ENCRYPTION AND DECRYPTION OF COMPRESSED IMAGES

6.857 FINAL PROJECT

Shiloh Curtis
shilohc@mit.edu

Ray Dedhia
rdedhia@mit.edu

Murielle Dunand
mdunand@mit.edu

ABSTRACT

Online image uploads have become ubiquitous through sites such as Patreon, Dropbox and Google Drive. As these become more common, users may want to encrypt their images before uploading the encrypted version to such sharing sites. However, most of these platforms also compress the image to allow for faster uploads, and this compression is known to be lossy. When decrypted, a lossy image could be entirely unrecognizable. Therefore, this work analyzes a current encryption scheme that is robust against noise and crop attacks, and therefore may be robust against compression. We find that it is most effective for smaller images, but further research is needed for larger and more compressed images.

1 Introduction

As technology advances, the security of image sharing has become more and more important. With images being widely available online, it may be necessary for these images to be encrypted, or only available to users with specific credentials. For example, sites like Patreon or Dropbox make images available to many users, but the uploader may want to encrypt their image so that only a few of those users can see the original image – paid subscribers, for example. In the case of Google drive, a user may want to encrypt images they upload so that the host cannot see the content of their image. The challenge that is presented with encrypting uploaded images is that most image hosting platforms employ lossy compression. Compression makes an image smaller and easier to load at the cost of image information. If the compression technique is "lossy", pixel information is lost in the compression process, and it standard for the JPEG image file type. Compressing an encrypted image could result in a corrupted decrypted image due to that information loss. In this work, we implement a published image compression technique that may be robust to compression, and evaluate its performance on different image sizes and levels of compression. We find that it is most effective on small images and that compression still loses out on image information, and more tests are required to find an encryption algorithm that is robust to compression.

2 Related Works

In this section, we will discuss other existing image encryption schemes. For example, Soleymani et al. recommend a public-key image encryption scheme using elliptic curves in their paper [1]. They argue that it is fast and resource efficient, and is thus useful for real-time applications such as image encryption. This was not implemented in the current work but can be useful for future works.

There also exists research on Encryption then Compression (EtC) systems, in which images are encrypted, compressed, (in some cases) decompressed, and then decrypted. For example, Kumar et al. present a EtC scheme uses DWT, singular value decomposition, and Huffman coding [2]. As part of EtC systems, image compression and decompression algorithms that work with different image encryption algorithms have been developed. For example, Schonberg et al. present a scheme to compress encrypted images based on linear error correcting codes [3]. Sathyalakshmi et al. propose a scheme for compressing and then decompressing AES encrypted images using discrete cosine transform (DCT) and discrete wavelet transform (DWT) compression [4].

Research has also been done on image encryption schemes that robust against existing image compression algorithms. For example, Chuman et al. propose a block scrambling-based encryption scheme that uses JPEG compression and decompression [5]. Ozturk and Sogukpinar analyze and compare two image encryption algorithms – Mirror-like image encryption and Visual Cryptography – with and without PNG and JPEG compression [6]. Wang et al. propose a low-cost encryption scheme using rate-distortion optimization where encrypted images can be accurately decrypted even after their lossless compression by a third party [7].

Related to our research project is the topic of simultaneous image encryption and compression. One such algorithm, which uses using compressed sensing (CS), structurally random matrix (SRM), and permutation-diffusion type image encryption, is described and recommended by Zhang et al. [8].

Also related to our research project is the topic of partial image encryption, which uses format-preserving encryption to encrypt only parts of an image, while leaving others unencrypted. Jang et al. describe a scheme for partial image encryption using format-preserving encryption schemes FF1 and FF3-1, which does not increase the size of the data [9].

3 Algorithm

In this project, we implemented and evaluated the encryption algorithm given in [10]. For convenience, we will also briefly summarize it here.

First, the algorithm generates three random messages m_1, m_2, and m_3, each of which is a 53-byte string. These are encrypted using RSA to yield the ciphertexts c_1, c_2, and c_3 [11]. The ciphertexts and messages are then used to initialize a quantum logistic map, which provides the keystream that is reversibly applied to the image, and the encrypted image is returned along with the ciphertexts. Since the private key is required to decrypt the ciphertexts, only the holder of the private key can regenerate the keystream and apply it to the encrypted image to recreate the original, assuming the keystream is truly pseudorandom.

The quantum logistic map is specified in terms of sequences x_i, y_i, and z_i, with

$$\begin{aligned}
x_0 &= \frac{1}{[m_1 - c_1 \mod (M+1)(N+1)] + r} \\
y_0 &= \frac{1}{[m_2 - c_2 \mod (M+1)(N+1)] + r} \\
z_0 &= \frac{1}{[m_3 - c_3 \mod (M+1)(N+1)] + r}
\end{aligned}$$

where

$$r = \sum_{i=1}^{M} \sum_{j=1}^{N} \sqrt[5]{(P(i,j) + i + j)^2},$$

the image dimensions are $M \times N$, and $P(i,j)$ is the color value at pixel (i,j) in the image. These are then updated according to the quantum logistic map:

$$\begin{aligned}
x_{i+1} &= r(x_i - |x_i|^2) - r y_i \\
y_{i+1} &= -y_i e^{-2\beta} + e^{-\beta} r \left((2 - x_i - x_i^*) y_i - x_i z_i^* - x_i^* z_i\right) \\
z_{i+1} &= -z_i e^{-2\beta} + e^{-\beta} r \left(2(1 - x_i^*) z_i - 2 x_i y_i - x_i\right)
\end{aligned}$$

where x_i^* is the complex conjugate of x_i and r and β are the logistic map parameters. We use the parameters $\beta = 6$ and $r = 3.99$, as given in [10].

Each channel of the image is subjected to five encryption rounds. Each round first permutes the rows and columns of the image matrix, performs a discrete cosine transform, permutes the rows and columns some more, then performs an inverse discrete cosine transform. The sequences x_i, y_i, and z_i are used to determine the row and column permutations; the full details are given in [10] and will not be repeated here.

4 Implementation

Since a reference implementation is not provided by [10], we have written one in Python.

4.1 Structure and Dependencies

Our implementation is written in Python; its primary dependencies are OpenCV [12] for working with images as matrices, Numpy [13] for performing mathematical operations on those image matrices, and rsa [14] and secrets [15] for cryptographic operations such as loading RSA keys and generating and encrypting the messages m_1, m_2, and m_3.

An image is split into three channels, each of which undergo five encryption rounds; the functions `enc` and `dec` respectively call `enc_channel` and `dec_channel` to encrypt or decrypt each channel of the image. Since encryption and decryption rounds are done in much the same way, both `enc_channel` and `dec_channel` use `encryption_round` to perform a single encryption round.

We implement the necessary row/column permutations and discrete cosine transforms using helper functions `permute_rows`, `permute_cols`, `dct`, and `inverse_dct`. The quantum logistic map is implemented by `update_xyz`, which takes in values for x_i, y_i, and z_i (in addition to the logistic map parameters) and returns $(x_{i+1}, y_{i+1}, z_{i+1})$.

The source code for our implementation is given in Appendix A.

4.2 Challenges

Our first challenge was that, due to the structure of the quantum logistic map, x_i, y_i, and z_i increase exponentially in size. Using the quantum logistic map equations verbatim from [10] caused (x, y, z) to overflow all integer representations we could find. Therefore, our implementation performs all arithmetic operations modulo $(M + 1) \cdot (N + 1)$, where M and N are the image dimensions.

We also experienced many issues with efficiency, mostly related to the discrete cosine transform (DCT). DCT inherently has time complexity $O(M^2 \cdot N^2)$, so even though we used Numpy optimizations where possible in our implementation, it is slow even for quite moderately sized images. Unfortunately, the encryption algorithm performs DCTs frequently: each encryption round requires one DCT and one inverse DCT, and a three-channel image will require fifteen encryption rounds. After Numpy optimization, a 96 × 72-pixel image took 40 seconds to encrypt, and a 162 × 208 image took 13 minutes.

5 Methods

We used two test images, shown in Table 1: a color photo of a flower [16], and a grayscale image with text [17].

(a) Color test image

Cherry-picked
results look great
(b) Grayscale test image

Table 1: The color and grayscale test images used in testing.

These images were resized to several different scales, in order to test encryption performance. We tested decryption of non-compressed images (which always worked perfectly), as well as decryption of images that had been subjected to various degrees of compression while encrypted. Since most encryption algorithms used by commercial platforms such as Instagram are proprietary, we used ImageMagick [18] to compress our images. This also allowed us to experiment with a variety of compression parameters.

The base ImageMagick command used to compress images was:

```
> convert input.jpg \
    -sampling-factor 4:2:0 \
    -strip \
    -quality 85 \
    -interlace Plane \
    -gaussian-blur 0.05 \
    -colorspace RGB \
    output.jpg
```

The main parameters we modified were the quality and Gaussian blur (sampling is only applied if quality is less than 90 and had little visually apparent impact in any case). According to the JPEG image format, "quality" is a value between 1 and 100 inclusive, but the ratio of input image size to output image size varies nonlinearly with quality. Nevertheless, quality is usually specified as a percentage. By default, ImageMagick uses 92% quality for JPEG images [19].

The Gaussian blur parameter refers to the σ value used in the blur equation

$$G(u, v) = \frac{1}{2\pi\sigma^2} e^{-(u^2+v^2)/(2\sigma^2)}.$$

A higher value for σ corresponds to an increased amount of blur [20].

Lastly, when testing our implementation, we used the same 1024-bit length RSA keys, which can be found with on GitHub with our source code at `https://github.com/shilohc/6857-proj`. We generated the keys with openssl [21] using the following commands:

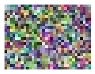
```
> openssl genrsa -out private.pem 1024
> openssl rsa -in private.pem -pubout -outform PEM -out public.pem
```

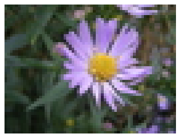
6 Results

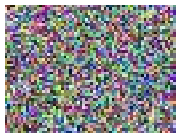
6.1 Large image sizes

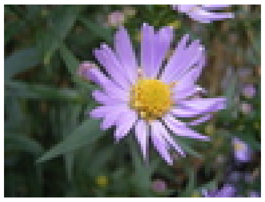
Because the algorithm had a long runtime, we started with very small images before gradually moving larger. This progression showed that results varied greatly between even small size differences of the same image.

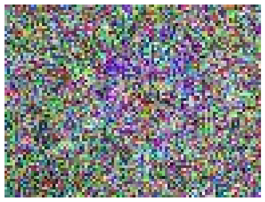
(a) 32 × 24 image

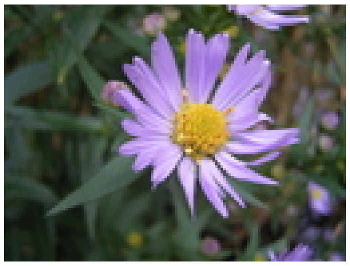

(b) 32 × 24 encrypted


(c) 64 × 48 image


(d) 64 × 48 encrypted


(e) 96 × 72 image


(f) 96 × 72 encrypted


(g) 128 × 96 image

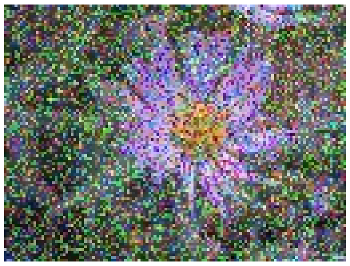
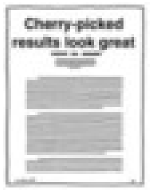
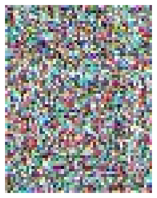

(h) 128 × 96 encrypted

Table 2: A comparison of the original and encrypted versions of various sizes of the same color image.

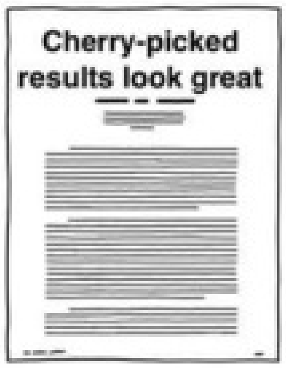
In Table 2, we see that when an image gets larger than 96 × 72 pixels, information begins to get leaked to the naked eye. To confirm these findings, we also tested on grayscale images, shown in Table 3.



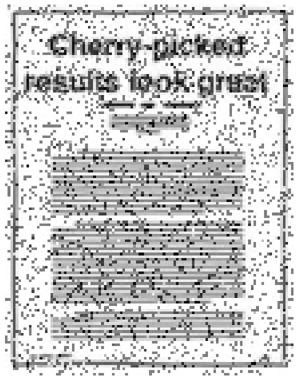
(a) 55 × 70 image



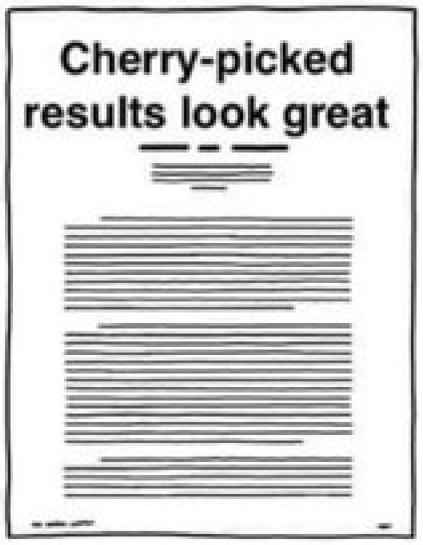
(b) 55 × 70 encrypted



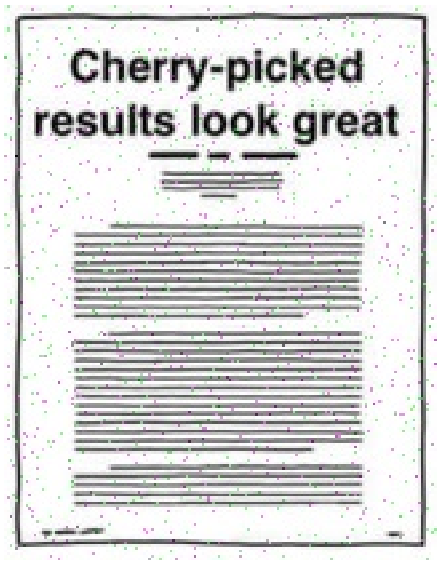
(c) 108 × 139 image



(d) 108 × 139 encrypted



(e) 162 × 208 image



(f) 162 × 208 encrypted

Table 3: A comparison of the original and encrypted versions of various sizes of the same gray-scale image.

As we can see in both Table 2 and Table 3, as the size of the image increases the decryption becomes ineffective and completely clear to the naked eye. This is consistent with the examples in the original paper, as they were all quite small, but not useful for a general purpose image encryption scheme.

6.2 Compression of encrypted images

To check how the algorithm performs under compression, we used ImageMagick, a set of commands that allows for compression from the command line. We used this tool also because it allowed us to change the parameters of the compression, and so we were able to vary the blur and compression level. The results on a color image are shown in Table 4.

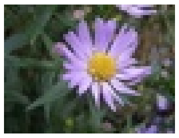
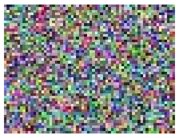
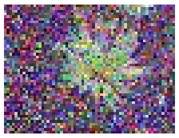
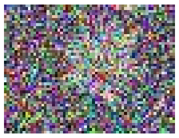
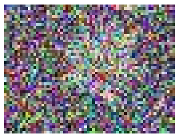
	Encrypted, compressed images	**Decrypted images**
No compression		
Compression quality 95%, no blur		
Compression quality 85%, Gaussian blur 0.05		
No compression, Gaussian blur 0.05		

Table 4: A comparison of the encrypted and decrypted images using different amounts of compression and blur.

Even with compression quality 95%, the image is nearly unrecognizable to the naked eye. This means that the image would not work as a key and only work to transfer information to humans in a few cases. Even with small images, the algorithm takes a very long time to run and returns noisy results.

7 Comparison with other algorithms

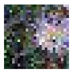
In this section, we discuss other encrpytion methods that may be effective with compression. These methods were already implemented by their creators, and so we tested on the existing implementations.

7.1 Image Encryption using Hénon Chaotic Map

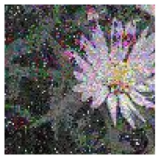
A chaotic map is a map that exhibits chaotic, or seemingly random, behavior. Chaotic maps are simple and fast functions, making chaos-based image encryption algorithms fast enough for real-time applications [22]. Cryptographic algorithms and chaotic maps have similar properties, such as their sensitivity to changes in their initial conditions and their pseudorandom behavior. As a result, it makes sense to develop chaotic based cryptographic algorithms for secure communication and cryptography.

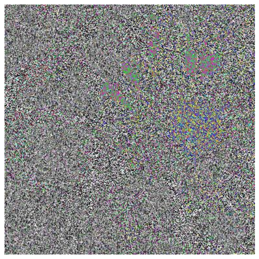
Image encryption using chaos is based on the ability of some dynamic systems to produce sequences of numbers that are random in nature. Messages are then encrypted using these sequences. Because of the pseudorandom behavior of chaotic maps, the output of the system will appear random to any attackers. However, the receiver will still be able to decrypt it using the system's initial parameters, i.e. the secret symmetric key. An important difference between cryptography and chaos maps is that encryption transformations are defined on finite sets whereas chaos maps have meaning only for real numbers. Each chaos map has parameters that are equivalent to encryption key in cryptography [23].

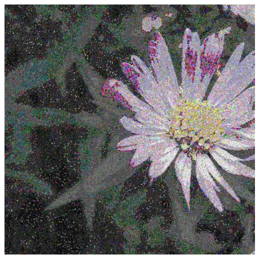
(a) 32×24 encrypted

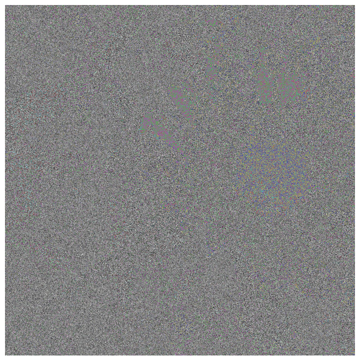

(b) 32×24 decrypted

(c) 128×96 encrypted


(d) 128×96 decrypted


(e) 512×384 encrypted


(f) 512×384 decrypted


(g) 2048×1536 encrypted

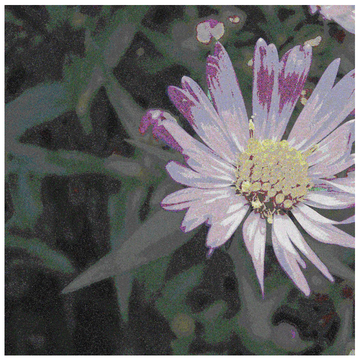

(h) 2048×1536 decrypted

Table 5: Images of different sizes were encrypted using a chaos Hénon map and then saved as JPEG files. The encrypted images were then extracted from the JPEG files and decrypted using the chaos Hénon map.

To compare chaotic image encryption with the public-key image encryption algorithm we implemented, we tested an open source implementation of chaotic image encryption that uses Hénon chaos maps [24]. We modified the code to handle color images, and tested it with different amounts of compression.

We encrypted images of different sizes, saved them as JPEG images, and then decrypted them. The results can be found in Table 5. Note that there is some loss when decrypting the JPEG versions of the encrypted images. This loss is not present when decrypting from a BMP version of the encrypted images.

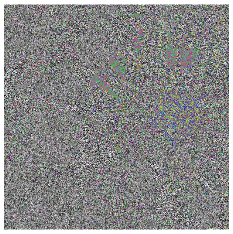
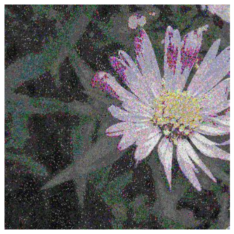
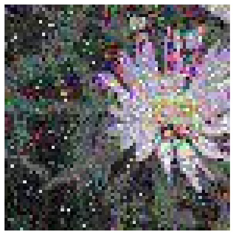
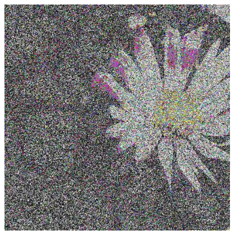
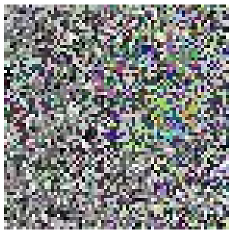
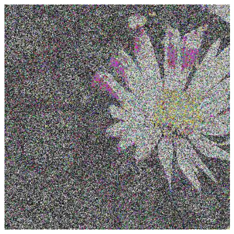
	Encrypted, compressed images	**Decrypted images**
No compression		
Compression quality 95%, no blur		
Compression quality 85%, Gaussian blur 0.05		
No compression, Gaussian blur 0.05		

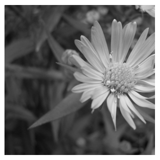
Table 6: We encrypted images with a chaos Hénon map, saved them as JPEG images, applied different amounts of compression and blur on them, and then decrypted the images using the chaos Hénon map.

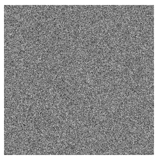
The results of the tests we ran on images with different amounts of compression can be found in Table 6. Note that this algorithm handles compression slightly better than the asymmetric image encryption algorithm we implemented for this paper. However, even low amounts of compression still quickly make it difficult to recognize the decrypted image as a flower.

Overall, the chaos encryption algorithm was much faster than the one we implemented, and it handled compression slightly better. We ran the tests using a symmetric secret key of $(0.1, 0.1)$. From some rudimentary testing, we found that changes to the secret key as small as $5e-16$ cause the decryption algorithm to output random noise. We also found that the secret key values can be any real number in the range $[-2.2, 1]$, inclusive. (Note that the lower bound may be slightly lower, and may differ with the size of the image.) Since the algorithm is so sensitive to noise in its initial conditions (i.e. the secret key), there are over $(\frac{3.2}{5e-16}+1) \times (\frac{3.2}{5e-16}+1) = 4e31$ possible secret keys that an attacker would have to test to break the encryption. Thus, even if you could test each key in one nanosecond, it would still take over one trillion centuries to test all possible keys.

7.2 Image Encryption using Chaotic Logistic Map

We also tested an open source implementation of chaotic image encryption using a chaotic logistic maps [25]. We encrypted a test image, saved it as a JPEG file, and compressed it with 85% quality and 0.05 blur. We then decrypted the original encrypted array, the encrypted image loaded from the JPEG file, and the encrypted image after compression and blurring. The results can be found in Table 7.


(a) no compression


(b) JPEG conversion

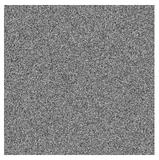
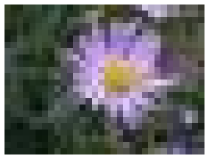

(c) 85% quality compression, 0.05 blur

Table 7: A 512×512 grayscale image was encrypted using chaotic image encryption with a logistic map. The table above contains the decrypted uncompressed image (fig 7a), the decrypted image after JPEG conversion (fig 7b), and the decrypted image after 85% quality compression and 0.05 blur (fig 7c).

Unlike with the implementation using Hénon maps, compressing an encrypted image by any amount, even just by storing the encrypted array as a JPEG image, rendered the decrypted image completely indecipherable, and visually indistinguishable from random noise.

7.3 Homomorphic Image Encryption

Homomorphic encryption is a form of encryption that allows users to perform computations on the encrypted data without having to decrypt it first. We tested an open source implementation of an image encryption scheme that extends Paillier's Homomorphic Encryption (PHE) scheme to operate over images [26].


(a) 32×24, no noise, decrypted

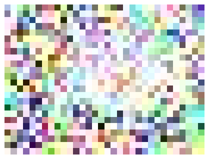

(b) 32×24, some noise, decrypted

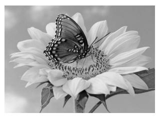
Table 8: A small image is encrypted using PHE. It is then decrypted before and after noise is added to it.

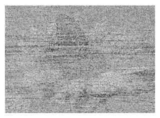
The encryption algorithm outputs an encrypted array of extremely large integers in the range $(2e150, 6e153)$, which cannot be converted into an image, so instead of trying to compress the encrypted array, we tried adding some random noise to it, using the code `numpy.random.choice([-1, 0, 1], p=[0.1, 0.8, 0.1], size=cipher.shape)` to generate the noise. However, when even this tiny amount of noise was added to the encrypted array, the decrypted image immediately became indecipherable, resembling random noise. See Table 8 for a comparison between the decryption of the unedited encrypted array, which is identical to the input image, and the decryption of the noisy encrypted array.

7.4 Image Encryption Algorithm Based on Rubik's Cube

Lastly, we tested an open source implementation of an image encryption algorithm based on the Rubik's Cube [27]. The algorithm scrambles the original image using the principle of Rubik's cube and applies the XOR operator to the rows and columns of the scrambled image using two secret keys. The original image is scrambled using the principle of Rubik's cube. According to the author of the code, experimental results and analysis have shown that the scheme achieves good encryption and can resist multiple forms of attack.

We used the code to encrypt a test image, then saved it as a JPEG file and compressed it with 85% quality and 0.05 blur. We then decrypted the original encrypted array, the encrypted image loaded from the JPEG file, and the encrypted image after compression and blurring.


(a) no compression


(b) JPEG conversion

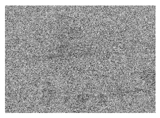

(c) 85% quality compression, 0.05 blur

Table 9: A 453×512 grayscale image was encrypted using an image encryption scheme based on Rubik's Cube. The table above contains the decrypted uncompressed image (fig 9a), the decrypted image after JPEG conversion (fig 9b), and the decrypted image after 85% quality compression and 0.05 blur (fig 9c).

The results can be found in Table 9. As you can see, the decrypted image after JPEG conversion is almost completely indecipherable. In addition, after JPEG conversion, compression with 85% quality, and 0.05 blurring, the decrypted image is indistinguishable from random noise.

8 Conclusions

Overall, we found that our implementation of the method shown by Jiao et al. is not very effective at hiding compressed images. As pictures got larger, the encrypted images became less and less hidden by inspection. This is consistent with the results in the original paper, which only used very small images. However, this shows that this method is not very effective for more typically sized images. It is possible that we might get better results using keys with a larger bit length, as we were using RSA keys with a bit length of 1024 for our tests. However, using larger keys significantly increases the computational cost of the algorithm, and consequently, we weren't able to test this.

Secondly, while the method claims to be effective against crop attacks and noisy images, it is not effective with compressed images. With even a 5 percent compression, the decrypted image is heavily corrupted, and blur has a strong effect on the decrypted image as well. It seems that the blur effect is disproportionate and requires a scheme that is specifically robust to blurring. However, this may mean that the scheme would not be entirely secure, since increased robustness increases the chance of collisions. Overall, the tested cryptographic method is not very effective for the encryption, and we would need to look for a scheme that is more specifically robust against blurring. However, other methods show promise and we believe that a method to return compressed and decrypted images is in the future.

9 Future Work

Given our results, there are plenty of avenues for future work. First, there are many other possible encryption scheme that could be effective for compression. Our next step would be to test out other published methods: since compression tends to involve blurring operations, it seems that methods designed to be robust to blurring might be more effective. As noted in Section 4.2, the algorithm we studied was slow even on quite small images; since performance is important for practical applications, it would be useful to compare encryption/decryption speeds against other encryption schemes.

Another option would be to try another compression type that does not rely as much on blurring. This includes "lossless" image formats, which by definition has no loss of image information. We would expect that a lossless format would decrypt successfully, but may not be compressible; characterizing compressibility of lossless encrypted images would also be useful for practical applications. Overall, this is a broad and useful topic with many options for expansion.

A Source Code

The full source code and test images are available on GitHub at `https://github.com/shilohc/6857-proj`. The encryption and decryption algorithms are implemented by `encrypt.py`:

```python
import cv2
import numpy as np
import rsa
import secrets
import sys
import struct
import time

def update_xyz(xyz, mod_n, r=3.99, beta=6):
    """
    Updates x, y, z according to the quantum logistic map.  Default values of
    the parameters beta and r are given on page 6 of the paper; the quantum
    logistic map is given in equations 1-3. Note that x, y, z are complex
    numbers.
    Calculates all values mod (N+1)(M+1) (where M and N are the dimensions of
    the input image).
    """
    x, y, z = xyz
    x_new = mod(r * (x - abs(x)**2) - r * y, mod_n)
    y_new = mod(-y * np.exp(-2*beta) + np.exp(-beta) * r * ((2 - x - np.conjugate(x)) * y - x *
    ↪  np.conjugate(z) - np.conjugate(x) * z), mod_n)
    z_new = mod(-z * np.exp(-2*beta) + np.exp(-beta) * r * (2 * (1 - np.conjugate(x)) * z - 2 * x *
    ↪  y - x), mod_n)
    return x_new, y_new, z_new

def dct(img):
    """
    Computes the discrete cosine transform of the array img.  Assumes that img
    is grayscale (single-channel).
    """
    M, N = img.shape
    out = np.zeros((M,N))
    i_array = np.array([[i for j in range(N)] for i in range(M)])
    j_array = np.array([[j for j in range(N)] for i in range(M)])
    sigma_array = np.array([[sigma(i, M) * sigma(j, N) for j in range(N)] for i in range(M)])
    for u in range(M):
        for v in range(N):
            cos_array = np.cos(u * (2*i_array + 1) * np.pi)/(2*M) * \
                        np.cos(v * (2*j_array + 1) * np.pi)/(2*N)
            out[u][v] = np.sum(img * cos_array)
    return out * sigma_array

def inverse_dct(img):
    """
    Computes the inverse discrete cosine transform of the array img.  Assumes
    that img is grayscale (single-channel).
    """
    M, N = img.shape
    out = np.zeros((M,N), dtype=np.float32)
    u_array = np.array([[u for v in range(N)] for u in range(M)])
    v_array = np.array([[v for v in range(N)] for u in range(M)])
    sigma_array = np.array([[sigma(u, M) * sigma(v, N) for v in range(N)] for u in range(M)])
    for i in range(M):
        for j in range(N):
            cos_array = np.cos(u_array * (2*i + 1) * np.pi)/(2*M) * \
                        np.cos(v_array * (2*j + 1) * np.pi)/(2*N)
            out[i][j] = np.sum(sigma_array * cos_array)
    return img * out

def permute_rows(img, primes):
    M, N = img.shape
    out = np.zeros((M, N))
    for i in range(M):
        for j in range(N):
            out[i][j] = img[i][int((j + primes[i] - 1) % N)]
    return out

def permute_cols(img, primes):
    M, N = img.shape
    out = np.zeros((M, N))
    for i in range(M):
        for j in range(N):
            out[i][j] = img[int((i + primes[j] - 1) % M)][j]
    return out

def encryption_round(X_rk, xyz_prev, mod_n, r, verbose=False):
    M, N = X_rk.shape

    # xyzs = [xyz_{500+rk(MN)}, xyz_{500+rk(MN)+1}, ..., xyz_{500+rk(MN)+MN}
    xyzs = [list(xyz_prev)]
    for i in range(M*N):
        xyzs.append(update_xyz(xyzs[-1], mod_n, r))

    # Calculate x_k', y_k', z_k', w_k', and s_k
    xk_primes = [(np.floor(xyzs[k+1][0] * 1e14)) % (N+1) for k in range(M)]
    yk_primes = [(np.floor(xyzs[k+1][1] * 1e14)) % (M+1) for k in range(N)]
    zk_primes = [(np.floor((xyzs[k+1][2] * 0.6 + xyzs[k+1][0] * 0.4) * 1e14)) \
            % (N+1) for k in range(M)]
    wk_primes = [(np.floor((xyzs[k+1][2] * 0.6 + xyzs[k+1][1] * 0.4) * 1e14)) \
            % (M+1) for k in range(N)]
    sk = np.array([[np.fix(sum(xyzs[M*j+i+1]) * 1e14) % 256 for j in range(N)] \
            for i in range(M)], dtype=np.int64)

    if verbose:
        print("Starting row and column permutations...")
    Xrk_permuted = permute_cols(permute_rows(X_rk, xk_primes), yk_primes)
    if verbose:
        print("Starting DCT coefficient matrix...")
    F = dct(Xrk_permuted)
    if verbose:
        print("Starting row and column permutations...")
    F_permuted = permute_cols(permute_rows(F, zk_primes), wk_primes)
    if verbose:
        print("Starting inverse DCT coefficient matrix...")
    G = inverse_dct(F_permuted)
    if verbose:
        print("Generating encrypted image...")
    xor_values = np.bitwise_xor(np.bitwise_xor(G.astype('int64'), sk),
            X_rk.astype('int64'))
    return xor_values, xyzs[-1]

def gen_ciphertexts(pkb):
    """
    Randomly select three messages, encoded as bytes, then
    use the messages and the public key to generate three
    ciphertexts. The messages can be at most (length of public key in bytes - 11) bytes long.
    Returns the messages and ciphertexts.
    """
    messages = [secrets.token_bytes(53) for i in range(3)]
    return messages, [rsa.encrypt(m, pkb) for m in messages]

def read_keys(public_filename, secret_filename):
    """
    takes any two public and secret key files
    returns rsa.key.PublicKey and rsa.key.PrivateKey types
    easily converted to string if need be
    """

    with open(public_filename, mode='rb') as public_file:
        key_data = public_file.read()
        public_key = rsa.PublicKey.load_pkcs1_openssl_pem(key_data)

    with open(secret_filename, mode='rb') as secret_file:
        key_data = secret_file.read()
        secret_key = rsa.PrivateKey.load_pkcs1(key_data)

    return public_key, secret_key

def sigma(x, L):
    """
    Helper function called by dct and inverse_dct.
    """
    if x==0:
        return np.sqrt(1/L)
    return np.sqrt(2/L)

def mod(a, n):
    """
    Returns `a mod n`, where `a` can be a complex or real number.
    """
    # If a is not complex
    if not isinstance(a, complex):
        return a % n

    return complex(a.real % n, a.imag % n)

def enc(img, pkb, verbose=False):
    """ Calls enc_channel on each channel in the image. """
    if verbose:
        print("Starting encryption...")

    image_dims = img.shape
    # If image has one channel (grayscale)
    if len(image_dims)==2:
        return enc_channel(img, pkb)

    # Else if image has 3 or 4 channels (RGB or RGBA)
    if len(image_dims)==3 and (image_dims[2]==3 or image_dims[2]==4):
        cipher, r, enc_img = ([], [], [])
        for channel in range(image_dims[2]):
            cipher_channel, r_channel, enc_img_channel =
                enc_channel(img[:,:,channel], pkb, verbose=verbose)
            cipher.append(cipher_channel)
            r.append(r_channel)
            enc_img.append(enc_img_channel.T) # (rows, cols) => (cols, rows)
        enc_img = np.array(enc_img).T # (channels, cols, rows) => (rows, cols, channels)
        return (cipher, r, enc_img)

    # Else invalid image
    return (None, None, None)

def enc_channel(img, pkb, verbose=False):
    """
    Takes in one channel of the plain image and the public key,
    and returns the ciphertexts, r, and the encrypted image.
    """

    if verbose:
        print("Encrypting new channel...")
    # Calculate r
    M, N = img.shape
    r = np.sum((np.concatenate(list(np.arange(j,N+j) for j in range(M))) + img.flatten())**(2/5))
    if verbose:
        print("Calculated r...")
    mod_n = (M+1)*(N+1) if (M+1)*(N+1)>256 else (M+1)*(N+1)*256

    # Calculate m, c
    m, c = gen_ciphertexts(pkb)
    m_int = [int.from_bytes(m_i, sys.byteorder) for m_i in m]
    c_int = [int.from_bytes(c_i, sys.byteorder) for c_i in c]
    if verbose:
        print("Calculated m, c...")

    # Calculate xyz_500
    xyz = [mod(1/(abs(m_int[i] - c_int[i]) + r), mod_n) for i in range(3)]
    for i in range(500):
        xyz = update_xyz(xyz, mod_n, r)

    # X_rk holds the value of the image at start of round rk; X_0 = plain image
    X_rk = np.array(img, dtype=np.float32)
    # xyz_prev holds the value xyz_{500 + rk(MN)} (xyz_500 before rk loop)
    xyz_prev = xyz

    for rk in range(5):
        if verbose:
            print("In encryption round {}...".format(rk))
        X_rk, xyz_prev = encryption_round(X_rk, xyz_prev, mod_n, r, verbose)

    # Output ciphertexts, r, and encrypted image
    return (c, r, X_rk)

def dec(img, ciphertexts, r, pkb, skb, verbose=False):
    """ Calls dec_channel on each channel in the img. """
    if verbose:
        print("Starting encryption...")

    image_dims = img.shape
    # If image has one channel (grayscale)
    if len(image_dims)==2:
        return dec_channel(img, ciphertexts, r, pkb, skb)

    # Else if image has 3 or 4 channels (RGB or RGBA)
    if len(image_dims)==3 and (image_dims[2]==3 or image_dims[2]==4):
        dec_img = []
        for channel in range(image_dims[2]):
            dec_img_channel = dec_channel(img[:,:,channel], ciphertexts[channel], r[channel],
                                          pkb, skb, verbose=verbose)
            dec_img.append(dec_img_channel.T) # (rows, cols) => (cols, rows)
        dec_img = np.array(dec_img).T # (channels, cols, rows) => (rows, cols, channels)
        return dec_img

    # Else invalid image
    return (None, None, None)

def dec_channel(img, ciphertexts, r, pkb, skb, verbose=False):
    if verbose:
        print("Encrypting new channel...")
    # Calculate r
    M, N = img.shape
    if verbose:
        print("Calculated r...")
    mod_n = (M+1)*(N+1) if (M+1)*(N+1)>256 else (M+1)*(N+1)*256

    # Calculate m, c
    c = ciphertexts
    m = [rsa.decrypt(i, skb) for i in c]
    m_int = [int.from_bytes(m_i, sys.byteorder) for m_i in m]
    c_int = [int.from_bytes(c_i, sys.byteorder) for c_i in c]
    if verbose:
        print("Calculated m, c...")

    # Calculate xyz_500
    xyz = [mod(1/(abs(m_int[i] - c_int[i]) + r), mod_n) for i in range(3)]
    for i in range(500):
        xyz = update_xyz(xyz, mod_n, r)

    # C_rk is the encrypted image at start of round rk
    # C_0 = original encrypted image
    C_rk = np.array(img, dtype=np.float32)
    # xyz_prev holds the value xyz_{500 + rk(MN)} (xyz_500 before rk loop)
    xyz_prev = xyz

    for rk in range(5):
        if verbose:
            print("In decryption round {}...".format(rk))
        C_rk, xyz_prev = encryption_round(C_rk, xyz_prev, mod_n, r, verbose)

    # Output decrypted image
    return C_rk

if __name__ == "__main__":
    img_filename = "testimage1_64x48.jpg"
    print("Starting...")
    start = time.time()
    pk, sk = read_keys("rsa-keys/public.pem", "rsa-keys/private.pem")
    img = cv2.imread("images/" + img_filename)
    c, r, enc_img = enc(img, pk, verbose=True)
    print("Time to encrypt:", time.time() - start)
    cv2.imwrite("encrypted/" + img_filename, enc_img)

    start = time.time()
    dec_img = dec(enc_img, c, r, pk, sk, verbose=False)
    print("Time to decrypt:", time.time() - start)
    cv2.imwrite("results/" + enc_filename, dec_img)
```

References

[1] A. Soleymani, M. J. Nordin, and Z. M. Ali, "A novel public key image encryption based on elliptic curves over prime group field," *Journal of Image and Graphics*, vol. 1, no. 1, pp. 43–49, 2013.

[2] M. Kumar and A. Vaish, "An efficient encryption-then-compression technique for encrypted images using svd," *Digital Signal Processing*, vol. 60, pp. 81–89, 2017.

[3] D. Schonberg, S. Draper, and K. Ramchandran, "On compression of encrypted images," in *2006 International Conference on Image Processing*, vol. 128, 11 2006, pp. 269 – 272.

[4] L. Sathyalakshmi and A. Mohanarathinam, "Efficient method of compressing encrypted images," *International Journal of Science and Research*, vol. 4, no. 2, pp. 1620–1623, 2015.

[5] T. Chuman, W. Sirichotedumrong, and H. Kiya, “Encryption-then-compression systems using grayscale-based image encryption for jpeg images,” *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1515–1525, 2019.

[6] I. Ozturk and I. Sogukpinar, “Analysis and comparison of image encryption algorithms,” *International Journal of Information Technology*, vol. 1, no. 2, pp. 108–111, 2004.

[7] C. Want, J. Ni, and Q. Huang, “A new encryption-then-compression algorithm using the rate-distortion optimization,” *Signal Processing: Image Communication*, vol. 39, no. A, pp. 141–150, 2015.

[8] J. Chen, Y. Zhang, L. Qi, C. Fu, and L. Xu, “Exploiting chaos-based compressed sensing and cryptographic algorithm for image encryption and compression,” *Optics & Laser Technology*, vol. 99, pp. 238–248, 2018.

[9] W. Jang and S.-Y. Lee, “Partial image encryption using format-preserving encryption in image processing systems for internet of things environment,” *International Journal of Distributed Sensor Networks*, vol. 16, no. 3, p. 1550147720914779, 2020.

[10] K. Jiao, G. Ye, X. Huang, B.-M. Goi, and W.-S. Yap, “An image encryption scheme based on public key cryptosystem and quantum logistic map,” *Scientific Reports*, vol. 10, no. 21044, 2020.

[11] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[12] OpenCV, “Open source computer vision library,” 2015. [Online]. Available: https://github.com/opencv/opencv

[13] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[14] S. A. Stü"vel, “Pure python rsa implementation,” 2021. [Online]. Available: https://pypi.org/project/rsa/

[15] “Generate secure random numbers for managing secrets.” [Online]. Available: https://docs.python.org/3/library/secrets.html

[16] TeunSpaans. (2004) New York aster (Symphyotrichum novi-belgii) at the Florence Nightingalepark, The Hague. Licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. [Online]. Available: https://commons.wikimedia.org/wiki/File:AsterNovi-belgii-flower-1mb.jpg

[17] N. Jaques and M. Kleiman-Weiner. Types of Machine Learning paper. [Online]. Available: https://twitter.com/natashajaques/status/1387859601555554304/photo/1

[18] The ImageMagick Development Team, “Imagemagick.” [Online]. Available: https://imagemagick.org

[19] ——, “Imagemagick command line options: Quality.” [Online]. Available: http://www.imagemagick.org/script/command-line-options.php#quality

[20] ——, “Imagemagick command line options: Gaussian blur.” [Online]. Available: http://www.imagemagick.org/script/command-line-options.php#gaussian-blur

[21] OpenSSL, “Open source toolkit for the transport layer security,” 2020. [Online]. Available: https://github.com/openssl/openssl

[22] A. Soleymani, M. J. Nordin, and E. Sundararajan1, “A chaotic cryptosystem for images based on henon and arnold cat map,” *The Scientific World Journal*, 2014. [Online]. Available: https://doi.org/10.1155/2014/536930

[23] P. Sankpal and P. Vijaya, “Image encryption using chaotic maps: A survey,” *Conference: Proceedings of the 2014 Fifth International Conference on Signal and Image Processing*, 2014. [Online]. Available: http://dx.doi.org/10.1109/ICSIP.2014.80

[24] S. Aileneni, “Image-encryption-using-chaos.” [Online]. Available: https://github.com/pilotcheetos/Image-Encryption-using-Chaos

[25] R. Jayaram, “Image encryption chaos maps.” [Online]. Available: https://github.com/RachanaJayaram/Image-Encryption-Chaos-Maps

[26] A. Das, “Homomorphic image encryption.” [Online]. Available: https://github.com/chronarchitect/Homomorphic-Image-Encryption

[27] S. S. Bedi, “Image encryption based on rubik’s cube.” [Online]. Available: https://github.com/sahibjotsingh/Image-Encryption-Based-on-Rubiks-Cube