Steganalysis of PixelKnot

Alexandre Martens[1]
Supervisor: Dr Julio Hernandez-Castro[2]

CO880: Project and Dissertation
Total word count: 3560

September 10, 2015

[1] *University of Kent, Canterbury, United Kingdom, alexmartens@hotmail.fr*
[2] *University of Kent, Canterbury, United Kingdom, J.C.Hernandez-Castro@kent.ac.uk*

ACKNOWLEDGEMENT

I thanks Dr Julio Hernandez-Castro for the help he gave me in understanding the concepts of steganography, and for having dedicated a lot of time to answer my questions.

ABSTRACT

The purpose of this paper is to find a way to detect messages hidden in images using PixelKnot software. This software hides text into an image thanks to steganography. To do so, I implemented two articles. The first one, "Steganalysis of JPEG images: Breaking the F5 Algorithm" of Jessica Fridrich, Miroslav Goljan and Dorin Hogea. And the second one, "JPEG steganography detection with Benford's Law" of Panagiotis Andriotis, George Oikonomou and Theo Tryfonas. So I implemented their work to check if they could be applied to PixelKnot. In order to validate and evaluate their research, I developed steganographic tools which are able to analyse image files.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Over the years, people have sought to conceal information in order to communicate privacely. One method used is steganography. This art consists of hiding information in other information. These days, and particularly during the past two decades, with the rise of the Internet, steganography has adapted in digital formats, including images, which are greatly divided on the canvas. It is used by all people seeking to communicate messages discretely, as for example the army, or terrorist organisations.

I became interested in this project because it deals with many areas which are computers, computer security and development of software. Though most steganography that was totally unknown to me, has the principle always interest me.

In the first part, I will deal with PixelKnot and steganography software. This will introduce the notions of JPEG and algorithm that we'll develop in the second part. Then in the last section, we will end by implementation of it, results and the further work.

2 BACKGROUND

2.1 PixelKnot

PixelKnot is an android application which enables to messages in picture. The text hidden in the image by a way completely invisible to the human eye. The image can then be sent to a receiver who can extract the text thanks to the application. Users can take a picture or select an image on their phone and write a a short message. They can also add a password to encrypt the message to increase the security in case of detection. The receiver can read the message by opening the same picture with PixelKnot. To do that, the application uses the F5 algorithm which is resistant to visual and statistical attacks. (PixelKnot, 2013) However, suck a change leaves traces in the images that can be exploited.

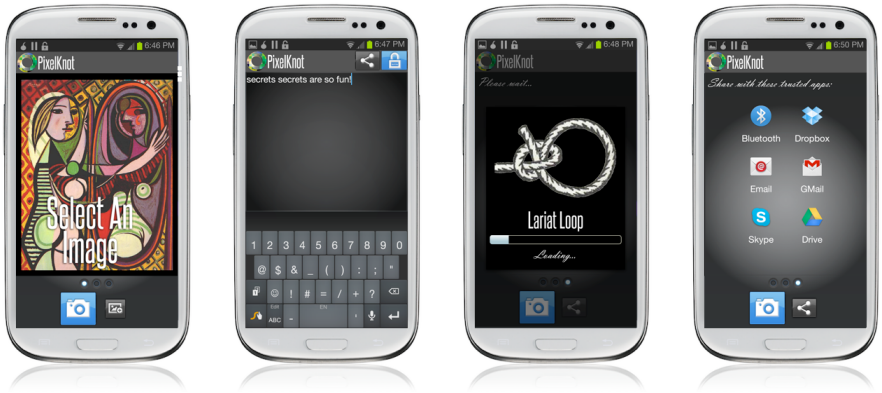

Figure 1: Screen about the use PixelKnot
from https://guardianproject.info/apps/pixelknot/.

2.2 Steganography

The most famous steganography technique known and easily implementable is LSB (Least Significant Beat).

This method involves the change of the least significant bit of the image pixels encoding. An image is an array consisting of a set of pixels. For each pixel, we code the color with three bytes: one for red, one for green and one for blue. Each byte indicates the intensity of the corresponding color, on a level of from 0 to 255. 255 corresponds to the native color. Move from the level N to N - n, where n is small enough to changes just a little bit the color, and this is precisely this that is based on the LSB method.

Takes a byte corresponding to one of the three colors of a pixel, for example 01101011. The last four bits, 1011 corresponds to the least significant bits. The idea is to replace these low order bits of information by those that one wishes to conceal. Either one byte of the image that hides 10011101 and a byte of the image that one wants to hide is 1001. The aim is to replace the low-weight bits of the image by significant bits of the image one wants to hide. Thus, we obtain the byte 01101001.

The problem with this technique is that have lot of weakness, can be discover easily if an image has been modified and can make the hidden message unreadable.

2.3 JPEG

Before discussing the exploitation of image, it is important to understand the functioning of the JPEG compression algorithm F5. Indeed, PixelKnot uses this algorithm and this one needs to convert image to Jpeg image to hide information inside.

JPEG is an acronym for (Joint Photographic Experts Group), born in 1982 from the merger of a group of experts and professionals in the field of imaging industry. Both groups came together to create a joint committee of experts of the photograph (JPEG). And it is this committee which gave its acronym to the open standard JPEG image compression, which was specified in 1991 and officially adopted in 1992, and all image files using this type of compression. (Wallace. G. K, 1991) Extensions to name the most common files for files employing JPEG compression are .jpg and .jpeg, however .jpe, .jfif .jif were also used. JPEG compression is divided into five steps:

1. Color processing and sub-sampling
2. Division into blocks
3. Implementation of the Discrete Cosine Transform (DCT).
4. Quantification of each block by a quantization matrix.
5. Entropy coding

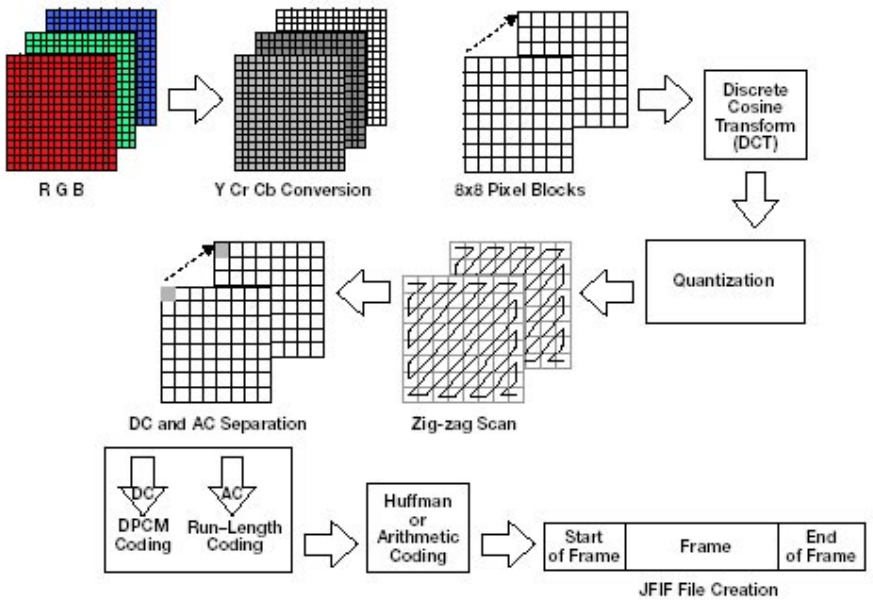

Figure 2: Steps of JPEG compression from http://www.eetimes.com/document.asp?doc_id=1225736.

2.3.1 *Color processing and sub-sampling.*

This first step converts the original image from its original color model (RGB usually) into the type of model chrominance / luminance YCbCr.

1. Luminance is the signal that determines the contrast values of an image, from the deepest black to the purest white.

2. Chrominance refers to the part of the signal which determines the color values of the image.

In this model, Y is the luminance information, and Cb and Cr are two chrominance information. Indeed, the best compression ratios are obtained with type of color coding luminance / chrominance because the human eye is quite sensitive to luminance (brightness) but less for chrominance (hue) of an image. To exploit this low sensitivity of the human eye to the chrominance, the principle of this operation is to reduce the size of several chrominance blocks in a single block.

2.3.2 *Division into blocks*

The image is divided into blocks of 64 (8 x 8) subpixels. The luminance information is not sub-sampled, each block is 8 x 8 pixels in the original image. Chrominance information, by cons correspond to the subsampling 8 x 8 in the original image.

2.3.3 *DCT transformation*

DCT (Discrete Cosine Transform) is a variant of the Fourier transform that is applied to each block. It considers a block as a real function of two variables and decomposes into a sum of cosine functions oscillating at different frequencies. This way, each block is expressed as frequency card and amplitudes unlike pixels and color coefficients. Thus the frequency value reflects the speed of change and the magnitude of this difference is associated each color change. The application of the DCT is theoretically an operation without loss of information; initial coefficients can be found by applying the "inverse DCT" to the result of DCT.

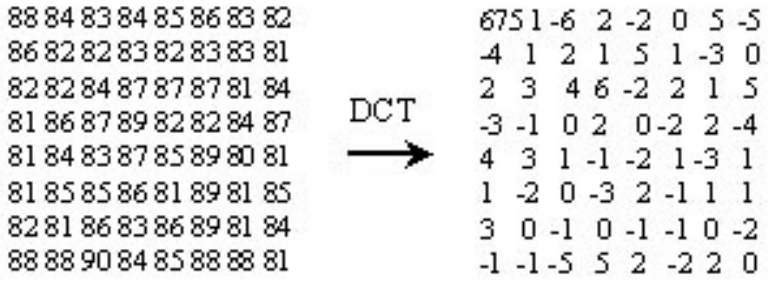

Figure 3: Transformation of original to DCT matrix from http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/mpeg1/dct.jpg

The results of a 64-element DCT transform are 1 DC coefficient and 63 AC coefficients. The DC coefficient represents the average color of the 8x8 region. The 63 AC coefficients represent color change across the block. Low-numbered coefficients represent low-frequency color change, or gradual color change across the region. High-numbered coefficients represent high-frequency color change, or color which changes rapidly from one pixel to another within the block. (Figure 3)

2.3.4 *Quantification*

It is during this step that most of information loss occurs and obviously where the most space id saved. The principle of this step is to divide the resulting coefficients matrix of the previous calculation (DCT) by another called quantization matrix containing 8x8 specifically chosen by the encoder. This aimes to reduce the high frequencies which the human eye is very insensitive. Quantification brings more coefficients to 0, those whose frequency is high and therefore the amplitude is low and keep only the important information. The frame of redundancy coefficients increases.

2.3.5 *Entropy coding*

Entropy coding is a special form of lossless data compression. It involves arranging the image components in a "zigzag" order employing run-length encoding (RLE) algorithm that groups similar frequencies together, inserting length coding zeros, and then using Huffman coding on what is left.

2.4 The F5 algorithm

The algorithm F5 is probably the first practical implementation of an embedding matrix technique steganography. It was invented by A. Westfeld and presented in 1999. The name comes from the previous algorithms developed by the author: F3 and F4. F5 inserts the message during JPEG compression, and uses the LSB of the DCT coefficients to perform the embedding matrix operation. The algorithm is secure against visual and statistical attacks as it adapts to the specific distribution of DCT coefficients.

The F5 algorithm. Westfeld is an algorithm on DCT coefficients non-zero quantized of an image (for JPEG compression), using the embedding matrix technique. To make sure against statistical attacks, it preserves the characteristics of the histogram of DCT coefficients of a natural image: when a bit has been change, the algorithm de-incremented the coefficient value. The only artefacts product is an increase in the number of coefficients to 0, however, this feature simply look like a JPEG compression with a lower quality factor. Also, to preserve the symmetry of the image, the odd and even negative positive coefficients have a value of 1 LSB, whereas even odd positive and negative coefficients have a value of 0 LSB. (Westfeld, 2001)

The algorithm presented by A. Westfeld takes 5 input parameters to insert message:

1. An image serving as a support (compressed or not).
2. A message to be inserted.
3. The quality factor to be used when quantizing.
4. A password to generate a random walk on the medium and the message.
5. A comment can be inserted into the header of the JPEG file.

It takes two input parameters to extract message:

1. A steganography image.
2. The password used when inserting.

2.5 Benford's Law

2.5.1 *The general Benford's law*

At the end of the nineteenth century, Newcomb finds uneven wear of logarithm tables (the first significant digit , the non-zero leftmost digit in the writing decimal number): The number 1 is most commonly used as the number 2, itself more used than the number 3 ... It establishes a statistical observation of this law. This law was "given to date" by Benford in the late 1930s and takes its name. Benford's law reflects the frequency of occurrence of the first digit of a number in a series of numbers. It is a distribution such that the probability of occurrence of a value "d" in the first position of a number is :

$$p(n) = \log_{10}(1 + \frac{1}{n}), n = 1, ..., 9. \quad (1)$$

Figure 4 gives a graphical representation. In 1972, Varian stressed the potential usefulness of this law to detect the presence of possible fraud in databases. The idea of using this law in accounting for detecting a risk of accounting fraud emerged in the late 1980s in a research report of a New Zealand researcher Charles Carslaw.

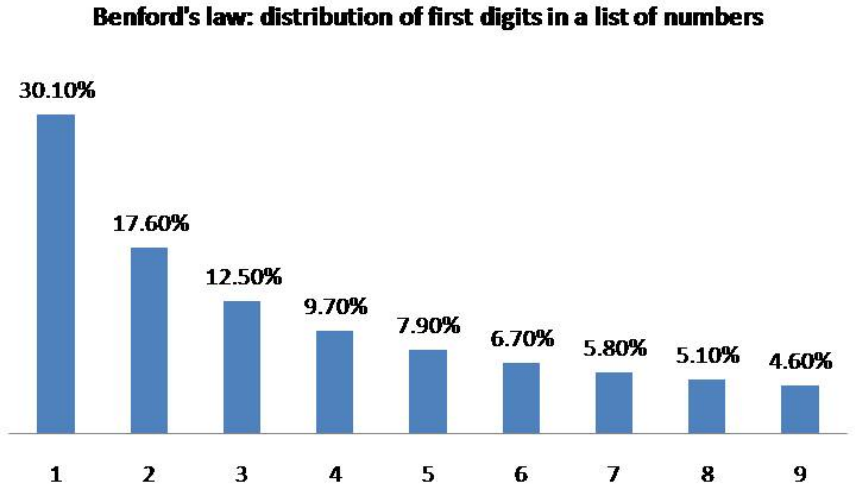

Figure 4: Benford's Law
from https://poliscizurich.files.wordpress.com/2010/03/benford3.jpg

2.5.2 *Benford's Law for image forensic*

In 2007 Fu and al. presented with a new approach to analysis image using the Benford's law and studied the behaviour of the Jpeg block coefficients of an image in profundity (Fu et al. 2007.) In this work he has some conclusions on the validity of Benford's law in the most meaningful numbers of DCT coefficients (before quantification) of the 8x8 pixel blocks of an grey image; coefficients DC is excluded from the research. After the computation of the apparition of meaningful numbers of the coefficients DCT in a series of images, their average division was gotten. The meaningful numbers of coefficient DCT congruent quite well with the law of Benford. Carrying out thorough experiments on the same set of images, the authors computed also the average division of the first numbers of the quantified DCT coefficients under different factor compression of quality. The results show that the distributions of these coefficients follow also a tendency logarithmic. A comparison between the average distributions that they got compression and distributions anticipated law for every quality Benford revealed that quantified coefficients do not follow the rule of the equation (1) in a same way that coefficients DCT make. However, there exists also a law logarithmic behind the distribution of the first numbers of the quantified coefficients DCT. The model that they offered east is describes by the following equation:

$$p(n) = N.\log_{10}(1 + \frac{1}{s + n^q}), n = 1, ..., 9. \quad (2)$$

N, s and q are parameters which describe different compression quality factors.

Q-factor	Model Parameters		
	N	q	s
100	1.608	1.605	0.0702
90	1.25	1.585	-0.405
80	1.344	1.685	-0.376
75	1.396	1.731	-0.3549
70	1.434	1.766	-0.339
60	1.514	1.843	-0.3114
50	1.584	1.909	-0.2875

The F5 default algorithm uses a quality factor of 80.

3 IMPLEMENTATION

During this project, I implemented the attack presented above. To do this, I developed a program in C++ and another in Python. For different attacks, it is crucial to recover the DCT coefficients of the image to be analysed. To do this, I used the "jpeglib" library that have available only on c ++.

```
int main() {
  const char* filename = "C:/Users/Alexandre/Dropbox/kent
  /Project_Research/project/pictures/stego/pixelknot-boat.jpg";

  FILE * infile;
  struct jpeg_decompress_struct srcinfo;
  struct jpeg_error_mgr srcerr;

  if ((infile = fopen(filename, "rb")) == NULL) {
    fprintf(stderr, "can't open %s\n", filename);
    return 0;
  }

  srcinfo.err = jpeg_std_error(&srcerr);
  jpeg_create_decompress(&srcinfo);
  jpeg_stdio_src(&srcinfo, infile);
  (void) jpeg_read_header(&srcinfo, FALSE);

  //coefficients
  jvirt_barray_ptr * src_coef_arrays =
        jpeg_read_coefficients(&srcinfo);
  read(srcinfo, src_coef_arrays);

  jpeg_destroy_decompress(&srcinfo);
  fclose(infile);
  return 0;
}
```

This feature will allow to read the coefficients contained in the images and to write into a file.

3.1 Fridrich's Implementation

The attack on the F5 algorithm was presented in 2002 by J. Fridrich, M. Goljan, D. Hogea. The attack is based on the ability to reconstruct an estimate of the histogram of the image before insertion of the message. Then, an analysis of the differences between the estimate and the intercepted image allows to derive an approximation of the probability of changing a coefficient, and therefore the size of the message inserted.

The attack presents itself in two points:

- Estimating the histogram of the image before inserting data.
- Deducting this histogram message length inserted.

1. In order to build the estimation of image histogram intercepted before insertion, the image is decompressed in space, offset by 4 pixels in both directions (horizontally and vertically), and then re-compressed using the same quality factor before insertion. This shift can "break" the block structure of the DCT coefficients to obtain an estimate of their values before quantization. In addition, a low-pass filter is applied immediately after the shift, and to mitigate the effects of blocks due to quantization.
2. To estimate the length of the message inserted, the authors first calculate the probability that a non-zero AC coefficient is changed. From this we can get the size of the message inserted. This requires two calculations with the values provided by the two histograms.

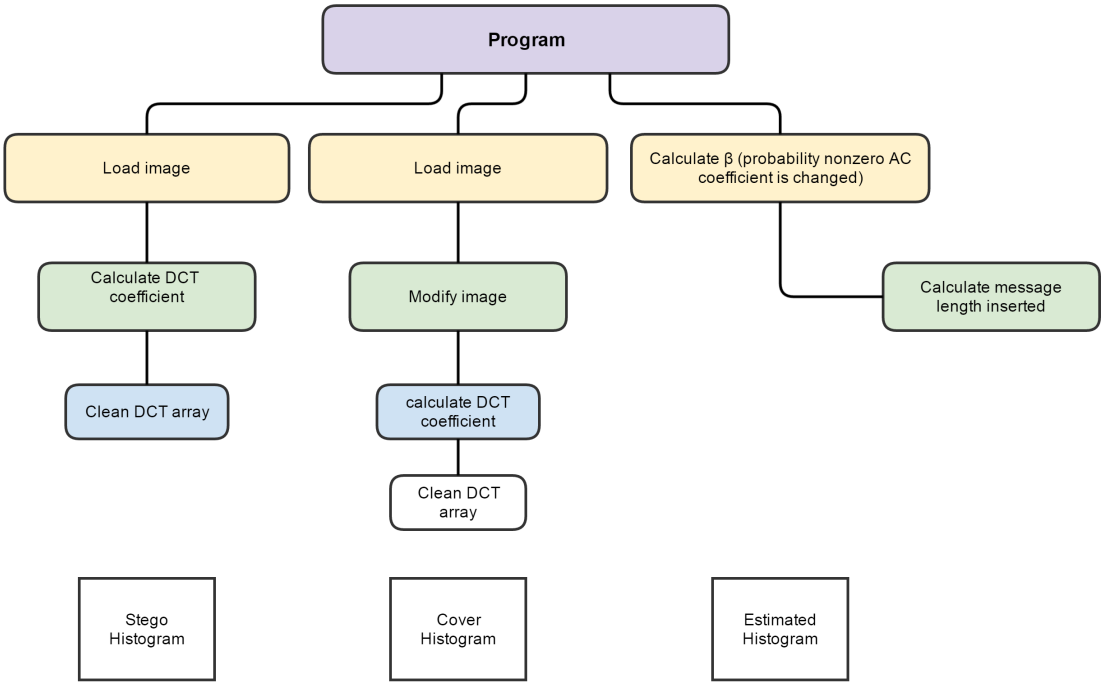

Figure 5: Diagram Source Code

Firstly, I recovered the DCT coefficients stores in the file to fill them in a dictionary. This gave me a Histogram of stego image. In this code, I open the text file and I recovered all the lines to store them in "content". A line (an 8x8 DCT block in string format) in each box. Then I go through each cell of the table and transforms each row in a table where each cell contains a coefficient. Finally, I filled "dctList" that becomes a two dimensional table.

```
1  def dctFridrich(dctFile):
2    dctList = []
3    with open(dctFile) as f:
4      content = f.readlines()
5    for i in range(0, len(content)):
6      z = 0
7      block = content[i].split()
8      dctList.append([])
9      for x in range(0,8):
10       dctList[i].append([])
11       for y in range(0,8):
12         dctList[i][x].append(block[z])
13         z += 1
14   return dctList
```

It then remains to estimate the histogram of the image before the insertion of information. To do that, I made several changes over the image. In this code, I open the JPEG image. I get the quantification table. I converted the image to bmp is an uncompressed format. I resize the image by cutting 4 pixels on each side. And then I save this image in JPEG format with the same quantization table.

```
def modifImage(path, name, type):
  jpeg = Image.open(path + name + type)
  quantization = jpeg.quantization
  jpeg.save(path + name + ".bmp", "BMP",  bmp_rle=True)
  bmp = Image.open(path + name + ".bmp")
  width, height = bmp.size
  jpeg_modif = bmp.crop((4, 4, width-4, height-4))
  jpeg_modif.save(path + name + "_modif.jpg", "JPEG",
    qtables=quantization)
```

Then I did the same process as described above. It gave me the estimate histogram of the cover image. Now, it's possible to perform both calculations to estimate the size of the hidden message.

In order to estimate the length of the message inserted, the authors first calculated the probability beta that an non-zero AC coefficient is altered. Let H, h, $\hat{h}$ the respective histograms of stego image, the image before inserting data, and estimation of the image. H(i) represents the number of coefficients for the stego image whose absolute value is equal to i. In addition, for a histogram X, we denote

$$X_{k,l}, 1 \leqslant k, l \leqslant 8 \tag{3}$$

, the histogram of the coefficient lying instead (k, l) in the 8x8 block (eg $H_{0,0}$ is the coefficient histogram of the DC stego picture). And $\hat{H}$ the histogram of the stego image. Beta will minimize the differences between the estimated and the histogram H intercepted H. The calculations are made for d = 0 and d = 1, these two values being the most affected in integration through F5 algorithm. In addition, since we have 8x8 possible results for beta (one result for each pair (h, k)), the authors choose to take the average of the results for the cases (2,1), (1,2) , (2.2), corresponding to low frequency AC coefficients. This is explained by the fact that the offset 4 pixels on both sides leads to discontinu-

ities in the middle of each block, so undesired high frequencies.

$$\beta_{k,l} = \frac{\hat{h}_{k,l}(1)[H_{k,l}(0) - \hat{h}_{k,l}(0)] + [H_{k,l}(1) - \hat{h}_{k,l}(1)][\hat{h}_{k,l}(2) - \hat{h}_{k,l}(1)]}{\hat{h}^2_{k,l}(1) + [\hat{h}_{k,l}(2) - \hat{h}_{k,l}(1)]^2} \tag{4}$$

and:

$$\beta = \frac{\beta_{2,1} + \beta_{1,2} + \beta_{2,2}}{3} \tag{5}$$

In this code, I go through the two arrays. (that of the image which has not been changed and the other). For each I count the number of coefficients equal to 0, 1 or 2 in the precise position of the blocks. Then I calculate beta thanks to the previous equation.

```
def Beta(mapDct, mapDctModif):
  beta = np.empty((8,8))
  H_0 = np.zeros((8,8))
  H_1 = np.zeros((8,8))
  H_2 = np.zeros((8,8))
  hbarre_0 = np.zeros((8,8))
  hbarre_1 = np.zeros((8,8))
  hbarre_2 = np.zeros((8,8))

  for i in range(0,len(mapDct)):
    for k in range(0,8):
      for l in range(0,8):
        if (mapDct[i][k][l] == '0'):
          H_0[k][l] += 1
        elif (mapDct[i][k][l] == '1' or mapDct[i][k][l] == '-1'):
          H_1[k][l] += 1
        elif(mapDct[i][k][l] == '2' or mapDct[i][k][l] == '-2'):
          H_2[k][l] += 1

  #SAME FOR MAPDCTMODIF

  for k in range(0,8):
    for l in range(0,8):
      a = ( hbarre_1[k][l] * (H_0[k][l] - hbarre_0[k][l]) ) +
        ( (H_1[k][l] - hbarre_1[k][l]) * (hbarre_2[k][l]
        - hbarre_1[k][l]) )
      b = ( hbarre_1[k][l] * hbarre_1[k][l] ) + ( hbarre_2[k][l]
        - hbarre_1[k][l] ) * ( hbarre_2[k][l] - hbarre_1[k][l] )
      beta[k][l] = a/b

  betaAverage = (beta[0][1] + beta[1][0] + beta[1][1]) / 3
  return betaAverage
```

From beta (the probability of non-zero AC coefficient change), we can get the message size inserted. Suppose that n is the total number of changes introduced by F5 algorithm, s the shrinkage, P the total number of non-zero coefficients AC and Ps probability of selecting a coefficient that may lead to shrinkage. We have:

$$P_s = \frac{h(1)}{P} \leftrightarrow s = nP_s \tag{6}$$

Also, we have:

$$n = m + s \leftrightarrow m = n(1 - P_s) \leftrightarrow m = n(1 - \frac{h(1)}{P}) \tag{7}$$

And:

$$n = \beta P \tag{8}$$

Hence the formula:

$$M = \frac{2^p}{2^p - 1} p\beta(P - h(1)) \tag{9}$$

In this code, I go through the table of the image having been intersected. I count the number of coefficients equals to 1 and number not equal to 0. Then I calculate the equation stated above.

```
def MsgLength(beta, mapDctModif):
    nbAcNoNull = 0
    nbCoefEgal1 = 0

    for y in range(8):
        for x in range(8):
            nbCoefEgal1 += mapDctModif[y][x].count('1')
            nbCoefEgal1 +=  mapDctModif[y][x].count('-1')
            for item in mapDctModif[y][x]:
                if (item != '0' and item != '-0'):
                    nbAcNoNull += 1
    k = (math.log(1 / beta) + 1) / math.log(2)
    capacity = np.around(nbAcNoNull - 0.51 * nbCoefEgal1)
    length = (math.pow(k,2) / (pow(k,2) - 1)) * k * beta
      * capacity
    return length
```

3.2 Benford's Law Implementation

To begin, we must retrieve the DCT coefficients that are saved in the file and put it into a two dimensions table. The DC coefficient and the value zeros are not taken into consideration on the Benford's Law. So, this function removes these two values. And I put the other on dctList.

```
1  def dctBenford(dctFile):
2    dctList = []
3    with open(dctFile) as f:
4      content = f.readlines()
5
6    for i in range(0, len(content)):
7      block = content[i].split()
8      del block[0]
9      while block.count('0') > 0:
10       block.remove('0')
11     dctList += block
12
13   return dctList
```

Now, the program must extract the first significant digits of the quantized DCT coefficients. Figure 5 illustrates this concept.

1.3e+3	4.7	3.2	-0.19	0.25	-0.5	-4.5	5.6
7.9	-0.7	0.6	-4.9	1.9	2.9	-3.7	3.3
-5.0	-0.2	-1.6	1.7	-0.6	-0.4	1.8	-2.2
2.3	1.1	1.7	0.9	-0.7	-1.3	0.2	1.1
-1.0	-1.2	-0.3	-1.4	1.7	1.1	-1.4	-0.6
1.2	0.4	-1.8	-0.1	-2.0	-0.7	1.6	0.7
-1.7	0.2	3.1	1.6	1.6	-2.2	-1.2	-0.9
1.3	-0.4	-2.4	-1.6	-0.8	1.9	0.5	0.6

(a)

	4	3	1	2	5	4	5
7	7	6	4	1	2	3	3
5	2	1	1	6	4	1	2
2	1	1	9	7	1	2	1
1	1	3	1	1	1	1	6
1	4	1	1	2	7	1	7
1	2	3	1	1	2	1	9
1	4	2	1	8	1	5	6

(b)

Figure 6: First digit of 8x8 blockDCT coefficient

Thereafter, it only remains to make a percentage of appearance of digits in order to compare them with those of Benford's law. This translates this thanks to two functions that follow. Find_leading_number will return the first number between 1 and 9. And calc_firstdigit going through the array, called the first function for each number and then calculated the percentage of occurrence of each digit.

```
1  def find_leading_number(line):
2    numbers = "123456789"
3    line = str(line)
4    index = len(line)
5    for i in range(0, index):
6      if line[i] in numbers:
7        return int(line[i])
8    return 0
9
10 def calc_firstdigit(dctList):
11   fdigit = [str(find_leading_number(value))
12     for value in dctList]
13
14   distr = [fdigit.count(str(i))/float(len(dctList))*100
15     for i in xrange(1, 10)]
16   return distr
```

4 RESULTS AND ANALYSIS

4.1 Fridrich's Results

Unfortunately, as shown in the table below for some examples, the values for this attack are completely absurd. Beta (who represents the probability that a non-zero AC coefficient is been changed) is more important in a pure image as a modified image. Moreover, for pure images, it gives some too important messages lengths to be ignored. And finally, for the same message in the pictures, it gives too different lengths.

Name	Beta(Pure)	Beta(Stego)	MsgLength(Pure)	MsgLength(stego)
Boat	1.29	0.18	2185.10	153.20
Cave	0.08	0.09	48.31	16.67
Field	1.49	0.70	-63.91	55.86
Hearth	0.53	0.60	245.39	289.60
House	0.00	0.01	1.27	3.94
Vine	1.55	0.45	-19.17	-10.92

This may be due to a security of PixelKnot. But I think rather that it is a mistake on my part. Or misunderstanding of a concept, or a programming error.

4.2 Benford's Law Results

In Figure 7, we can see the first digits curve for an "pure image", without any modification. We can see that it follows perfectly the Benford's curve. However, in Figure 8, we can see the two curves with the same image modified with PixelKnot. We can see that the distribution of digits does not follow Benford's law. This brings into focus that changes in the image disturbs the DCT coefficients.

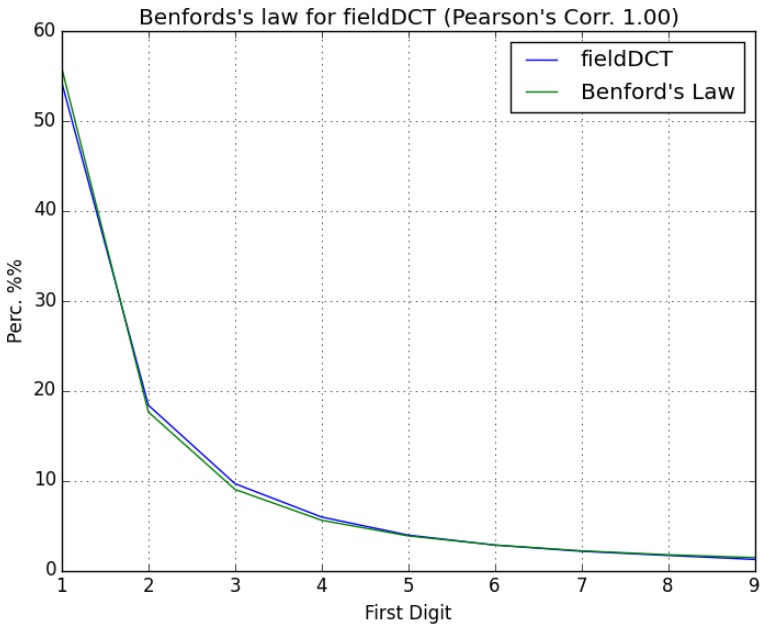

Figure 7: First Digits of pure image (without modification)

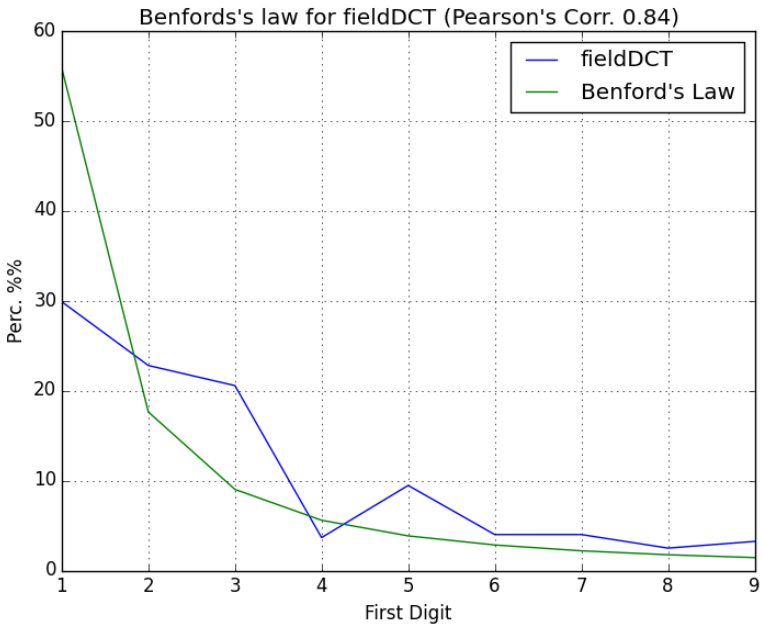

Figure 8: First Digits with PixelKnot modification

First Digits	Benford's Law	Stego value	Difference
1	55.83	29.86	-25.97
2	17.61	22.78	5.17
3	9.01	20.54	11.53
4	5.58	3.66	-1.93
5	3.85	9.45	5.6
6	2.83	3.99	1.16
7	2.19	3.99	1.8
8	1.75	2.49	0.74
9	1.44	3.23	1.8

On a set of 48 images, here is a table summarizing the differences with the digits 1 to 9 combined.

Differences	0-9	10-19	20-29	30-39	40-49	51+
Nbr of images	4	6	11	4	9	13

We can see that in most images, the deviations of the digits are significant for images having been modified. This is proof that it is possible to detect if an image has been modified by Pixelknot.

5 CONCLUSIONS

To conclude, even if PixelKnot says impervious to statistical attacks, we can see that it did not protect against attacks using Benford's law. It is possible to detect if an image has been modified with steganography thanks to PixelKnot.

Finally, this project was for me something positive, which I learned some interesting things, including steganalysis, compression and decompression jpeg and image processing, which was a completely unknown universe but exciting. I was also confronted with the study of more recent research articles, and writing documents with LaTeX, which is a plus for my career.

You can find my source code, images used, DCT files and the results of my analysis on my github. At the following address: https://github.com/twinz/MscStegano

6 FUTURE RESEARCH

Even if the results of the Benford's law attack were encouraging, they would use more images to have a better representation. Indeed, 48 pictures are really few. Unfortunately, I was not able to automate this android application on windows using an emulator. When I get a picture of my computer or google photo, the application stopped working. Moreover, they would be interesting to understand why the results of the Fridrich's attack are not good. This is probably a problem of comprehension or programming.

7 BIBLIOGRAPHY

PixelKnot:

1. Mark (2013). Our Newest App: PixelKnot. [Online] Guardian Project Website. Availible from : https://guardianproject.info/2013/07/18/pixelknot/. [Accessed 14th February 2015]

Jpeg:

2. Wallace, G. K. (1991). The JPEG Still Picture Compression Standard. [Online]. Available from : http://www.cis.temple.edu/ vasilis/Courses/CIS750/Papers/doc_jpeg_c_5.pdf. [Accessed 24th February 2015]
3. Cabeen, K. & Gent, P. (unknown). Image Compression and the Discrete Cosine Transform. [Online]. Availible from : http://www.lokminglui.com/dct.pdf. [Accessed 28th February 2015]

F5 algorithm:

4. Westfeld, A. (2001). F5—A Steganographic Algorithm High Capacity Despite Better Steganalysis. [ONLINE] Available from: http://citeseerx.ist.psu.edu/viewdoc/ download?doi=10.1.1.115.3651&rep=rep1&type=pdf. [Accessed 26 June 15].

Breaking F5 algorithm:

5. Fridrich, J., Goljan, M. & Hogea, D. (2002). Steganalysis of JPEG Images: Breaking the F5 Algorithm. [Online]. Available from : http://ws2.binghamton.edu/fridrich/Research/f5.pdf. [Accessed 14th February 2015]
6. Aboalsamh, H. A., Mathkour, H. I., Mursi, M. F. M., & Assassa, G. M. R. (2008). Steganalysis of JPEG Images: An Improved Approach for Breaking the F5 Algorithm. [Online]. Available from : http://www.wseas.us/e-library/conferences/

2008/crete/Computers/160-computers.pdf. [Accessed 24th February 2015]

Benford's Law:

7. Andriotis, P., Oikonomou, G. & Tryfonas, T. (2013). JPEG steganography detection with Benford's Law. [Online]. Available from : http://fortoo.eu/m/page-media/4/jpeg-steganography.pdf. [Accessed 20th March 2015]

8. Dongdong, F., Yun, Q. & Wei, S. (2007). A generalized Benford's law for JPEG coefficients and its applications in image forensics. [Online].
Available from : https://web.njit.edu/ shi/PaperDownload/ forensics/SPIE-BF-Law.pdf. [Accessed 20th March 2015]

8 APPENDIX

8.1 Source Code

You can find my source code to the following address: https://github.com/twinz/Msc
Otherwise, here is my source code:

```
1  ##########################################################################
2  #                          ALEXANDRE MARTENS
3  #          MSc Project. Breaking PixelKnot (Breaking F5 Algo)
4  #          Implementation of Fridrich Attack and Benford Attack
5  ##########################################################################
6
7  import cv2
8  import numpy as np
9  import matplotlib.pyplot as plt
10 from PIL import Image
11 import math
12 import os
13
14 ##########################################################################
15 #                       BREAKING THE F5 ALGORITHM
16 #                           Fridrich Attack
17 ##########################################################################
18
19 def modifImage(path, name, type):
20     jpeg = Image.open(path + name + type)
21     quantization = jpeg.quantization
22     jpeg.save(path + name + ".bmp", "BMP",  bmp_rle=True)
23     bmp = Image.open(path + name + ".bmp")
24     width, height = bmp.size
25     jpeg_modif = bmp.crop((4, 4, width-4, height-4))
26     jpeg_modif.save(path + name + "_modif.jpg", "JPEG", qtables=quanti
27
28 def Beta(mapDct, mapDctModif):
29     beta = np.empty((8,8))
30     H_0 = np.zeros((8,8))
31     H_1 = np.zeros((8,8))
32     H_2 = np.zeros((8,8))
33     hbarre_0 = np.zeros((8,8))
34     hbarre_1 = np.zeros((8,8))
35     hbarre_2 = np.zeros((8,8))
36
37     for i in range(0,len(mapDct)):
38         for k in range(0,8):
39             for l in range(0,8):
```

```
40                  if (mapDct[i][k][l] == '0'):
41                      H_0[k][l] += 1
42                  elif (mapDct[i][k][l] == '1' or mapDct[i][k][l] == '-1
43                      H_1[k][l] += 1
44                  elif(mapDct[i][k][l] == '2' or mapDct[i][k][l] == '-2'
45                      H_2[k][l] += 1
46
47      for i in range(0,len(mapDctModif)):
48          for k in range(0,8):
49              for l in range(0,8):
50                  if (mapDctModif[i][k][l] == '0'):
51                      hbarre_0[k][l] += 1
52                  elif (mapDctModif[i][k][l] == '1' or mapDctModif[i][k]
53                      hbarre_1[k][l] += 1
54                  elif(mapDctModif[i][k][l] == '2' or mapDctModif[i][k][
55                      hbarre_2[k][l] += 1
56
57      for k in range(0,8):
58          for l in range(0,8):
59              a = ( hbarre_1[k][l] * (H_0[k][l] - hbarre_0[k][l]) ) +  (
60              b = ( hbarre_1[k][l] * hbarre_1[k][l] ) + ( hbarre_2[k][l]
61              beta[k][l] = a/b
62
63      betaAverage = (beta[0][1] + beta[1][0] + beta[1][1]) / 3
64      return betaAverage
65
66  def MsgLength(beta, mapDctModif):
67      nbAcNoNull = 0
68      nbCoefEgal1 = 0
69
70      for y in range(8):
71          for x in range(8):
72              nbCoefEgal1 += mapDctModif[y][x].count('1')
73              nbCoefEgal1 +=  mapDctModif[y][x].count('-1')
74              for item in mapDctModif[y][x]:
75                  if (item != '0' and item != '-0'):
76                      nbAcNoNull += 1
77      k = (math.log(1 / beta) + 1) / math.log(2)
78      capacity = np.around(nbAcNoNull - 0.51 * nbCoefEgal1)
79      length = (math.pow(k,2) / (pow(k,2) - 1)) * k * beta * capacity
80      return length
81
82  ###########################################################################
83  #                              FOR FRIDRICH
84  #                  Read DCT on file , Put on a 2D list
85  ###########################################################################
```

```
86
87  def dctFridrich(dctFile):
88      dctList = []
89      with open(dctFile) as f:
90          content = f.readlines()
91      for i in range(0, len(content)):
92          z = 0
93          block = content[i].split()
94          dctList.append([])
95          for x in range(0,8):
96              dctList[i].append([])
97              for y in range(0,8):
98                  dctList[i][x].append(block[z])
99                  z += 1
100     return dctList
101
102 ##########################################################################
103 #                                FOR BENFORD
104 #       Read DCT on file, del DC coeff and 0. Put on a list
105 ##########################################################################
106
107 def dctBenford(dctFile):
108     dctList = []
109     with open(dctFile) as f:
110         content = f.readlines()
111
112     for i in range(0, len(content)):
113         block = content[i].split()
114         del block[0]
115         while block.count('0') > 0:
116             block.remove('0')
117         dctList += block
118
119     return dctList
120
121 ##########################################################################
122 #                                BENFORD
123 ##########################################################################
124
125 def benford_law():
126     N = 1.344
127     S = -0.376
128     q = 1.685
129
130     return [(N * math.log10(1 + (1 / ( S + math.pow(i, q)))))*100.0 fo
131
```

```
132 def find_leading_number(line):
133     numbers = "123456789"
134     line = str(line)
135     index = len(line)
136     for i in range(0, index):
137         if line[i] in numbers:
138             return int(line[i])
139     return 0
140 
141 def calc_firstdigit(dctList):
142     fdigit = [str(find_leading_number(value)) for value in dctList]
143 
144     distr = [fdigit.count(str(i))/float(len(dctList))*100 for i in xran
145     return distr
146 
147 def pearson(x,y):
148     nx = len(x)
149     ny = len(y)
150     if nx != ny: return 0
151     if nx == 0: return 0
152     n = float(nx)
153     meanx = sum(x)/n
154     meany = sum(y)/n
155     sdx = math.sqrt(sum([(a-meanx)*(a-meanx) for a in x])/(n-1) )
156     sdy = math.sqrt(sum([(a-meany)*(a-meany) for a in y])/(n-1) )
157     normx = [(a-meanx)/sdx for a in x]
158     normy = [(a-meany)/sdy for a in y]
159     return sum([normx[i]*normy[i] for i in range(nx)])/(n-1)
160 
161 ##############################################################################
162 #                                    PLOT
163 ##############################################################################
164 
165 # FOR BENBORD
166 def plot_comparative(aset, bset, dataset_label):
167     aset = [0] + aset
168     bset = [0] + bset
169     plt.axis([1, 9, 0, 60])
170     plt.plot(aset, linewidth=1.0)
171     plt.plot(bset, linewidth=1.0)
172     plt.xlabel("First Digit")
173     plt.ylabel("Perc. %%")
174     plt.title("Benfords's law for %s (Pearson's Corr. %.2f)" % (dataset
175     plt.legend((dataset_label, "Benford's Law"))
176     plt.grid(True)
177     return plt.show()
```

```
178
179 if __name__ == "__main__":
180
181     ## TEST  BENFORD
182     file = open("result.txt", "w")
183     path = "C:\Users\Alexandre\Dropbox\kent\Project_Research\MscStegan
184     tab = {}
185     for picture in os.listdir(path):
186         name = os.path.splitext(picture)[0]
187         bendordLaw = benford_law()
188         dctListPure = dctBenford(path + picture)
189         pure = calc_firstdigit(dctListPure)
190         dctListStego = dctBenford("C:\Users\Alexandre\Dropbox\kent\Pro
191         stego = calc_firstdigit(dctListStego)
192         i = 0
193         diff = 0
194         file.write("Picture: " + name + "\n")
195         file.write("First Digits\tDeviations(pure)\tDeviations(stego)\
196         for value in bendordLaw:
197             file.write(str(i + 1) + "\t\t\t\t" + str(round(pure[i] - b
198             diff += abs((pure[i] - bendordLaw[i]) - (stego[i] - bendor
199             i += 1
200         tab[name] = round(diff,2)
201     print tab
202
203     # TEST  FRIDRICH
204     dctArray = dctFridrich("C:\Users\Alexandre\Dropbox\kent\Project_Re
205     dctArray2 = dctFridrich("C:\Users\Alexandre\Dropbox\kent\Project_R
206     beta = Beta(dctArray, dctArray2)
207     print "Beta = " + str(beta)
208     msgLength = MsgLength(beta, dctArray2)
209     print "msgLength = " + str(msgLength)
```