Cryptography Lab CS355

Cryptography Project Report

Implementing Visual Cryptography and Exploring Various Applications

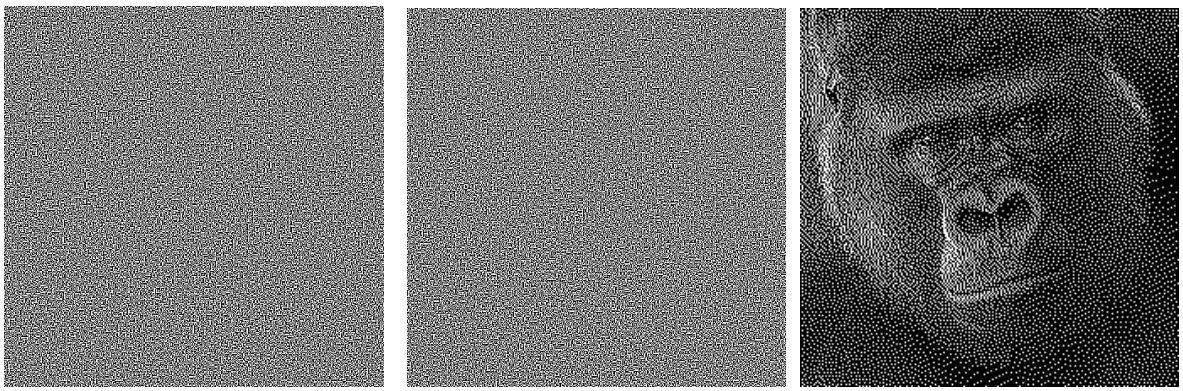
Aman Seervi (21CSB0B01) & Siddartha Gallipelli (21CSB0F25)

Introduction

Visual cryptography is a cryptographic that allows for the encryption of images such that decryption can be performed visually without the need for complex computations. It involves splitting an image into multiple shares, where each share alone reveals no information about the original image, but when combined with other shares, the original image can be revealed visually.

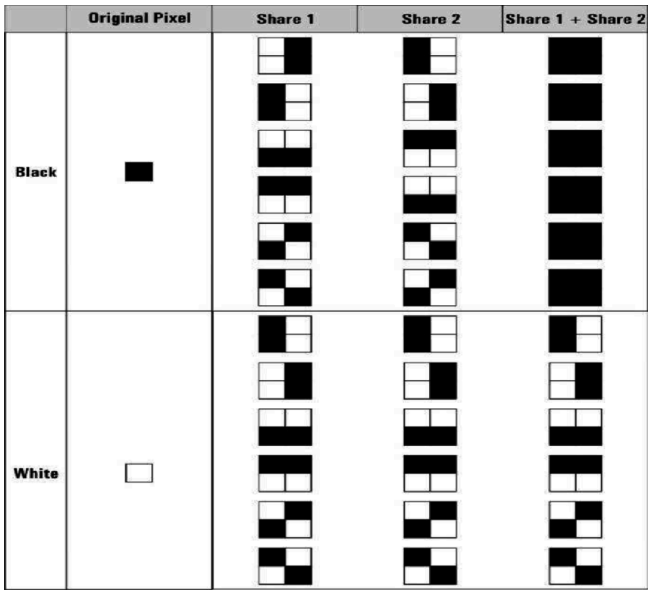
It was first proposed by Moni Naor and Adi Shamir in 1994 [1]. Since then, various extensions and improvements have been made, including methods for generating meaningful shares and enhancing security. Applications have expanded to fields such as secure authentication, watermarking, and privacy-preserving data sharing.

An example :



Original Approach

In the original implementation, each pixel is divided into subpixels and the original image is split into 2 shares which are then overlapped to get the final decrypted image. Each collection of subpixels will have an equal number of black and white subpixels. The colors of the sub-pixels are assigned in such a way that when the two shares are overlapped, the previously black-colored pixel will still have all its sub pixels black, and the previously white-colored pixel will have half of its subpixels black.



Since there are six different blocks (4C2 options for subpixels collection) for each pixel in a share that are randomly chosen, decryption with a single share is impossible, taking $6^{(m*n)}$ states ($m * n$is the size of the original image) for a brute force attack to decrypt the secret from a single share. The same idea can be extended to generate n shares.

This original technique had some major disadvantages :

1. It works only on Binary Images
2. It doubles the size of the image (i.e from $m * n$ to $2m * 2n$)
3. Decrypted image doesn't have the same contrast as the original image (i.e it's lossy)

Improvements

Various other algorithms have since been put forward to tackle the disadvantages of the original method. Additional advancements have extended the technique to handle grayscale as well as color images.

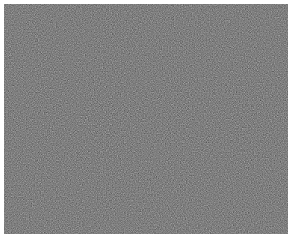
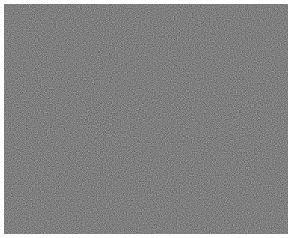
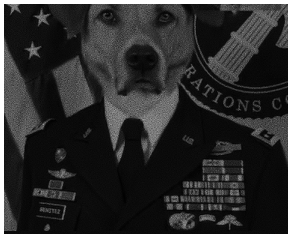
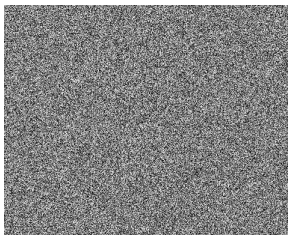
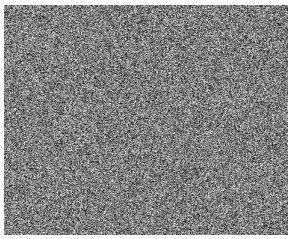
We implemented two different algorithms to generate shares , 1 of which only deals with grayscale images, while the other can handle both grayscale and color images.

1. Bit Level Decomposition :
    - Works for grayscale images
    - Much better contrast than the previous algorithms
    - Still generates shares that are double the size of original image
2. Modular Arithmetic :
    - Uses the idea of modular arithmetic and its cyclic ring nature
    - Generates shares that are of same size as the original image
    - Even better decrypted images than the previous method

It's important to highlight that with the modular arithmetic method, although shares are generated, they can no longer be simply printed and overlapped to reconstruct the original image.

Below is a demonstration of the techniques at work :

Input images :

Algorithm	Share 1	Share 2	Decrypted Image
Bit Level Decomposition			
Modular Arithmetic (Grayscale)			
Modular Arithmetic (Color)			

Implementations of some more algorithms and their performance in terms of the loss incurred in the decrypted image can be found here.

Extending Applicability To Videos

Having successfully applied the technique to images, the natural reaction was to apply it to encrypt videos as well. Videos are nothing but a continuous stream of individual frames, each representing a specific moment in time. When played in succession at a high speed, these frames create the illusion of motion, forming the basis of video content.

So, we implemented the code for this which takes the live recording from the front camera and splits it into two shares. But when implementing, some things need to be taken into account. First of all, considering a 30fps video, our encryption algorithm would be given a frame every **1/30 =0.033 seconds**. This time can be thought of as the threshold and an implementation taking more time than this will not be able to keep up with the live video.

While the exact performance still further depends on the system usage at that particular moment, we can think of this as an upper limit. I tried encryption using the modular arithmetic approach on images of different sizes and here are some results :

Dimension	**Average Time (10 tries at 4500/7600 RAM)**
540*360	0.0198
700*495	0.03513
1400*700	0.113

Clearly, while the technique could work for real time video calls for videos of lower quality (like 540*360), it may barely be usable on most systems for videos of dimension 700*500 and above, which is not considered good in modern times. Anything beyond that and it becomes too slow to even be considered useful.

Based on our expertise, the code was optimized to the best of our abilities in Python, and the recorded time accurately reflects real-world average computer performance. But the problems don't end at that. Even if a further optimized implementation (maybe in languages like C++), coupled with perhaps hardware support for the encryption was made available, it would still not make sense to make 2 shares and then send them to the client.

To overcome this, we implemented a strategy where one share remains constant, generated randomly at the beginning. This share is provided to the client initially. Subsequently, the sender encrypts the frames based on this share and transmits them to the client. Upon reception, the client decrypts the frames using the provided share. The problem with this technique is that while the individual frames sent by the sender still do not convey any meaning, when they are put together, the overall movement in the video can be seen. For example, if it is a person sitting and waving his hand, you may not be able to identify the person, but the fact that it is someone waving his/her hand would be clearly noticeable. Similar facts regarding problems with implementing visual cryptography for videos were discussed here and are very much parallel with our conclusions.

A Probable Solution

We came up with yet another technique to overcome this, wherein similar to previous technique, the initial random share is given to the client (perhaps as a base64 string encrypted using either symmetric or asymmetric system). After the client receives the first frame from the sender, it decrypts it to get the original frame. At this point of time, only both the sender and receiver have knowledge of the first frame of the video. The sender can now set this frame as one of the shares for encrypting the second frame. The receiver also knows the scheme and can hence use the retrieved frame to decode the next share it receives. The shares need not be encrypted and can be sent directly. The process continues until the stream is complete.

Problems With This Solution

There are still problems which would need to be addressed if one is to implement this technique. Modern video streams use compression techniques. The visual cryptography based approach however is applied on pixel values and the shares obtained on the other side will not be the exact same as sent by the sender due to the compression. This will not only decrease the quality of that particular frame, but also cause a problem to all the upcoming frames since the nth frame at the sender and receiver is not the same.

We try to better visualize this problem. Up till now the results obtained above were by decrypting using the numpy arrays of the shares, i.e the shares were never saved as jpg or png and hence left the main memory. But if we save the shares, then they will undergo loss upon reconstruction and the image quality of the final decrypted image will be reduced.

We can clearly see that there is a significant loss. If we want to avoid this loss on the streams, we would need to send the individual pixel values which again is unsuitable due to the bandwidth it would demand.

A General Discussion on the efficiency of Visual Cryptography Schemes

At this point when discussing the efficiency of visual cryptography, it is important to understand the reason why it is hard to use it even for exchanging photos , let alone any real time streaming applications. The main problem is that visual cryptography works on individual pixels and not the underlying binary data. Take for example the below image:

The dimension of the image is 1400*700 pixels and has 3 color channels. Given that each color value for each pixel is represented by a byte, the image is of a total 29,40,000 bytes, i.e 2.8 MB when stored as an array in the primary memory. Visual cryptography would work on each of these pixel values. But when we look at the actual space this image takes on the disk, it is only 145.4 kB. This is because the actual pixel values aren't stored. Rather, it's stored in the jpg format, which essentially does some compression. Although this

compression is lossy, the image obtained on opening is good enough for anyone to notice any changes. If we were to use AES as our encryption scheme, it would be operating on this small data (145 kB). Clearly there's a world of difference between the two. Besides, AES is lightning fast. Some of the most efficient implementations for AES encryption use lookup tables. Coupling this with the hardware support given by intel for AES, we get down to a speed of up to 1.3 clock cycles/byte. This equates to encrypting 1GB of data in around 0.634 seconds. Even without this top speed, AES is still way ahead of the visual cryptography techniques.

Is All Lost?

So, does visual cryptography not find any suitable application? The above discussion only shows that it is hard for visual cryptography to replace the traditional methods when it comes to the most common application like exchanging photos or videos over a traditional client server architecture at scale. Any approach to make it work leads in obstruction due to one or multiple of the following reasons :

- Does not work well if lossy compression is applied to shares
- Required more bandwidth if pixels are sent as it is
- Shares if stored on disk will take much more space (due to pixel values begin stored)
- The encryption itself is much slower as compared to famous algorithms like AES

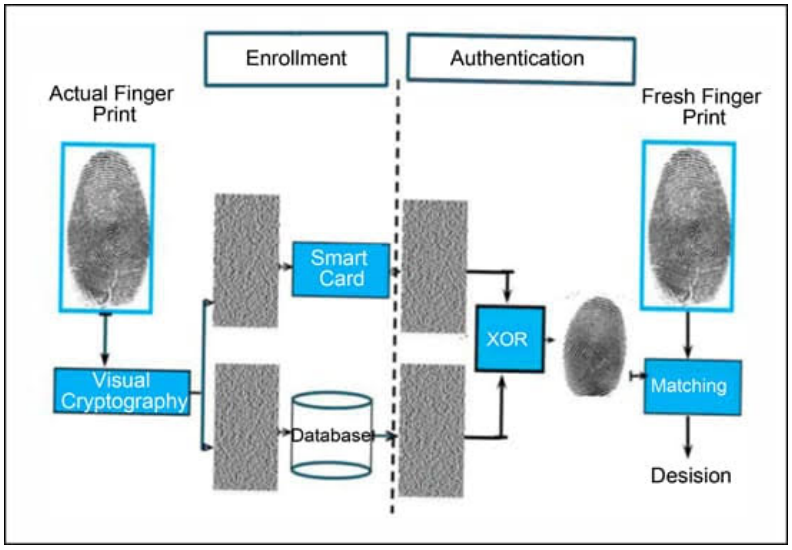
But this does not mean that it is of no use. It can still be used if an n-secret sharing scheme is required. It finds use in applications that do not require the storage of the shares that are generated in image format like jpeg/png etc. It can be used in applications where we can store the images physically in an encoded format and in applications where authentication is not completely digital and requires some human element. One specific use case is in the implementation of a voting system,where different shares of an image can be used to authenticate the voter.

Voting System

First let us draw an overview of how such a system works then let's talk about how this is implemented. We know that any person willing to vote should first register themselves as a voter in the registry after which they would be given a voter id which allows them to vote.

Let's track the entire voting process for better understanding. First comes the registration process. When a person goes to register, his fingerprint along with other details are taken to uniquely identify him. This is done by an independent machine set up by the voting commission. This machine creates 2 shares of this fingerprint and embeds one share into the voter id and stores the other in the database of the voting commission.

After registration comes the actual voting. When the person goes to vote he carries the voter id issued to him and uses this to authenticate himself. An independent machine recreates his fingerprint using the 2 shares, one from the voter id and one from the database. This image generated is compared with the actual fingerprint of the user. Based on a similarity of the fingerprints the voter is authenticated i.e either allows to vote or denies the voter.



Now let's talk about how this was implemented, there are 3 entities: the election commission which manages the database, the user i.e. the voter and an independent machine that issues the voter id and also verifies the user during voting.

Implementation

In the implementation we simulated the share being encoded in the voter id, by storing it in a database containing user data. So the independent entity will now fetch both the shares from the database in order to verify with the actual fingerprint of the user. This image

generated by the independent entity is compared with the fingerprint of the voter by using the histogram of cv2 library in python. The user enters his fingerprint both during registration and during the verification process. Entering a fingerprint is emulated by allowing the user to upload an image.

For the implementation, a full-stack website was created. React is used for the front-end and flask for the backend. Flask was a natural choice because python provides a large number of image processing libraries which made the task of converting images into the appropriate storage type much easier.MongoDb is used as a database which contains 2 collections: one for election commission and one for user data. The election commission collection contains one share of the fingerprint obtained during registration. User collection contains the user details and the share that is supposed to be encoded into the voter id, which as mentioned above has been simulated by storing in the database. The shares had to be stored as it is, i.e pixel values of the image to avoid loss and the disadvantage that come with it, as discussed previously. For this, the numpy array representing the shares was converted to a base64 string. The format was chosen owing to the ease of converting images directly into base64 and vise-versa, and storage of a string in a collection does not provide any extra challenges.

So as a dry run, the voter initially registers himself by providing details such as name,age,address etc. Along with this he also uploads an image of his fingerprint. This image will be broken into shares in the backend and stored as base64 strings in admin and user collection. When the user goes to the voting page, he is asked to verify himself, which he does by uploading an image of his fingerprint. To check the authenticity of the voter ,the 2 base64 strings are fetched and the original image of the fingerprint is reconstructed. These 2 images are compared using the histogram from cv2 library, and if the similarity is greater than the value of the threshold that is set, which in our case is 80% (can be changed based on the required accuracy) user is allowed to vote.

The code and other materials are available at :
https://github.com/Siddartha25/Voting-System-Using-Visual-Cryptography-Techniques