# "Visual Cryptography" for Biometric Data Privacy

Daniel Moctezuma

Abstract

The purpose of this project was to implement a visual cryptographic technique using Shamir's Secret Sharing Scheme for improved Biometric Data protection. The techniques used were based on the theory that was discussed in class regarding Shamir's Secret Sharing Scheme.

## 1. Problem Statement:

The use of biometric data for privacy and security has now become a viable option for many applications. Biometric data is a convenient and often more secure way of preventing attackers from gaining access to personal items like laptops and computers. However, when this data is kept in a centralized source, it is very possible for malicious (and semi-honest) parties to gain very personal data about individuals by compromising the database storing the information. Furthermore, the inherent and literally personal nature of biometric data means that when such data is compromised, the consequences go far beyond intercepting a password.

## 2. Proposed Solution

By using a non-centralized form of data storage, it is possible to increase the amount of security of such sensitive data, thus improving the user's privacy. In this project, I examined the potential of Shamir's Secret Sharing to store images that reveal biometric data in a safe, non-centralized and secure way, focusing on creating separate shares that can be saved in different databases for improved security. The secret sharing scheme is a cryptographic technique that divides a message up into various shares, such that no single share contains enough information to recreate the original message. By combining *t* out of *n* shares, the image can then be restored. In my experiments and implementation, I examined the creation of shares for images and of "keypoint" shares for fingerprint points of interest. Though I had originally intended to implement the t,n, threshold visual cryptography scheme, a little research showed that these schemes are not always secure and that they degrade the quality of the original image. Additionally, this technique limits the number of shares that can be made, thus affecting the utility of the technique. In order to maximize security and retain image quality, I decided to devise a scheme that would encrypt the image in the traditional method of Shamir's Secret Sharing Scheme (using Lagrange Polynomials) but that could be flexible and more secure.

## 3. Implementation Details:

The implementation uses Python and uses a couple of different external modules. The Shamir Secret Sharing implementation I am using is a customized version of the implementation found on the Wikipedia website for Shamir Secret Sharing subject. Other modules I am using are numpy, OpenCV (and the extended image processing library) for computer vision functions, and PyFingerprint for reading information from a fingerprint sensor.

### A. Implementation Milestones

1. **Implement a Secret Sharing Cryptographic scheme on basic images.**

In the first part of the project, I implemented a “visual cryptographic” version of Shamir’s Secret Sharing Scheme and tested it on ordinary images. The goal was to create “shares” of a grayscale image by creating shares of each pixel in the image, thus creating *n* shares of the image which could then be reconstructed by combining *t* of the encoded shares. These resultant images are in color.

**2. Implement a Secret Sharing Cryptographic scheme on biometric data.**

The next part of the project was to apply this technique of “pixel encryption” to a set of thumbprint data, thus encrypting the biometric data of the thumbprint. In this section, I only created shares of the “keypoints” of the thumbprint. Keypoints are the actual unique points of the thumbprint that are used for verification in most thumbprint readers. These keypoints are found by applying a couple of computer vision techniques such as Corner Detection and image skeletonization. By encrypting these keypoints, one can create a much faster visual cryptographic encoding technique than the encryption of every pixel. Because this method only saves keypoints and not actual thumbprint images, this is also potentially safer. What we need to save here are the positions of these interest points instead of pixel values.

**3. Attempt to construct a reader and authenticator of biometric data.**

Lastly, I sought to try and construct a fingerprint “reader/authenticator” by using a fingerprint sensor. My goal was to read the fingerprint, extract the keypoints and then compare it to stored data to see if the fingerprint was “registered.” Due to limited time, I was unable to get full functionality in this segment, but I was able to get the sensor running and capture the image.

**B. Algorithm for Visual Secret Sharing Cryptographic Scheme**

To encode the image for t,n shares:

1. Read image and convert to array (OpenCV does this automatically when reading image)
2. Convert image to grayscale if image is not grayscale
3. Create an array of pictures (the shares) [n][i][j][3] where *n* is the number of shares you will be creating, *i* is the number of pixel rows in the image, and *j* is the number of pixel columns in the image
4. For all rows 0 to i-1 of image:

```
For all columns 0 to j-1 image:
        Secret_value = image[i][j]          //secret is pixel value of the image
        Shares[] = secret_share_Scheme(secret_value, t,n)  // n pixel shares
        For all n shares:
                Picture[n][i][j][0] = shares[k]/255        //result is val < 255
                Picture [n][i][j][1] = rangom in range (0,255)  //random pixel
                Picture [n][i][j][2] = shares[k] % 255   //remainder
```

Essentially, we read the image and convert to grayscale to get all the pixel values. We then run the secret sharing implementation on a pixel and get a much larger number. We hide this large number into the pixel values of a BGR color image by setting the value/255 as the Blue pixel value and value%255 as the Red pixel value. We then choose a random pixel value for the G pixel. The pixels thus contain one of the “share” values. The shares are spread across n random images. We do this for all pixels.

To decode the image from t shares:

```
1.  Read images and save them to an array of image arrays
2.  For all rows 0 to i-1 of image dimensions:
        For all columns 0 to j-1 of image dimensions:
            For t shares:
                Restored_value[t] = imaget[i][j][0]*255 + imaget[i][j][2]
            Restored_image_pixel = decrypt_secret_share(all t restored_values)
```

To decode, we need t of n images. We decode each pixel of the original image by looking at the BGR pixel values of each corresponding (i,j) pixel in the image and recovering each share value ($x_i$, $y_i$) from the images and “recover” the original message (the original pixel value of the image). By restoring all of the pixels, we are then able to recover the grayscale image.

For the thumbprint keypoint data we run a similar algorithm, but we encode the x and y coordinates of the keypoints using the same shamir secret sharing process. Recovery is done in a similar manner.

The computer vision algorithm for detecting the keypoints is done using the Harris Corner Detection function in the OpenCV library. There are some tuning parameters for the threshold, skeletonization and harris corner detection functions that affect the number of keypoints detected.

Harris Corner Detection (high-level algorithm):

```
1.read image
2. convert to grayscale (if needed)
3.  threshold image
4.  Thin/skeletonize image (I use opencv ximg skeletonization function)
5.  Run OpenCV Harris Corner detection function
6.  return detected corner points
```

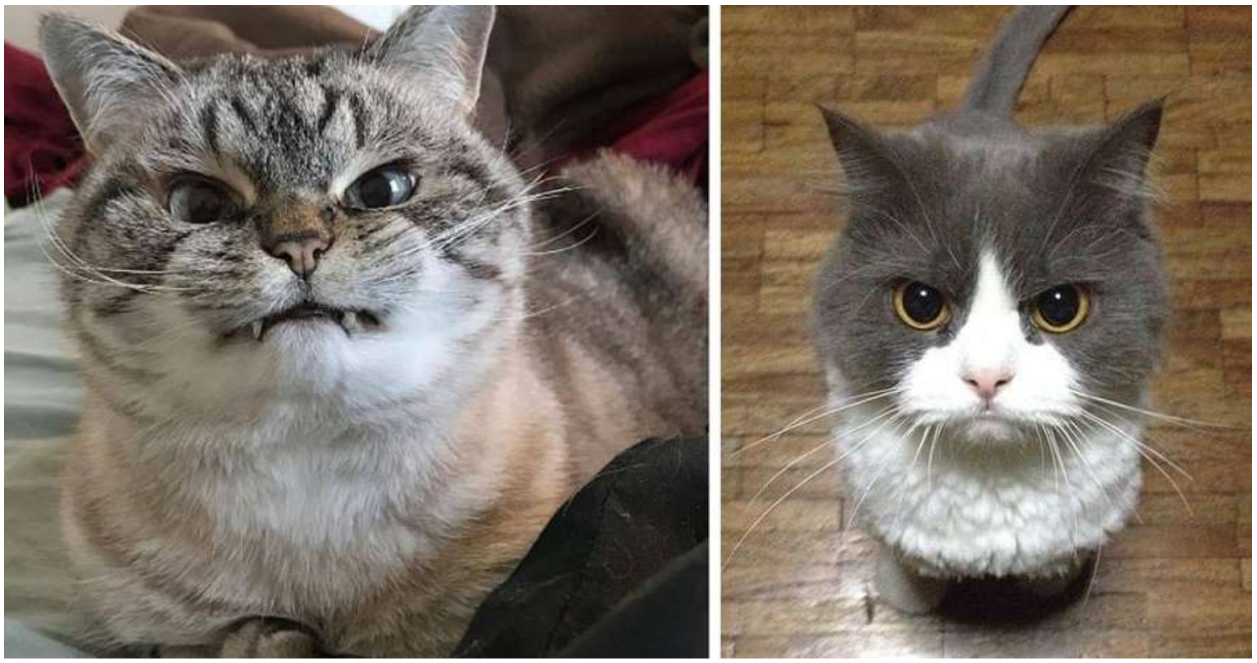
I am using an implementation that is similar to the one found in the book *OpenCV 3 Blueprints*, by Puttemans, Howse, Sinha and Hua

## 4. Results and Discussion:

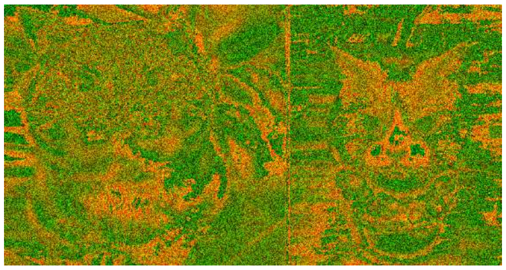
### A. Encrypting basic images:

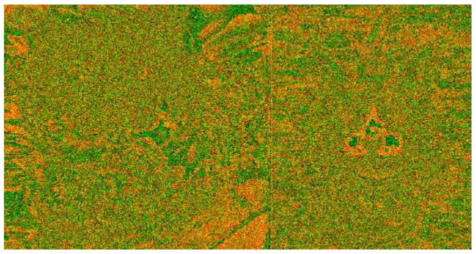
This technique works best on smaller images, as its complexity is O(k x m), where k is the number of rows and m is the number of columns in the image. The larger the image, the longer it takes for the algorithm to run. There are still sum bugs in the implementation, but it is mostly coming from the shamir secrete sharing implementation I was using. After a certain value of *t* and *n,* the program fails because of overflow issues in the calculations. Still, there are some interesting observations that can be made. One of the things I noticed, from testing on different images, is that images that are very homogenous, tend to not randomize the image very well. This makes sense because there are only 256 possible pixel values, but I noticed that when there was an image standing in a very homogenous background, the “shadow” of the image had a tendency to remain in the randomized pictures. I was most successful with the images when values of *t* were $< 4$. 3 seemed to work best, 4 was finicky, 5 is not very functional for restoration (the program tends to abort).

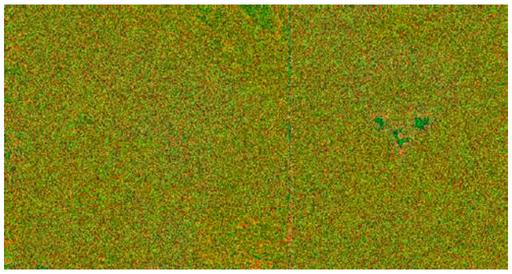
## A Cat Mugshot:

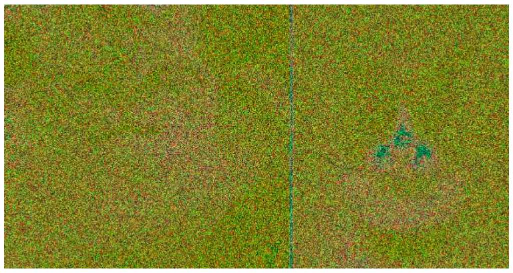


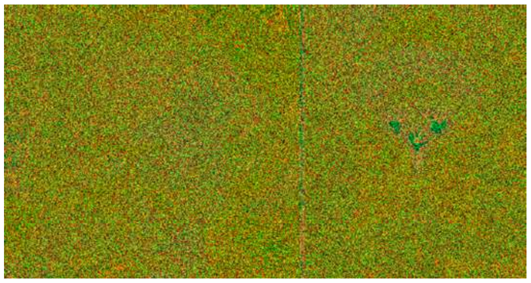
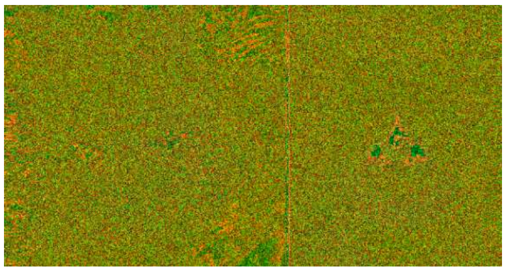
The picture of the cats was the first image I tested on, and the results were surprisingly good.  The image has a lot of contrasting colors, so the grayscale image has significant variation as well. Running a 3,8 threshold sharing scheme yields images like this:



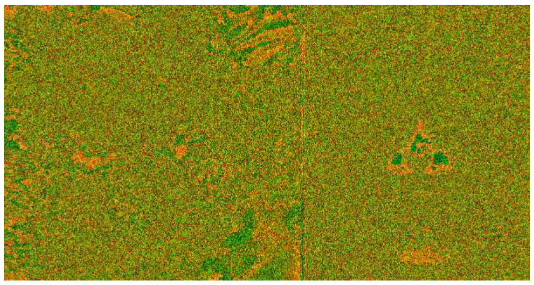
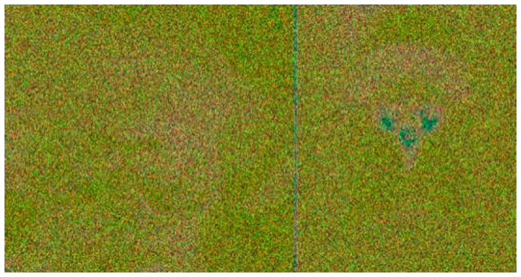




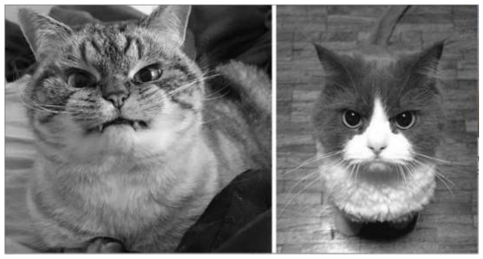


As you can see, the images are a little bit distinguishable in some shares, but only if you know what you are looking for.  Later on, it is much harder to tell what the image is.  As you can see from the cat on the right, the white of his nose is visible even in the more scrambled shares.

Due to the high amount of contrast in the image, the picture "encrypts" fairly well.  The only problematic thing was the amount of time it took to encrypt.
By combining (and decrypting) the information from any 3 shares of the 8 above, the image below results.

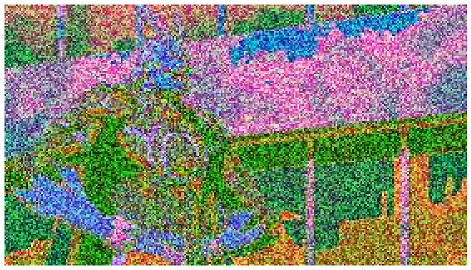


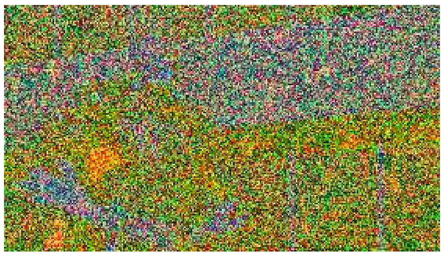
## The Superhero

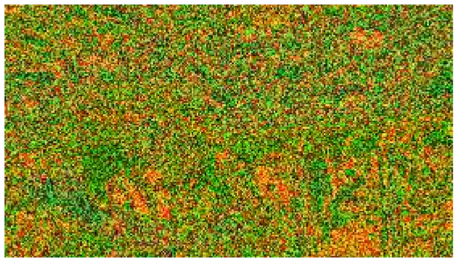
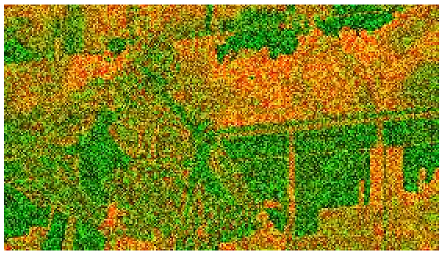
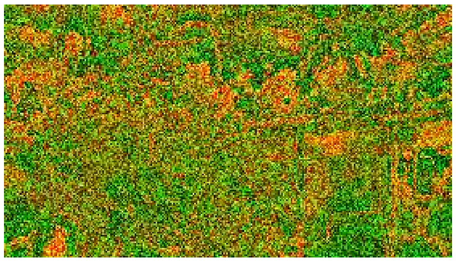
Running the encryption on this image, we encounter some of the less successful properties of this encryption properties.  Her are the results of a 4,6 threshold encryption on the picture.

Not how much easier it is to identify the building because of the lack of contrast between the pixels.

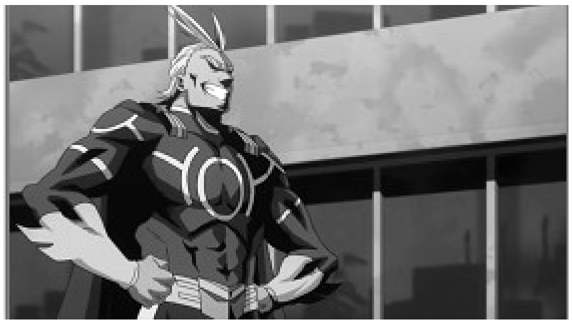
The results are mixed, but there are enough successful randomization that the best 4 shares could be used and the others discarded.

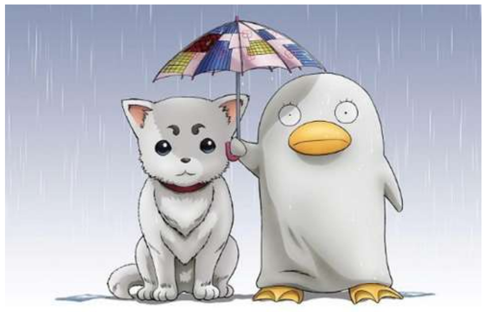




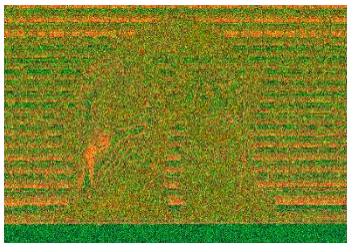
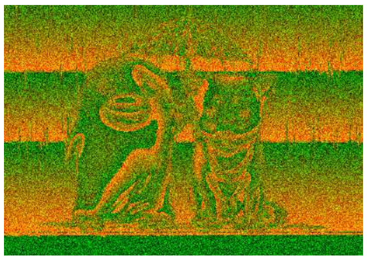
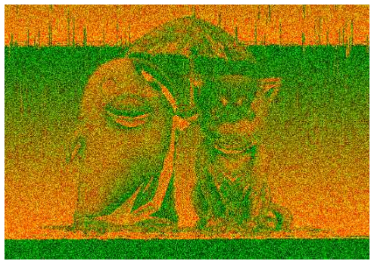
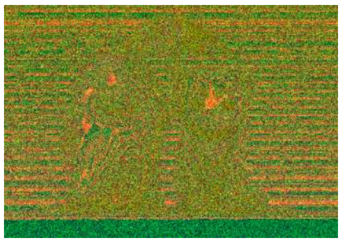
Still, we are able to get a successful reconstruction.

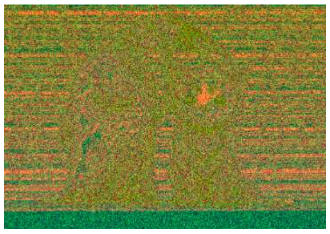
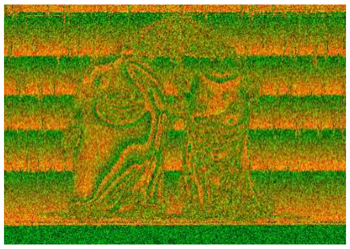
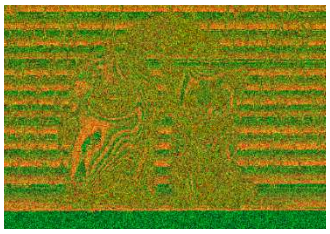
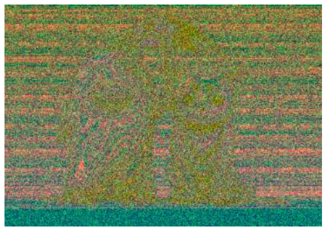




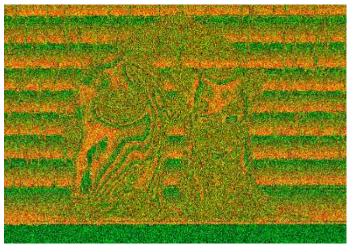
## The Dog and the Duck

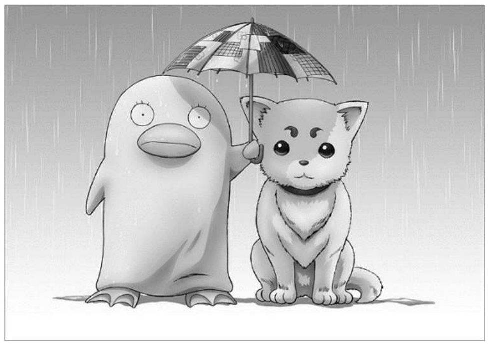
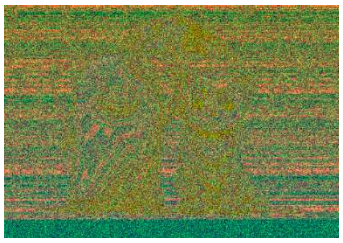
An image like this is where this method of image encryption is not very successful. In order to get a significantly scrambled image, we need to create a lot of shares, which involves more computation. Likewise, at the current iteration of the implementation, we need to have a low parameter for *t*. Another thing you will notice is the fairly uniform stripe pattern that emerges from the homogenous background. I think this may have to do with some of the randomization that turns out to not be so randomized. In the early stages, the picture is quite noticeable. With enough shares, the picture starts to be less recognizable, but if someone knew what the picture was, it would be easy to identify for many of the iterations.

In some iterations we see more blue, due to the randomization on the value of the green pixel.  Still, the reconstruction is once again lossless.



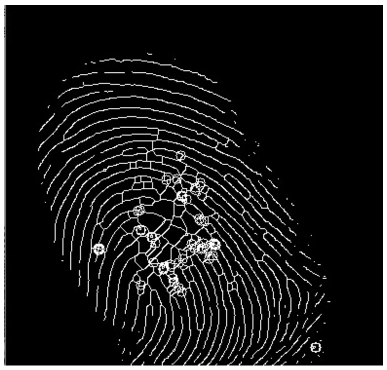



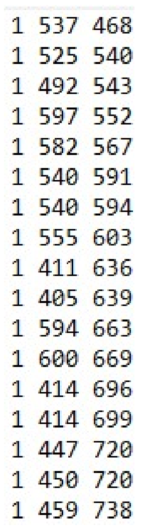
**Conclusion:**
There could be practicality in the use of this technique for things that require more scrutinous differentiation and specific identification.  As you see with the cats, even the image that closest resembles the original image is hard to distinguish, as there are no distinct markings that would tell you which cat is which.  Applying this to a database of pictures of people that need to be protected could secure their privacy by splitting up the shares of their faces on different protected servers.  That way, gaining access to the identity of the protected person would be much harder.

## B. Fingerprint Encoding Results (3,6 threshold test)

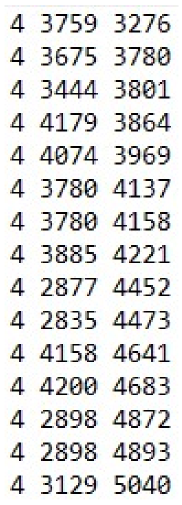
After doing a bit of research on the way fingerprints are used for biometric data, I came to the conclusion that the use of biometric fingerprint data was actually fairly good at preserving a person's privacy. In most fingerprint implementations, the fingerprint image of the registered user is not what is stored, rather it is what they call "areas of interest" that are mapped and saved as features in identifying a registered user. These areas of interest are a combination of corners, edges and other distinct features on the user's thumbprint. For this implementation, I captured some keypoints of interest using the harris corner detection algorithm and stored them in tsv (tab spaced value) files that could then be combined to get the original values. The program selects a fingerprint at random, detects the areas of interest, saves them on an "answer key" and then creates shares. It then picks 3 shares at random and tries to restore them. Those values are found in the "decrypted" file. Each run of this program will overwrite the previous result, so you can see that the values are indeed changing.

Below is a fingerprint that has been thinned so that the algorithm can then locate points of interest on the fingerprint. There are a lot of parameters in the smoothing, thinning and the corner/feature detection that can be changed such that each biometric authenticator/verifier can be unique. This introduces a slight amount of pseudo-randomization that improves the privacy/security for the user's biometric identity. Pictures of these thumbprints are found in the folder DB1_B.



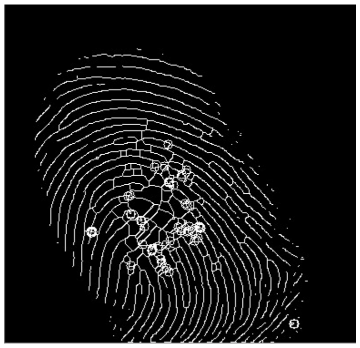


```
1 537 468
1 525 540
1 492 543
1 597 552
1 582 567
1 540 591
1 540 594
1 555 603
1 411 636
1 405 639
1 594 663
1 600 669
1 414 696
1 414 699
1 447 720
1 450 720
1 459 738
```



```
4 3759 3276
4 3675 3780
4 3444 3801
4 4179 3864
4 4074 3969
4 3780 4137
4 3780 4158
4 3885 4221
4 2877 4452
4 2835 4473
4 4158 4641
4 4200 4683
4 2898 4872
4 2898 4893
4 3129 5040
```

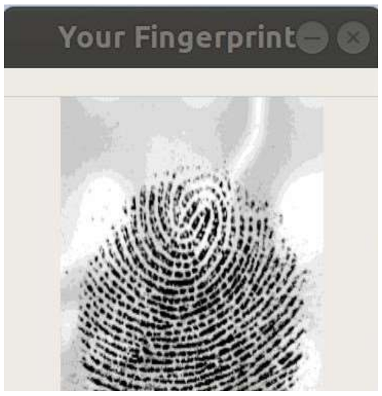
To the left, you see a sample of the data that is stored. This is a sample from two different tables that represent that values of the regions of interest. Verification is done by comparing the relative distance between points of interest and scoring them. If the score is below a threshold, the user is "verified". At the bottom right is the result of redrawing the circles on the points of interest after reconstructing the answer key.

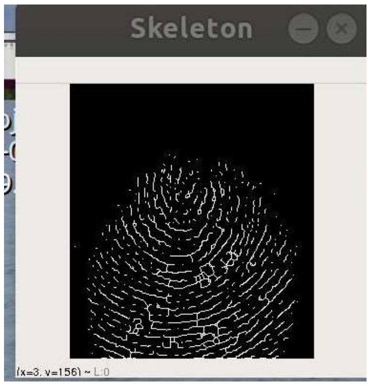


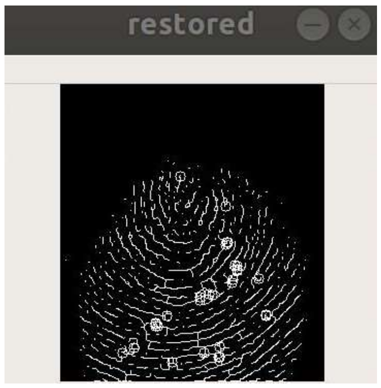
**C. “Home Brew” Fingerprint Reader and Verifier**

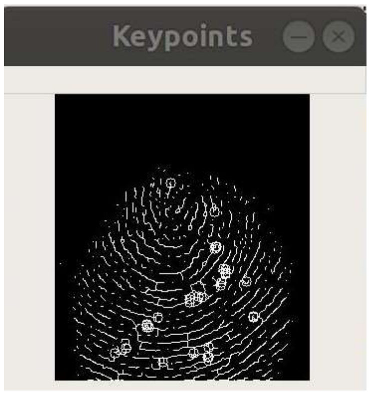
I was hoping to be able to implement a full fingerprint scanner, authenticator and verifier using the encrypted versions of keypts found in an image, but unfortunately I did not have enough time to perfect the verification process (there are **many** factors that are used in good verification). Two problems that need to be solved are done through the tuning of two other OpenCV functions that I was not able to perfect in order to get a full working verifier. This also affected the results in the second part of the assignment as well, as I was hoping to be able to match the fingerprints between the dataset. The two operations that need to be fine-tuned are the ORB (or SIFT) algorithm and the bruteforce Hamming matching function. Combined, these two functions are able to match descriptor points (our keypoints in the above section) and determine a score based on relative distance between points of interest. Unfortunately I was unable to complete that implementation. The pyfingerprint library that I used has a matching function, but it is tied very closely to the lower level functions of the computer and sifting through the code is a little bit of work. I also didn’t want to just ripoff the implementation, I wanted to try and create my own verifier.

I was, however, able to get the fingerprint sensor working and able to take a picture of my fingerprint and then analyze the keypoints, like in section 2. Results can be found below.









## 5. Conclusion

Having not known much about biometric data and having selected this project before I knew we were going to be studying Shamir's Secret Sharing Scheme, I was pleasantly surprised by the results of the experiment. Though I was unable to completely finish my ambitious goal of creating a fully functioning fingerprint authenticator and verifier, the implementations I was able to make were successful in improving the security of both the images and the fingerprint data that was collected.

For the image encryption algorithm, the success doesn't come from obscuring the "what" of the image, but more the "who exactly" of the image. Even in the most visible of the encrypted images of the cats, it was hard to tell distinct features on the cat's face. An encryption technique like that is more successful as a generalizer of an image as opposed to a complete obscurer. The advantage of such a technique however, is in its ability to preserve the quality of an image. As I had mentioned, I originally intended to use Shamir's Visual Secret Sharing scheme, but the technique was fairly limited in its utility and involved decreasing the quality of the image, two things I wanted to avoid. As stated earlier in the report, this technique could be used for encrypting images of people whose identities need to be protected for a specific purpose, think something like a protected witness list. If multiple computers hold shares of these images, theft of this information from one government database (or something of that nature) would not yield the identity of the protected witnesses or suspects. In the future, I would like to find a way to introduce more randomization in the scrambling of the images.

While working through the fingerprint reader implementation, I was pleased to also discover the difficulty and complexity of collecting fingerprint data. The amount of fine tuning of computer vision parameters involved means that the likelihood of all fingerprint readers collecting the same data is incredibly low. When applications do not store actual images of a fingerprint, the data collected is useless without knowing the methodology in which the data is stored. For example, if only certain keypoints from the fingerprint are stored, stealing the data will not be of much use (though technically, we should assume the encryption/decryption algorithms are public). Being able to split the data into shares only helps the security of the user. Because this information is already fairly obscure (the attacker has to know exactly what it is they are obtaining), further obfuscation yields better protection.

## 6. Resources (Books, Libraries and Implementations):

1. *OpenCV 3 Blueprints,* Packt publishing 2015
   Howse, Puttemans, Hua Sinha
2. OpenCV
3. https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing , "Shamir's Secret Sharing"
4. Bastianraschke, https://github.com/bastianraschke/pyfingerprint, pyfingerprint library