Image Secret Sharing

Jordi Panadès Closes

ICT Systems Engineering

Seguretat i Secret de la Codificació de la Informació TIC

Course 20-21, Project X.3

Table of Contents

Figures

1. Introduction

How to keep a secret has always been a very relevant issue, having an impact in many applications. Two major approaches to achieve this are information hiding and secret sharing. A very common and interesting method for information hiding visual secret sharing.

The gist of it is to encode a secret image intro n shadow images called *shares*, where any k or more of them can be combined to recover the secret image, but any combination of $k-1$ or fewer don't reveal any information of the secret image. This is denoted as a *(k,n)-threshold* visual cryptography scheme and was first described by Naor and Shamir (1).

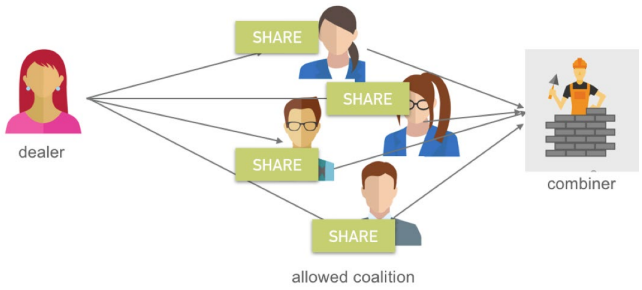

Figure 1. Visual secret sharing example

The objective of this project has been to implement a program with GUI to allow the user to generate shares from a source image using different secret sharing schemes and also be able to combine multiple shares to obtain the secret image.

Some other papers that have been useful to this project are *"Visual cryptography for gray-level images by dithering techniques"* (2) and *"Compartició de secrets"* (3).

2. Functionalities

2.1. Visual Secret Sharing Schemes – Dealer and Combiner

In this project multiple *(k,k)-threshold* visual secret sharing schemes have been implemented:

- Digital Black & White (binary XOR)
- Digital Color (bitwise XOR)
- Visual Black & White (logical OR, Naor and Shamir scheme)

The program has the option for both generating and combining shares from any scheme.

2.2. Black and White tool

A tool to convert the color input image to black and white is enabled when using any black and white scheme. To convert them to black and white two options are provided, using a threshold or a dithering algorithm.

2.3. GUI

To make thing user-friendly, all the interactions are done through a GUI. It's divided into two tabs at the top: *Generator* and *Combiner*. Both tabs have the same structure, an *Input/s* area at the left and an *Output/s* at the right.

In the *Generator Input* area, you can:

- Load the source image to share
- Chose the secret sharing scheme
- Edit it if it needs to be in black and white
- View the source image and edited version
- Select how many shares you want

In the *Generator Outputs* area, you can:

- Generate, view and save the shares
- Get notified if you need to regenerate the shares because of a change in the input settings

In the *Combiner Inputs* area, you can:

- Load and view the shares (must match size)
- Unload them from the program (all at once or a specific one)
- Chose the secret sharing scheme

In the *Combiner Outputs* area, you can:

- Combine the shares to get the secret image
- View and save the secret image
- Get notified if you need to regenerate the shares because of a change in the input settings

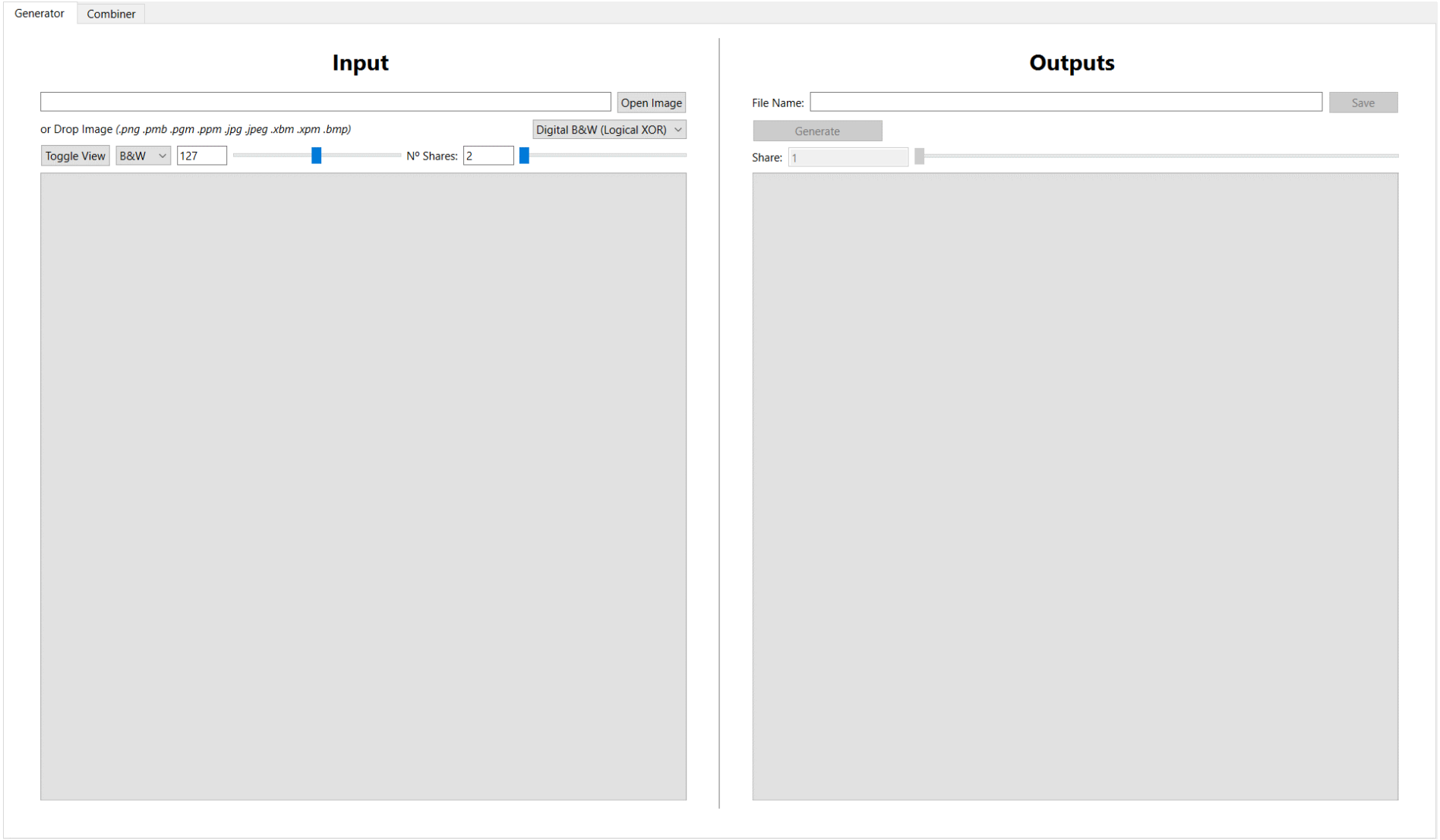

Figure 2. Generator GUI

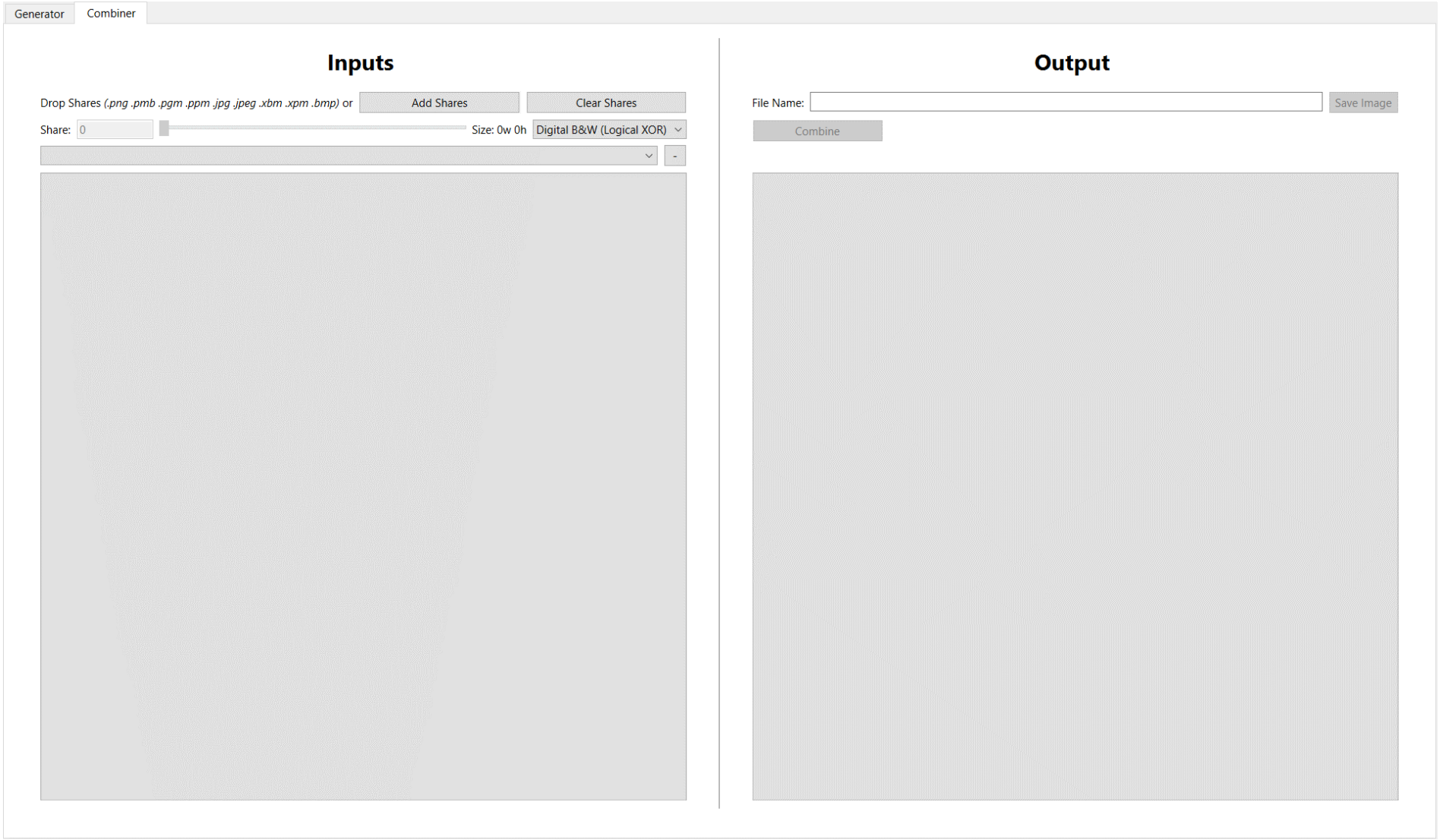

Figure 3. Combiner GUI

3. Implementation

In these sections we won't go in depth on how source code work since the GUI is a pig part of it and would obfuscate the secret sharing schemes. Still, we'll explain the basic concepts so that anyone can recreate these visual secret sharing schemes.

If the reader wants to understand how the GUI is implemented, we recommend reading the source code and consulting the *PySide* documentation (4).

3.1. Programing language and dependencies

To implement the program Python has been used because of its simplicity, readability and libraries that has. The libraries used are:

- PySide*:* library of the corss-pataform GUI toolkit Qt.
  - It has been used for creating all the user GUI and has also been extended to two custom modules (widgets).
  - *QRandegWidgets:* creates a widget that has a slider and a text area, both widgets are "linked" and a change in one affects the other.
  - *QPhotoViewer:* it's a modified version of a StackOverflow code (5). It's used to display, zoom and navigate an image.
- PIL*:* it's a library that adds image processing capabilities. It has been used for:
  - Reading images
  - Convert images to black and white using a threshold
  - Convert images to black and white using dithering.
  - Convert images to a *NumPy* arrays
  - Convert images to a `ImageQt` for *PySide* to display using the build module *QPhotoViewer.*
- NumPy*:* library that adds support for large, multidimensional arrays, matrices and their operations, and allows to work with them as vectors, improving performance.

3.2. Basic image manipulation used

3.2.1. Image to Arrays and vice versa

For reading images, as said before, the *PIL* and *NumPy* library have been used. Given a path, we can easily have it as an array:

```
img = Image.open(path)
img_array = numpy.array(img)
```

Once we have processed the array, it's also possible to convert a *NumPy* array to a *PIL* image.

```
img = Image.fromarray(img_array, 'L')
```

You can indicate the mode to use when reading the array with the second argument (6).

3.2.2. Image to black and white

For processing the image in black and white given a threshold we'll use the *PIL* library.

```
threshold = 127
img = Image.open(path)
fn = lambda x : 1 if x > threshold else 0
img_bw = img.convert('L').point(fn, mode='1')
```

And for converting the image to black and white using a dithering algorithm we only have to:

```
img = Image.open(path)
img_bw = img.convert('1', dither=True)
```

3.2.3. QPhotoViewer module - Display Images using PySide

Displaying images in a window becomes very easy when using the custom module *QPhotoViewer*. For example, an app for rendering images that are loaded using drag and drop only requires this code:

```
app = QtWidgets.QApplication()
window = QtWidgets.QWidget()
window.setGeometry(500, 300, 800, 600)
viewer = QPhotoViewer(window)
VBlayout = QtWidgets.QVBoxLayout(window)
VBlayout.addWidget(viewer)
window.show()
app.exec()
```

Also, when the image/s are dropped, the widget emits the paths as a list of paths to any connected function suing the `photoDropped` signal. To connect a function, you only have to do:

```
viewer.photoDropped.connect(self.get_dropped_paths)

def dropped_paths(self, paths):
   …
```

The class *QPhotoViewer* can also be updated with the function `setPhoto()`, render an image from a given path:

```
viewer.setPhoto(<pixmap:QPixmap>, <fit:bool>)
```

or

```
viewer.setPhoto(<path:str>, <fit:bool>)
```

This means that we can render an image from *PIL*:

```
pix = QPixmap.fromImage(ImageQt.ImageQt(self.image))
viewer.setPhoto(pix, fit)
```

3.3. Visual Secret Sharing Schemes

3.3.1. Digital Black & White (logical XOR)

This scheme basically takes any black and white image and interprets each pixel as a Boolean value:

Color	Boolean
White	True (1)
Black	False (0)

Figure 4. Digital Black and White color interpretation

Once we have the image as a 2-dimensional matrix *(S),* we need to generate k shares (S_k) with the same size as S so that the logical XOR of these reveals the original image S. This can be mathematically written as:

$$S_0 + \cdots + S_k = S$$

But how do we generate the shares? If they must be random so not to give any information away, how do we ensure that they add up to generate S? If we rearrange the expression, we'll see the solution:

$$S_0 + \cdots + S_k = S$$
$$S_0 + \cdots + S_{k-1} = S - S_k = S + S_k$$
$$S_0 + \cdots + S_{k-1} - S = S_k$$
$$S_0 + \cdots + S_{k-1} + S = S_k$$

So, we only need to randomly generate $k - 1$ shares and then XOR them with the source image to get the last share S_k.

To obtain the secret image, we'll need to gather all the images and the inverse process, which is also a logical XOR.

A simple demonstration is to use a string of bits and only two shares (k=2).

Generation:

$$\begin{array}{rcccc} & 0 & 0 & 1 & 1 \\ + & 0 & 1 & 1 & 0 \\ \hline & 0 & 1 & 0 & 1 \end{array} \quad \begin{array}{l} = S_0 \\ = S \\ \hline = S_1 \end{array}$$

Combiner:

$$\begin{array}{rcccc} & 0 & 0 & 1 & 1 \\ + & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array} \quad \begin{array}{l} = S_0 \\ = S_1 \\ \hline = S \end{array}$$

Considering that we already a black and white image in *NumPy* array form called `img_bw`, the scheme code is:

```
n_shares = 10
shares = []

# Generate random S_i, ... S_(i-1)
shape = img_bw.shape
for i in range(n_shares - 1):
   share = numpy.random.choice([False, True], shape)
   shares.append(share)

# S = sum(S_1, ..., S_i) --> S_i = S + sum(s_1, ..., S_(i-1))
share_n = numpy.copy(img_bw)
for i in range(n_hares - 1):
   share_n = numpy.logical_xor(share_n, shares[i])
   shares.append(share_n)
```

And the code to combine the shares into the original image will be:

```
# S = sum(S_1, ..., S_i)
image = shares[-1]
for i in range(len(shares) - 1):
   image = numpy.logical_xor(image,shares[i])
```

3.3.2. Digital Color (bitwise XOR)

This scheme basically does the same as the digital black and white but instead of using a black and white image, can be used with any image. This can be achieved because each RGB pixel is operated at a bitwise level, so if a pixel is:

$$(29, 129, 67)$$

It will be operated as a string of bits:

$$0100\ 1111\ 1000\ 1011\ 0100\ 0011$$

Considering that we already a color image in *NumPy* array form called `img`, the scheme code will be very similar to the previous scheme:

```
n_shares = 10
shares = []

# Generate random S_i, ... S_(i-1)
shape = img.shape
for _ in range(n_shares - 1):
   share = numpy.random.randint(0, 255, shape, dtype=numpy.uint8())
   shares.append(share)

# S = sum(S_1, ..., S_i) --> S_i = S + sum(s_1, ..., S_(i-1))
share_n = numpy.copy(img)
for i in range(n_hares - 1):
   share_n = numpy.bitwise_xor(share_n, shares[i])
   shares.append(share_n)
```

And, like before, the code to combine the shares into the original image will be:

```
# S = sum(S_1, ..., S_i)
image = shares[-1]
for i in range(len(shares) - 1):
   image = numpy.bitwise_xor(image, shares[i])
```

3.3.3. Visual Black and White

This last scheme a is the more complex one, as before referenced, the algorithm is explained by Nair and Shamir in their paper (1). We really encourage the reader to read the mentioned paper since here we'll go over some basic things and show the code, but you won't be able to fully grasping why or how it works.

The basic idea is that we have two special binary matrix that are k by 2^{k-1} in size used to generate the shares. One matrix is used to encode zeros and another to encode ones, so we'll name them C_0 and C_1. These matrices must ensure that:

- The OR (combination) of any k of the n rows of C_0 the relative color appears white.
- The OR (combination) of any k of the n rows of C_1 the relative color appears black.
- Any submatrix of rows of C_0 and submatrix of rows of C_1 are indistinguishable in the sense that they contain the same matrices with the same frequencies.

From here, we go through each pixel of the original image. If it's a 0 we use a column permuted C_0 matrix and assign each C_0' row to a share, if it's a 1 we do the same but with C_1.

To get C_0 and C_1, from the paper (1)*: "consider a ground set* $W = \{e_1, e_2, \ldots, e_k\}$ *of k elements and let* $\pi_1, \pi_2, \ldots \pi_{2^{k-1}}$ *be a list of all the subsets of even cardinality and let* $\sigma_1, \sigma_2, \ldots \sigma_{2^{k-1}}$ *be a list of all the subsets of W of odd cardinality (the order is not important).*

Each list defines the following $k \times 2^{k-1}$ *matrices* S^0 *and* S^1*: For* $1 \leq i \leq k$ *and* $1 \leq j \leq 2^{k-1}$ *let* $S^0[i,j] = 1 \; iff \; e_i \in \pi_j$ *and* $S^1[i,j] = 1 \; iff \; e_i \in \pi_j$*. As in the construction above, the collections* C_0 *and* C_1 *are obtained by permuting all the columns of the corresponding matrix."*

```
# Parameters
k = 5
W = list(range(1, k+1))
pi = []
sigma = []
for i in range(0, len(W)+1):
   # Combinations of size i from W elements
   listing = [list(subset) for subset in itertools.combinations(W, i)]
   if (len(listing[0])%2 == 0):
      pi.extend(listing)
   else:
      sigma.extend(listing)
pi[0] = [0]
S0 = make_S(W, pi)
S1 = make_S(W, sigma)

def make_S(W, set):
    rows = len(W)
    columns = pow(2, rows-1)
    S = numpy.ones((rows, columns), dtype=numpy.bool8())
    for i in range(rows):
        for j in range(columns):
            if W[i] in set[j]: # S0[i][j] = 1 iff W[i] in set[j]
                S[i][j] = 0
    return S
```

When generating the shares, we have to be careful, because if we simply substitute each pixel for a row, will be only dividing the pixel in the *width* axis. Consequently, the shares and revealed image will have a different width from the source. To be more precise, their size will be *heigh* by *width*2*$^{k-1}$.

A solution to the previous problem is to divide each pixel in both dimensions. To do it in the most efficient way we will try to square the row of C_i assigned to each share, if the length of the row isn't big enough, we'll can add the values from the same row, starting from 0. To find into how many subpixels we have to divide each pixel we use the area of a square:

$$Area = c^2 = len(S_{0\_row})$$
$$c = \sqrt{Area} = \sqrt{S_{0\_row}}$$

```
# Generate empty shares
subpixel_size = math.ceil(math.sqrt(len(S0[0])))
C_len = pow(subpixel_size, 2)
h, w = self.bw.shape
shares = []
for _ in range(k):
   share = [[] for _ in range(h*subpixel_size)]
   shares.append(share)

# Compute
Scols = S0.shape[1]
for i in range(h):
   for j in range(w):
      if self.bw[i][j] == 1: # White
         C = permute_matrix(S0)
      else: # Black
         C = permute_matrix(S1)
   for l in range(k):
      C_row = C[l]
      while len(C_row) < C_len:
         C_row = numpy.append(C_row, C_row)
      subpixel = C_row[:C_len].reshape(-1, subpixel_size)
      for sub_i in range(subpixel_size):
         shares[l][i*subpixel_size+sub_i].extend(subpixel[sub_i])

# Convert to numpy array
for i in range(self.n_shares.value):
   share = numpy.array(shares[i])
   self.shares.append(share)

def permute_matrix (M): # O(n)
    permutation = numpy.random.permutation(M.shape[1])
    idx = numpy.empty_like(permutation)
    idx[permutation] = numpy.arange(len(permutation))
    return M[:, idx]
```

To obtain the source image we physically overlay each image. Intuitively we would say this action is equivalent to a logical OR. But, since white is 1 and black is 0 and not white 0 and black 1, we do opposite operation, AND.

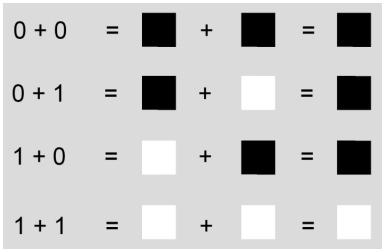

Figure 5. Truth table of visually overlaying two shares

So, if we want to do the physical overlaying in the computer, the code to combine the shares into the original image will be:

```
# S = sum(S_1, ..., S_i)
image = shares[-1]
for i in range(len(shares) - 1):
   image = numpy.logica_and (image, shares[i])
```

3.3.3.1 Improving execution speed

When testing the code, I realized that with big images it requires a considerable amount of time to even generate two shares. So, the code was optimized a bit to make it faster.

To test the changes in execution speed, the 300x300 image */test-img/stop.png* has been used to generate two shares. These tests have been done in a XPS 9500 laptop. Each improvement, also includes the previous improvements.

Permuting S

To permute the S_ matrices I was using some fancy NumPy indexing (7), which was around $O(n)$ where $n = \#\ S\_\ columns$:

```
   ...
   # Compute
   for i in range(h):
      for j in range(w):
         if self.bw[i][j] == 1: # White
            C = permute_matrix(S0)
         else: # Black
            C = permute_matrix(S1)
   ...

def permute_matrix (M): # O(n)
    permutation = numpy.random.permutation(M.shape[1])
    idx = numpy.empty_like(permutation)
    idx[permutation] = numpy.arange(len(permutation))
    return M[:, idx]
```

This works, but there's a more efficient way (8):

```
...
# Compute
Scols = S0.shape[1]
for i in range(h):
   for j in range(w):
      if self.bw[i][j] == 1: # White
         C = numpy.take(S0, numpy.random.permutation(Scols), axis=1, out=S0) # Permute
      else: # Black
         C = numpy.take(S1, numpy.random.permutation(Scols), axis=1, out=S1) # Permute
...
```

This made the generation 0,35 seconds faster on average, from 2.15 to 1.80 seconds. A decent improvement.

Not showing which pixel is being processed

Since the generation of medium and big images takes a lot of time, I thought of adding a counter of which pixel is being process.

```
...
# Compute
Scols = S0.shape[1]
for i in range(h):
   for j in range(w):
      self.forceInformUpdate(f'<i> Computing Pixel {i},{j}</i>')
...
```

This ended up being a bad idea, it took 715.90 seconds. Pretty bad compared to the optimized 1.80 seconds.

I also thought of randomizing when it was updated, to reduce the time the text is updated.

```
...
# Compute
Scols = S0.shape[1]
for i in range(h):
   for j in range(w):
       if numpy.random.randint(1,10001) > 9999:
         self.forceInformUpdate(f'<i> Computing Pixel {i},{j}</i>')
...
```

This indeed helped, but it was still bad, 2.3 seconds on average.

The solution has been to only update the text every 1 second, which has little impact, around 0,02 seconds more, an average of 1,82 seconds.

```
...
# Compute
text_timer = time.time()
for i in range(h):
   for j in range(w):
      if time.time()-text_timer > 1:
         self.forceInformUpdate(f'<i> Computing Pixel {i},{j}</i>')
         text_timer = time.time()
...
```

The power of NumPy - final code

NumPy is a very efficient python library to work with arrays this is because it has been implemented in C, which is much faster than python. Thanks to all the tools the library supports the code can be really optimized by: using NumPy arrays from the start, extending the whore S_ array instead of doing it to each line, and "embedding" the small subpixel array into the share (9). The final code for the generation of visual shares is:

```
# Generate empty shares
subpixel_size = math.ceil(math.sqrt(len(S0[0])))
C_len = pow(subpixel_size, 2)
h, w = self.bw.shape
self.shares = []
for _ in range(k):
   empty_share = numpy.zeros((h*subpixel_size, w*subpixel_size), dtype=numpy.bool8())
   self.shares.append(empty_share)

# Compute
Scols = S0.shape[1]
text_timer = time.time()
self.forceInformUpdate(f'<i style=" color:#aa0000"> Computing Pixel 0,0</i>')
for i in range(h):
   for j in range(w):
      if self.bw[i][j] == 1: # White
         C = numpy.take(S0, numpy.random.permutation(Scols), axis=1, out=S0) # Permute
      else: # Black
         C = numpy.take(S1, numpy.random.permutation(Scols), axis=1, out=S1) # Permute
      C_ext = C
      while C_ext.shape[1] < C_len: # Extend C so pixel can be divided and still squared
         C_ext = numpy.append(C_ext, C, 1)
      C_ext = C_ext[:,:C_len] # Get only needed values for squared pixel
      for l in range(k): # Distribute the pixel to each share
         subpixel = C_ext[l].reshape(-1, subpixel_size)
         self.shares[l][ i*subpixel_size:(i+1)*subpixel_size,
j*subpixel_size:(j+1)*subpixel_size] = subpixel
```

This last optimization has a very big impact tot the execution time, making it 0,68 seconds faster on average, from 1.82 to 1.14 seconds.

Overall improvement

Like before, the test has been done with a XPS 9500 using the images found in */test-img* and generating two shares.

Image	Size [px]	Before [s]	After [s]	Improvement [s]
stop	300 x 300	2,15	1,14	1,01
women	512 x 512	6,5	3,56	2,94
city	1216 x 684	21,06	11,1	9,96
mountains	3840 x 2160	200,52	109,92	90,6

Figure 6. Execution time improvements after optimizing the visual algorithm

4. References

1. Naor M., Shamir A. Visual Cryptography. *SpringerLink.* [Online] 1995. [Cited: 12 19, 2021.] https://doi.org/10.1007/BFb0053419.

2. Visual cryptography for gray-level images by dithering techniques. *ScienceDirect.* [Online] 2002. [Cited: 12 20, 2021.] https://www.sciencedirect.com/science/article/abs/pii/S0167865502002593.

3. COMPARTICIÓ DE SECRETS. *UPC Commons.* [Online] [Cited: 12 19, 2021.] https://upcommons.upc.edu/bitstream/handle/2117/127207/TFG%20Juan%20Rivas.pdf?sequence=1&isAllowed=y.

4. Qt for Python. *Qt.* [Online] 2021. [Cited: 12 19, 2021.] https://doc.qt.io/qtforpython/.

5. How to enable Pan and Zoom in a QGraphicsView. *StackOverflow.* [Online] 2 19, 2016. [Cited: 12 19, 2021.] https://stackoverflow.com/questions/35508711/how-to-enable-pan-and-zoom-in-a-qgraphicsview.

6. Modes. *Pillow.* [Online] [Cited: 12 20, 2021.] https://pillow.readthedocs.io/en/stable/handbook/concepts.html#concept-modes.

7. Rearrange columns of numpy 2D array. *Stack Overflow.* [Online] 11 12, 2014. https://stackoverflow.com/questions/20265229/rearrange-columns-of-numpy-2d-array.

8. *Stack Overflow.* [Online] 12 12, 2013. https://stackoverflow.com/questions/20546419/shuffle-columns-of-an-array-with-numpy/20546567#20546567.

9. How to "embed" a small numpy array into a predefined block of a large numpy array? *Stack Overflow.* [Online] 8 118, 2011. https://stackoverflow.com/questions/7115437/how-to-embed-a-small-numpy-array-into-a-predefined-block-of-a-large-numpy-arra.