 3 from crt_image_shares/share_3_numerical_crt.npy
Loaded and verified numerical share 5 from crt_image_shares/share_5_numerical_crt.npy
Reconstructing image from 3 CRT shares (indices: [1, 3, 5])...
  Reconstructed row 51/512
  Reconstructed row 102/512
  Reconstructed row 153/512
  Reconstructed row 204/512
  Reconstructed row 255/512
  Reconstructed row 306/512
  Reconstructed row 357/512
  Reconstructed row 408/512
  Reconstructed row 459/512
  Reconstructed row 510/512
  Reconstructed row 512/512
```

```
  Reconstructed row 510/512
  Reconstructed row 512/512
Image reconstruction complete.
------------------------------
7. Verifying Integrity...
Reconstructed hash: 58ac1737...
Original hash:      58ac1737...
Integrity Check PASSED.
------------------------------
8. Displaying Results...
```

7.Comparing the Entropy Analysis of CRT and SSS:

We have compared the entropy analysis of the two algorithms, that is CRT and SSS.

Below are the results:

Function CalculateEntropy(dataArray)

Input: dataArray (2D NumPy Array or similar structure containing numerical values)

Output: entropyValue (Float, in bits per value)

If dataArray is Null OR dataArray.size == 0:

Return 0.0

End If

values = Flatten(dataArray) // Get a 1D list/array of all values

uniqueVals, counts = GetUniqueValuesAndCounts(values)

totalCount = length(values)

If totalCount == 0: Return 0.0 // Avoid division by zero

probabilities = counts / totalCount

entropyValue = ShannonEntropy(probabilities, base=2)

Return entropyValue

End Function

Function ReportEntropyAnalysis(originalEntropy, sssAverageEntropy, crtAverageEntropy)

Input: originalEntropy (Float), sssAverageEntropy (Float), crtAverageEntropy (Float)

Output: None (Prints analysis to standard output)

Print "Goal: Shares should have much higher entropy than the original."

Print Format(" - Original Image   : {:.4f}", originalEntropy)

Print Format(" - Avg SSS Share   : {:.4f}", sssAverageEntropy)

Print Format(" - Avg CRT Share   : {:.4f}", crtAverageEntropy)

Print " (Reference: Max entropy for uniform 8-bit (0-255) is 8.0 bits)"

If originalEntropy > 0 AND sssAverageEntropy > originalEntropy AND crtAverageEntropy > originalEntropy:

Print " Observation: Both share types exhibit significantly higher entropy than the original, indicating effective randomization."

Else If originalEntropy > 0:

Print " Observation: Compare share entropies to the original; effective shares should be much higher."

Else:

Print " Observation: Original image entropy is very low or zero."

End If

Print Separator("--- End Entropy Analysis ---")

End Function

Algorithm MainWorkflow:

```
    origEntropy = CalculateEntropy(originalImageData)
    sssEntropiesList = []
    For each sss_share in sssShares:
        sssEntropiesList.Add(CalculateEntropy(sss_share))
    End For
    sssAvg = Mean(sssEntropiesList)
    crtEntropiesList = []
    For each crt_share in crtShares:
        crtEntropiesList.Add(CalculateEntropy(crt_share))
    End For
    crtAvg = Mean(crtEntropiesList)
    ReportEntropyAnalysis(origEntropy, sssAvg, crtAvg)
End Algorithm
```

8.Maxflow Network Algorithm for CRT:

```
// --- Max Flow Algorithm: Edmonds-Karp ---
// Assumes:
//   Graph G = (V, E) with nodes V, edges E
//   Capacity function c(u, v) for each edge (u, v) in E
//   Source node S, Sink node T
```

Function FindAugmentingPathBFS(ResidualGraph G_f, Source S, Sink T)

```
Input: G_f (Graph with residual capacities c_f), S, T (Nodes)
Output: Path (List of nodes from S to T) OR Null (if no path exists)
queue = InitializeQueue()       // Queue for nodes to visit
parentMap = InitializeMap()   // Stores parent[v] = u to reconstruct path
visited = InitializeSet()     // Keep track of visited nodes
Enqueue(queue, S)
MarkVisited(visited, S)
parentMap[S] = Null // Source has no parent
While queue is not empty:
   u = Dequeue(queue)
   For each neighbor v of u in G_f:
      If v is not in visited AND G_f.ResidualCapacity(u, v) > 0:
         parentMap[v] = u          // Set parent for path reconstruction
         MarkVisited(visited, v)
         Enqueue(queue, v)
         If v == T:
            path = ReconstructPath(parentMap, S, T)
            Return path
         End If
   End For
End While
```

Return Null

End Function

Function AugmentFlowAndUpdateGraphs(Path P, FlowGraph F, ResidualGraph G_f)

Input: P (List of nodes representing augmenting path),

F (Graph storing current flows f(u,v)),

G_f (Graph storing residual capacities c_f(u,v))

Output: bottleneckCapacity (The amount flow was increased by)

bottleneckCapacity = Infinity

For i from 0 to length(P)-2: // Iterate through edges (u,v) in path

u = P[i]

v = P[i+1]

bottleneckCapacity = Minimum(bottleneckCapacity, G_f.ResidualCapacity(u, v))

End For

If bottleneckCapacity <= 0:

Return 0

End If

For i from 0 to length(P)-2:

u = P[i]

v = P[i+1]

F.IncreaseFlow(u, v, bottleneckCapacity)

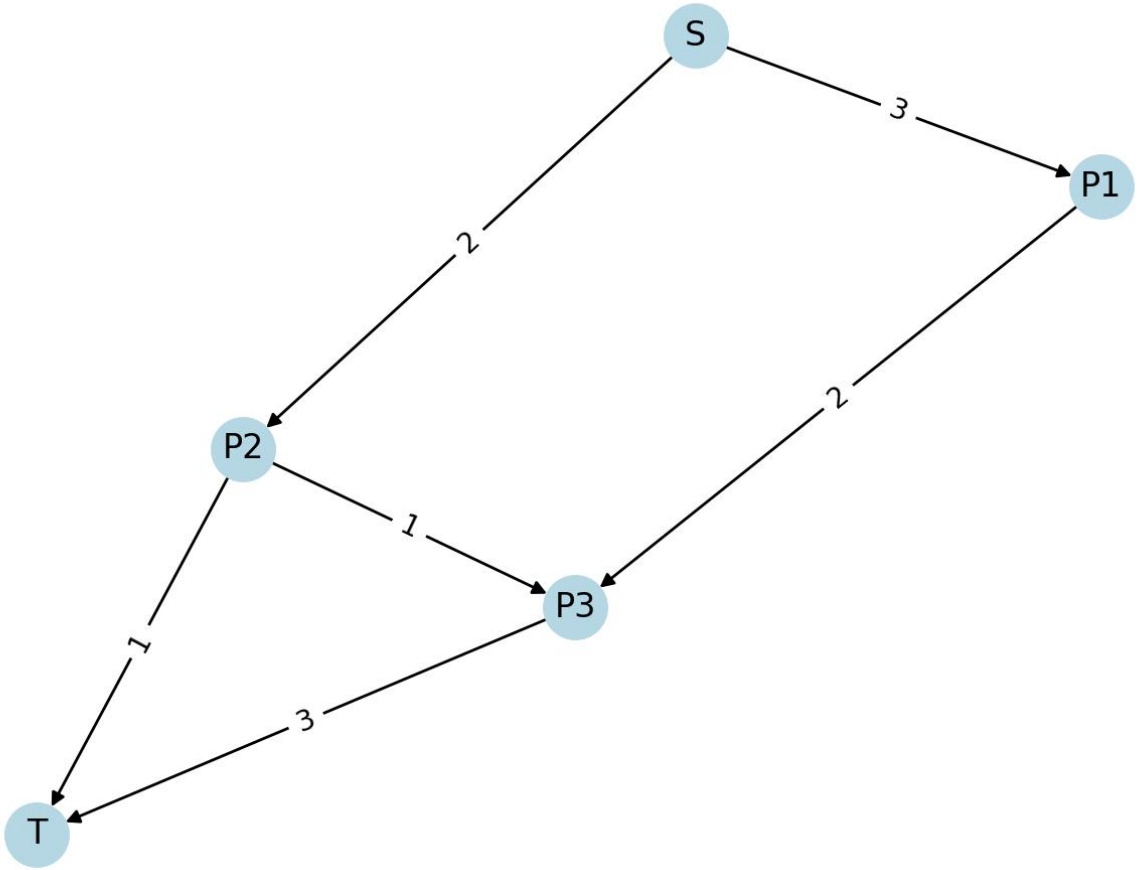
G_f.DecreaseResidualCapacity(u, v, bottleneckCapacity)

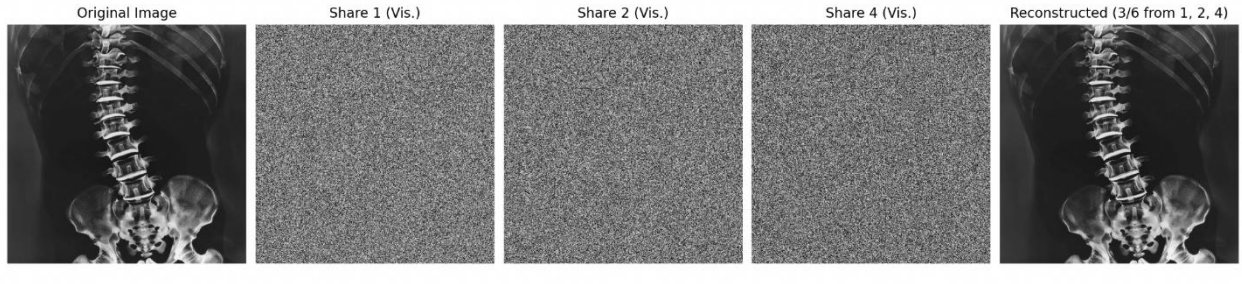
```
    //    Increase capacity on backward edge (critical for finding future paths)
    G_f.IncreaseResidualCapacity(v, u, bottleneckCapacity)
  End For
  Return bottleneckCapacity // Return the amount flow was augmented
End Function
```

Algorithm EdmondsKarpMaxFlow(Graph G_orig, Source S, Sink T)

```
  FlowGraph F = InitializeFlowGraph(G_orig, initialFlow=0)
  ResidualGraph G_f = InitializeResidualGraph(G_orig, F)
  maxFlowValue = 0
  Loop indefinitely:
    // a. Find a path in the current residual graph
    augmentingPath = FindAugmentingPathBFS(G_f, S, T)
    If augmentingPath is Null:
      Break Loop
    End If
    flowIncrease = AugmentFlowAndUpdateGraphs(augmentingPath, F, G_f)

    maxFlowValue = maxFlowValue + flowIncrease
  End Loop
  Return maxFlowValue
End Algorithm
```

Output:


S
P1
P2
P3
T
3
2
2
1
1
3


Original Image
Share 1 (Vis.)
Share 2 (Vis.)
Share 4 (Vis.)
Reconstructed (3/6 from 1, 2, 4)

```
(base) Rohiths-MacBook-Pro:DAA2 rgk$ python max.py
Starting CRT Image Secret Sharing with Max Flow Analysis...
NOTE: Requires numpy, Pillow, matplotlib, networkx.
Install with: pip install numpy Pillow matplotlib networkx
Enter the path to the image file: ./ic.png
Enter total shares (n <= 45): 6
Enter threshold (k, 2 <= k <= 6): 3

------------------------------
1. Selecting & Checking Moduli...
Selected moduli m_1...m_6: [263, 269, 271, 277, 281, 283]
  CRT Check: Prod(smallest 3) = 19172437
  CRT Check: p0 * Prod(largest 2) = 20437411
  WARNING: Asmuth-Bloom condition NOT satisfied.
Condition not met. Proceed anyway? (yes/no): yes

------------------------------
2. Preprocessing Image...
Image loaded (512x512).
Original hash: 58ac1737...

------------------------------
3. Creating CRT Shares...
Creating 6 CRT shares (k=3, p0=257, A_max=74599)...
  CRT: Processed row 51/512
  CRT: Processed row 102/512
  CRT: Processed row 153/512
  CRT: Processed row 204/512
  CRT: Processed row 255/512
  CRT: Processed row 306/512
  CRT: Processed row 357/512
  CRT: Processed row 408/512
  CRT: Processed row 459/512
  CRT: Processed row 510/512
  CRT: Processed row 512/512
CRT Share creation complete. Time: 1.19s

------------------------------
4. Max Flow Network Capacity Analysis...
  Defining EXAMPLE distribution network graph...
  Example Network: Nodes=['S', 'T', 'P1', 'P2', 'P3'], Edges=[('S', 'P1', {'capacity': 3}), ('S', 'P2', {'capacity': 2}), ('P1', 'P3', {'capacity': 2}), ('P2', 'P3', {'capacity': 1}), ('P2', 'T', {'capacity': 1}), ('P3
', 'T', {'capacity': 3})]

--- Max Flow Analysis (Source=S, Sink=T) ---
Calculated Maximum Flow Value: 4
Network capacity (4) MEETS or EXCEEDS the required flow (3).

------------------------------
5. Saving Shares...
Saving CRT shares and info to crt_shares_maxflow...
  Saved moduli information to crt_shares_maxflow/moduli_crt.txt
Shares and info saved in directory: crt_shares_maxflow

------------------------------
6. Simulating Reconstruction...
Enter 3 unique share indices (1-6, space-separated): 1 2 4
Attempting reconstruction using shares: [1, 2, 4]
  Loaded p0=257 and 6 moduli from crt_shares_maxflow/moduli_crt.txt
Loading CRT shares [1, 2, 4]...
Successfully loaded 3 shares.
Reconstructing image from 3 CRT shares (indices: [1, 2, 4])...
  CRT Recon: Processed row 51/512
  CRT Recon: Processed row 102/512
  CRT Recon: Processed row 153/512
  CRT Recon: Processed row 204/512
  CRT Recon: Processed row 255/512
  CRT Recon: Processed row 306/512
  CRT Recon: Processed row 357/512
```

```
  CRT Recon: Processed row 408/512
  CRT Recon: Processed row 459/512
  CRT Recon: Processed row 510/512
  CRT Recon: Processed row 512/512
CRT Image reconstruction complete. Time: 1.64s

------------------------------
7. Verifying Integrity...
Reconstructed hash: 58ac1737...
Original hash:      58ac1737...
Integrity Check PASSED.

------------------------------
8. Displaying Results...

Processing finished.
(base) Rohiths-MacBook-Pro:DAA2 rgk$
```

Summary of Content and Algorithms

The project aimed to design, implement, and evaluate methods for securely sharing an image using a threshold mechanism, where k out of n shares are needed for reconstruction. Fewer than k shares should reveal no information. The focus was on security and efficiency. Several algorithms and variations were explored:

1. **Shamir's Secret Sharing (SSS):**
    - **Concept:** Represents each pixel's value (secret S) as the constant term (P(0)) of a random polynomial P(x) of degree k-1 over a finite field GF(P). Shares are points (x, P(x)) on this polynomial. Reconstruction uses Lagrange Interpolation with k points.
    -
    - **Implementations:**
        - Basic (k, n) scheme using P=257 (lossless for 8-bit grayscale).
        -
        - SSS with N+1 Share: An (n+1)-th share is created by summing the first n shares (mod P). Allows reconstruction from any k of the n+1 shares. If the (n+1)-th share is used, reconstruction involves solving a system of linear equations instead of Lagrange interpolation. Versions using P=251 were explored, potentially based on specific papers, introducing minor information loss for pixel values 251-255.
        -
        - Parallel SSS: Used concurrent.futures to speed up share creation and reconstruction by processing image rows in parallel. Significant speedup observed, especially for reconstruction.
        -
        - Encrypted SSS: Added AES-GCM encryption (using cryptography library) to the numerical shares before saving, deriving the key from a user password via PBKDF2HMAC. Plaintext hash saved separately for integrity check *after* decryption.
        -
2. **Visual Cryptography (VC):**
    - **Concept:** A simpler, visual scheme. Works on binary images. Each pixel is expanded into a block (e.g., 2x2) of subpixels in two shares. For a white

pixel, both shares get the *same* pattern (half black/white subpixels); for a black pixel, they get *complementary* patterns.

-
- **Implementation:** A 2-out-of-2 scheme was implemented. Reconstruction involves simply overlaying the shares (simulated using np.minimum), where black + anything = black. Original white pixels appear gray in the reconstruction.
-
- **Algorithm:**.

3. **Chinese Remainder Theorem (CRT) - Asmuth-Bloom:**

    - **Concept:** Uses CRT to solve congruences. Requires a base modulus p0 > 255 and n pairwise coprime moduli m_i satisfying the Asmuth-Bloom condition (Product(smallest k) > p0 * Product(largest k-1)). Each secret pixel S is encoded as y = S + A*p0 (A is large random number). Shares are s_i = y mod m_i. Reconstruction involves solving for y using CRT with k shares (s_i, m_i) and then finding S = y mod p0.
    - **Implementation:** Code implemented share creation and reconstruction based on this. Includes selection of moduli from a predefined list and checking the crucial Asmuth-Bloom condition. Calculates the required range (A_max) for the random component A.
    -

4. **Max Flow Network Analysis (Conceptual):**

    - **Concept:** Model the share distribution network (servers, connections, capacities) as a graph. Use a Max Flow algorithm (like Edmonds-Karp) to determine the maximum capacity between a source and sink.
    - **Implementation:** Code added a *capacity check* using networkx. It defines an *example* network, calculates max flow, and compares it to the threshold k to see if the network *could theoretically* support delivering k shares. **Note:** This does not implement path finding or optimize the actual distribution route.

Inferences from Outputs & Analysis

- **Effectiveness:** All methods (SSS, CRT, VC) successfully split the image into shares that individually appeared random (or hid information) and allowed reconstruction when the threshold k was met (for SSS/CRT) or when overlaid

(VC). Integrity checks using hashing consistently passed for successful SSS/CRT reconstructions.

- **Entropy:** Both SSS and CRT generated shares with very high Shannon entropy, extremely close to the theoretical maximum for their respective value ranges (approx 8.00 bits for SSS with P=257; slightly **higher on average for CRT** depending on moduli). This quantitatively confirms excellent randomization and information hiding in the shares compared to the lower entropy original images.
- **Performance:**
    - Parallelism significantly speeds up SSS processing, especially reconstruction, compared to sequential execution.
    - The relative speed of SSS vs. CRT share creation depends on parameters. CRT was slightly slower for low k/n, similar for moderate k/n, and significantly faster for higher k/n (n=10, k=5) in the examples tested. This suggests a trade-off between SSS polynomial complexity (increases with k) and CRT large integer arithmetic (increases with A_max, which depends on k and moduli).
- **Asmuth-Bloom Condition:** This condition is vital for CRT. The tests showed it *failed* for some parameter choices (n=5, k=3 with the specific moduli list), highlighting the need for careful moduli selection or parameter constraints for guaranteed security/reconstruction.
- **CRT A_max:** The range for the random component A in CRT grows extremely large as k increases, demonstrating the scaling of internal numbers in the scheme.
- **VC:** Simple to understand and implement, reconstruction is trivial (overlay). However, it requires image binarization and results in contrast loss.
- **Max Flow:** The capacity check demonstrated feasibility, showing the example network could support k=3 shares (max flow was 4).

Summary of Implemented Features

- **Core Schemes:** Shamir's Secret Sharing (SSS), Asmuth-Bloom CRT, 2-out-of-2 Visual Cryptography.
- **SSS Variations:** Basic (k,n), N+1 Share method (with linear algebra reconstruction option).
- **Security Features:** Threshold property (k-out-of-n), Hashing (SHA-256) for integrity checks (shares and final image), Symmetric Encryption (AES-GCM via Fernet) for shares at rest.
- **Performance:** Parallel processing implementation for SSS using concurrent.futures. Comparative timing analysis.

- **Analysis:** Shannon entropy calculation for original images and shares. Conceptual Max Flow capacity check for distribution networks.
- **Usability:** Command-line interface for user input (n, k, passwords, indices), visual display of results using Matplotlib. Saving/loading of numerical shares (.npy/.enc), visual shares (.png), hashes (.txt), and CRT parameters (.txt).

Shortcomings and Limitations

- **P=251 in SSS:** Some versions used P=251, causing minor, irreversible data loss for the brightest pixels (251-255). Using P>=257 avoids this.
- **CRT Moduli Selection:** Relies on a predefined list of primes. Does not dynamically generate primes or guarantee the Asmuth-Bloom condition will hold for all user-chosen k and n values. The check warns but allows proceeding if the condition fails.
- **VC Implementation:** Limited to the basic 2-out-of-2 scheme for binary images; reconstruction suffers contrast loss.
- **Max Flow Implementation:** Only a conceptual capacity check using a hardcoded example network was implemented. It doesn't model real network complexities or optimize actual share distribution paths.
- **Encryption:** Uses symmetric encryption relying on password security. No advanced key management or asymmetric options were explored. Password handling via getpass is basic.
- **Parallelism Overhead:** For very small images or specific parameters, the overhead of parallel process management might potentially make it slower than sequential execution (though speedups were generally observed for larger images/reconstruction).

Future Enhancements

- **Guaranteed Lossless SSS:** Enforce the use of P>=257 in all SSS implementations.
- **Robust CRT Parameterization:** Implement dynamic prime number generation and selection methods that *guarantee* the Asmuth-Bloom condition is met for the user's chosen k and n, or provide clearer guidance on valid parameter ranges.
- **Advanced VC Schemes:** Explore and implement k-out-of-n VC or schemes that handle grayscale/color images (e.g., using error diffusion or more complex pixel expansion).

- **Meaningful Max Flow Integration:** Allow users to define their network topology and capacities. Implement algorithms based on max flow for finding disjoint paths or optimizing distribution based on user-defined criteria (cost, reliability).
- **User Interface:** Develop a Graphical User Interface (GUI) for ease of use.
- **Asymmetric Encryption:** Incorporate public-key cryptography for more flexible and secure key management during share distribution.
- **Error Correction:** Integrate error-correcting codes (ECC) with secret sharing to handle not just missing shares but also corrupted/tampered shares up to a certain limit.
- **Hybrid Approaches:** Combine techniques, e.g., use VC for a quick preview and SSS/CRT for the secure numerical data.

Final Conclusion

This project successfully implemented and compared multiple threshold secret sharing schemes (SSS, CRT, VC) for secure image distribution. Key variations including the N+1 share method, parallelism for efficiency, and encryption for share security were demonstrated. Analysis using hashing confirmed integrity, while entropy calculations verified the essential randomness of generated shares for both SSS and CRT. A conceptual check using Max Flow was integrated to analyze network capacity for distribution.

The comparison highlights the trade-offs: SSS offers simpler setup, while CRT requires careful parameter validation (Asmuth-Bloom condition) but showed potential performance advantages in some scenarios (higher k/n). VC provides visual simplicity but is limited in application and fidelity. The addition of encryption provides crucial practical security for stored shares. While the core goals were met, future work could focus on more robust parameter selection for CRT, advanced VC schemes, and a more practical implementation of network analysis for optimized share distribution. Overall, the project provides a solid foundation and comparison of techniques for secure threshold image sharing.