Creating a Truly Random Number Generator for Seeding OpenSSL Pseudo-Random Number Generation Using Images as Input

https://github.com/dswynne/Image_Entropy

Jonathan Garner and David Wynne

ENEE408G

Submitted 4/19/20

Table of Contents

I. Abstract

In encryption it is important to generate unpredictable or random encryption keys so that a malicious party cannot decrypt the data that is being sent. Computers cannot generate truly random data themselves and instead rely on pseudo-random number generators (PRNGs) that use complex enough algorithms that it is exceptionally computationally expensive for a malicious party to decrypt the data without the decryption key [1]. A truly random number generator (TRNG) makes the computational costs even larger and is an added layer of security on top of existing algorithms. The goal of this project was to develop an image processing algorithm that converted an input image into a cryptographic seed for OpenSSL's PRNG. Randomness is a concept discussed heavily in this paper. For our algorithm to be considered truly random each new image inputted into it must produce a truly random seed.

II. Introduction

A. Inspiration

The inspiration for this project was Cloudflare's modern implementation of LavaRand which uses a wall of lava lamps in their main offices [2, 3]. For Cloud Fare they have a confined area in which they rely on the natural entropy of lava lamps to gain their random source of data. For this project, the challenge was removing all constraints of image size, content, etc. and still creating a source of truly random data.

B. OpenSSL

At a high level the algorithm developed for this project is a piece in the larger system that is OpenSSL's encryption algorithms. To better understand how this project fits into it is best to use a flow graph.

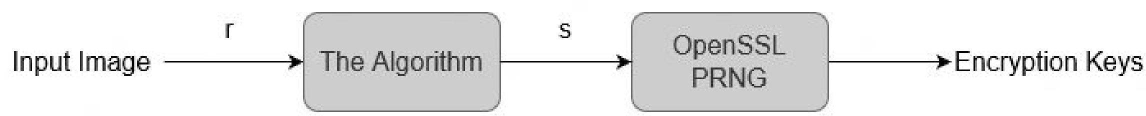

Figure 1. Block diagram outlining the algorithm's input into the OpenSSL system. Where r are the three $n \times m$ matrices of the RGB color channels (a single $n \times m$ for grayscale). The output of the algorithm s is a k long bit string that is used to seed the OpenSSL PRNG.

OpenSSL is an open source implementation of the Secure Socket Layer (SSL)/Transport Security Layer (TLS) protocol. There are four primary components used in SSL [4].

- Symmetric key (secret key) encryption
- Asymmetric key (public key) encryption
- Message Digests and digital signatures
- Certificates

The purpose of this project was strictly focused on key generation. Because of this digital signatures and certificates will not be discussed in detail.

B.1. Asymmetric Key Encryption

Asymmetric encryption uses a public and private key so that there is a lower chance of a malicious party decrypting the data. A data transfer can only be decrypted using both the sender's public key and the receiver's private key. This means that as long as the private key is kept secure, and unable to be guessed, the asymmetric encryption is secure. [4]

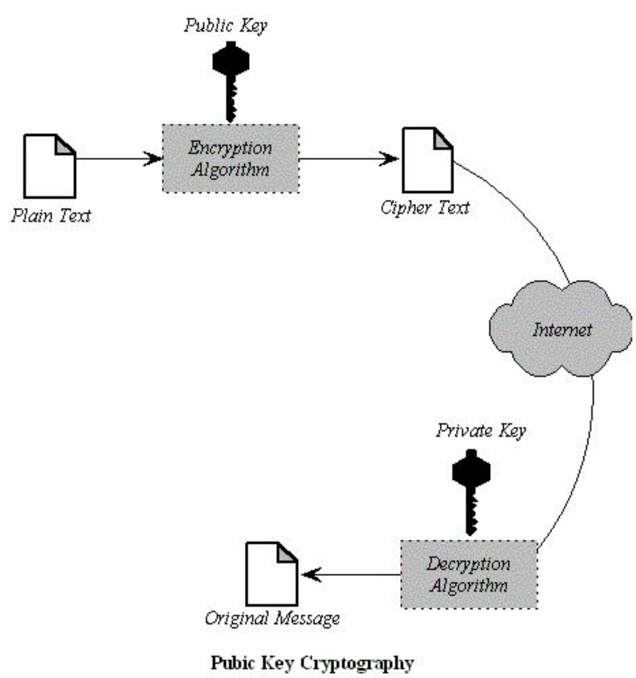

Figure 2. Standard flowchart for asymmetric encryption. [4]

The most common asymmetric encryption algorithm is Rivest–Shamir–Adleman (RSA) which uses large prime numbers to generate a public and private key. The advantage of this algorithm is that it is difficult to factor large composite prime numbers. [5]

B.2. Symmetric Encryption

Symmetric encryption only has one key that both the sender and receiver need access to to be able to encrypt and decrypt the message. This means that the sender and receiver need to agree on the symmetric key through a secure means of communication or they risk a malicious party intercepting the symmetric key. [4]

In SSL a process called the SSL Handshake uses asymmetric encryption keys to securely transfer symmetric encryption keys. With the symmetric encryption key securely transferred to both parties data can then be transferred using symmetric encryption. [6]

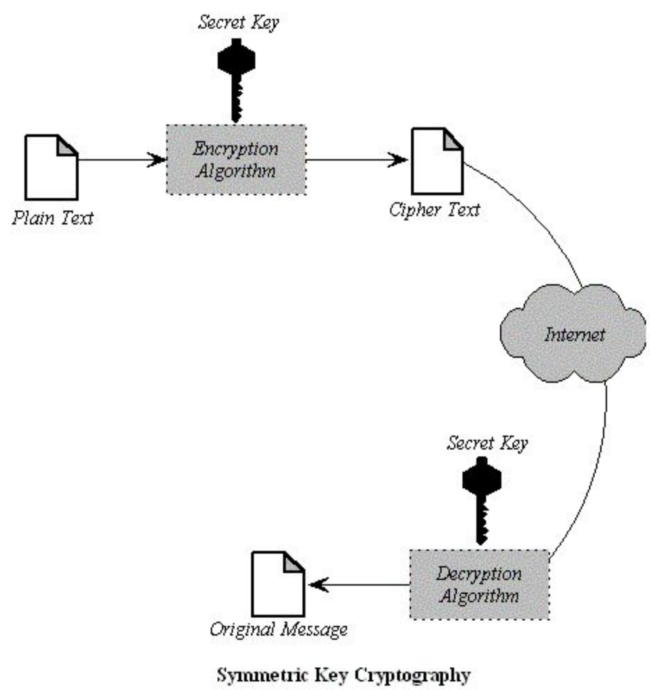

Figure 3. Standard flowchart for symmetric encryption. [4]

The most commonly used symmetric encryption algorithm is the Advanced Encryption Standard (AES). AES is a symmetric block cipher meaning it is made up of a deterministic algorithm that operates on fixed-length groups of bits or blocks. The symmetric label means that the same key is used for encryption and decryption. AES is specifically made up of three block ciphers, AES-128, AES-192, and AES-256, which use keys of varying bit length to encrypt and decrypt blocks of messages. Each block cipher performs multiple operations on the data arrays including substitution using a substitution table, shifting data rows, and mixing columns. Multiple rounds of transformations are completed within each block cipher. [7]

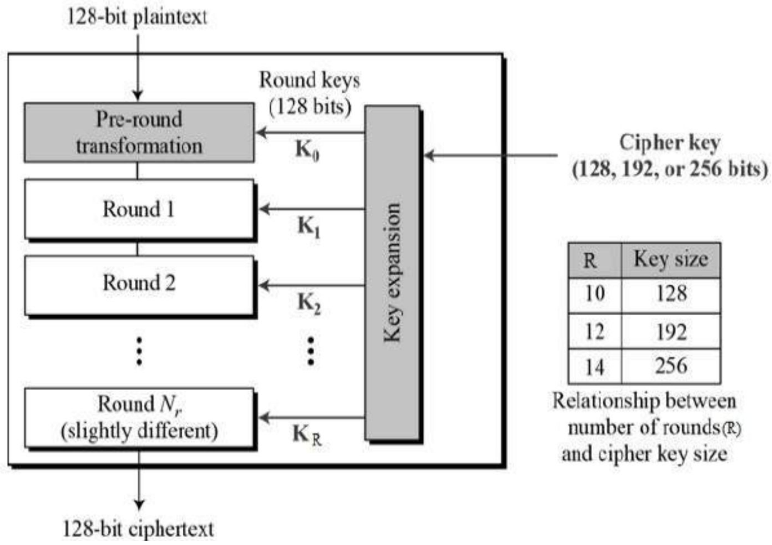

Figure 4. Schematic of AES Structure for a single cipher block. N rounds are performed in the cipher. The cipher key length corresponds to the current cipher block. In the table on the right, R corresponds to the number of rounds for a certain key size. [8]

B.3. Seeding

OpenSSL's PRNG handles the actual generation of these keys. The PRNG needs to be seeded with a random number so that it can continuously output random RSA and AES keys. For the PRNG to generate sufficiently random numbers the seed needs to be 256 bits long. Our goal was to design an algorithm that took an input image and generated this seed. The seed itself needs to be truly random for the PRNG to become a TRNG. [1]

III. The Algorithm

Due to the nature of data JPEG compression an image stored in this format is not a truly random source of data [9]. Because of this, numerous image processing and bit manipulation techniques were used to aid in creating a TRNG. Image processing and bit manipulation alone were not enough to generate a TRNG. To aid in randomizing the data the Secure Hashing Algorithm 2 (SHA-2) was used to generate the final seed that is fed into the OpenSSL PRNG.

Figure 5. Block diagram outlining the main steps in the algorithm. Where *r* are the three *n x m* matrices of the RGB color channels (a single *n x m* for grayscale). The output of the image processing block *b* is a *n*m* one dimensional array of intensity values. The output of the bit manipulation block is one bit string of length *p,* where *p* is the desired input length for the hashing block.

A. Image Processing

As the purpose of this project was to introduce image processing techniques to existing cryptographic algorithms the most important consideration was how to take an input image and ensure it was a truly random source of data. The goal was also to add enough complexity that even if a malicious party could guess the content of the input image they would not be able to reproduce the seed. Great care was taken to ensure that the processing techniques did not end up weakening the randomness of the data or prove to be computationally expensive and unnecessary.

A.1. Bad image detection

A.1.1. Algorithm

The algorithm for detecting a potentially bad image and modifying the brightness and contrast is shown in Figure 6. Each image undergoes adjustment to ensure the brightness and contrast is not too high or too low.

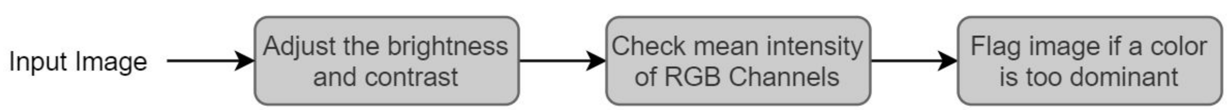

Figure 6. Overview of Bad Image Detection and Correction Algorithm

As shown in Figure 7 below, once the alpha and beta constants are calculated for an image, each pixel of the RGB image is modified by Equation (1) below [10]. The value for the output pixel is restricted to [0, 255]. The results of the image adjustment are shown below in Figures 8, 9, 10 & 11.

$$outputPixel(row, col, channel) = alpha * inputPixel(row, col, channel) + beta \tag{1}$$

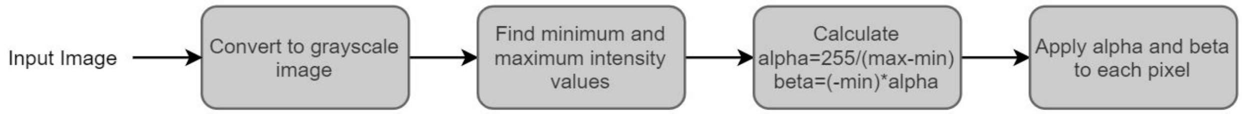

Figure 7. Overview of Image Adjustment

To determine if a single color is too dominant in the image after the brightness and contrast adjustment, the algorithm checks if the mean value of any of the RGB channels is greater than an upper threshold. If it is, the algorithm then checks if the mean value of the other RGB channels are below a lower threshold. If there are large differences between mean RGB channel values, it may indicate that there is too much of a single color in the image. The image flags the image as a bad image and alerts the user to use a new image.

A.1.2. Image Adjustment Results

Figure 8. Example of modifying an image. The left is the original, the middle is after salt and pepper filtering, and the right is after auto adjustment. Alpha = 1.51 and Beta = -55.83

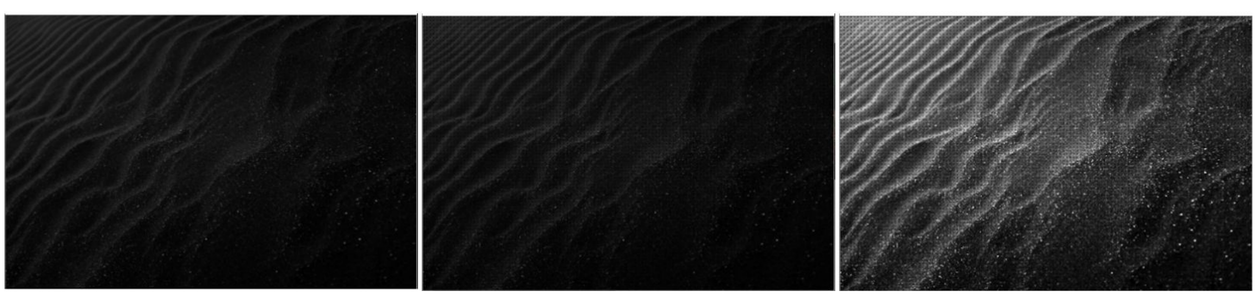

Figure 9. Example of modifying a dark image. The left is the original, the middle is after salt and pepper filtering, and the right is after auto adjustment. Alpha = 3.86 and Beta = -3.86

Figure 10. Example of modifying a low quality dark image. The left is the original, the middle is after salt and pepper filtering, and the right is after auto adjustment. The salt and pepper filter can be seen best in this example. Alpha = 5.43 and Beta = 0.00

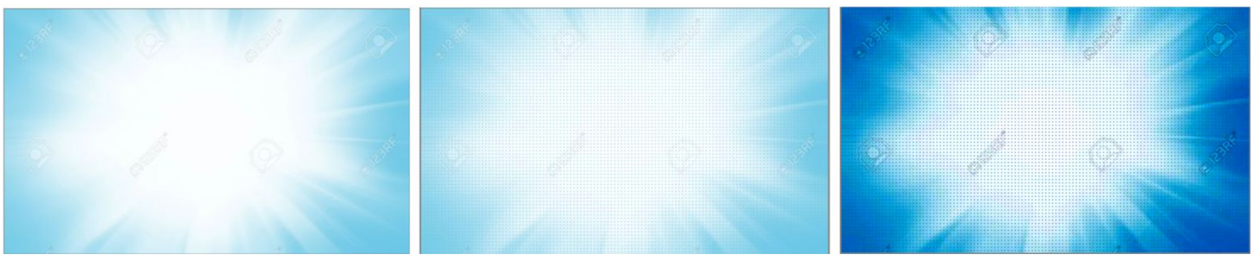

Figure 11. Example of modifying a light image. The left is the original, the middle is after salt and pepper filtering, and the right is after auto adjustment. Alpha = 3.27 and Beta = -572.16

A.2. Filtering

A.2.1. Testing

Multiple filters from the 408G Lab assignments were tested to add random data to the images. We tested a filter filled with all random values and multiple templates of known filters filled with random values. The goal of filtering was to add random data without changing the image too much and removing the need for the user to take an image.

-1	-1	-1	-1	-1
-1	A	B	C	-1
-1	D	E	F	-1
-1	G	H	I	-1
-1	-1	-1	-1	-1

Figure 12. Template for Random 5x5 Lithographic Filter

```
0 -1 -2 -3 -4
0 -1  A  B  1
0 -1  C  D  1
0 -1  E  F  1
0 -1 -2 -3 -4
```

Figure 13. Template for Random 5x5 Psychedelic Filter

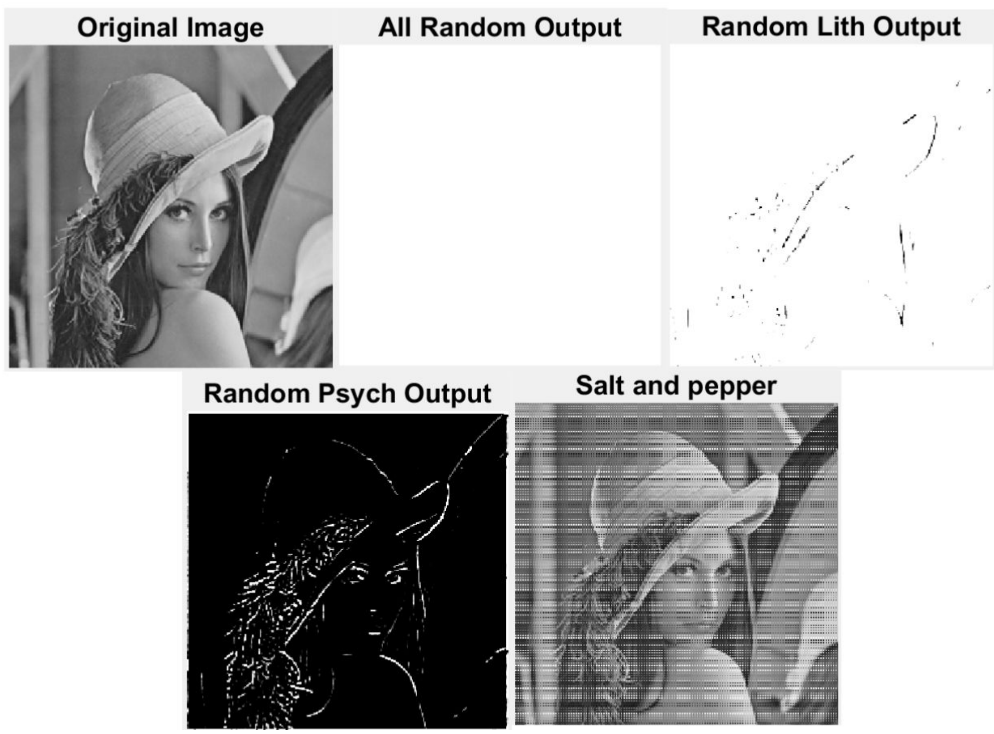

Figure 14. Results of four different filtering techniques. Besides the salt and pepper filter, each filter was 5x5. The random values for the filter or the salt and pepper filter were [1, 10].

As seen in Figure 14 above, the filters used in 408G made significant changes to the image. We chose to use a salt and pepper filter to add random pixels without destroying the image or relying heavily on a random number generation function to produce the filter values.

A.2.2. Algorithm for Filtering

If the image is grayscale a Salt & Pepper filter is applied to the one channel and then the image is passed to the next step. If the image is RGB then the randomly generated Salt & Pepper noise filter was applied to each of the RGB color channels separately to create three sources of data that had now been altered from their original state. The method for random generation of the Salt & Pepper filter was just a call to the built in random functions of C++. Typically this is considered an insecure method of RNG. However, through our testing this was not found to degrade the classification of the system as a TRNG.

Figure 15. RGB input image

Figure 16. Image split into its red (left), green (middle), and blue (right) color channels

Figure 17. The three color channels with salt and pepper noise added to them

A.3. Matrix divisions

After each of the RGB color channels has had a noise filter applied to a series of matrix manipulations are applied to further alter and randomize the input image.

A.3.1. Channel Blending

First, the three separate channels are XOR'd at each pixel so that a highly colorful image will produce a more random collection of pixels.

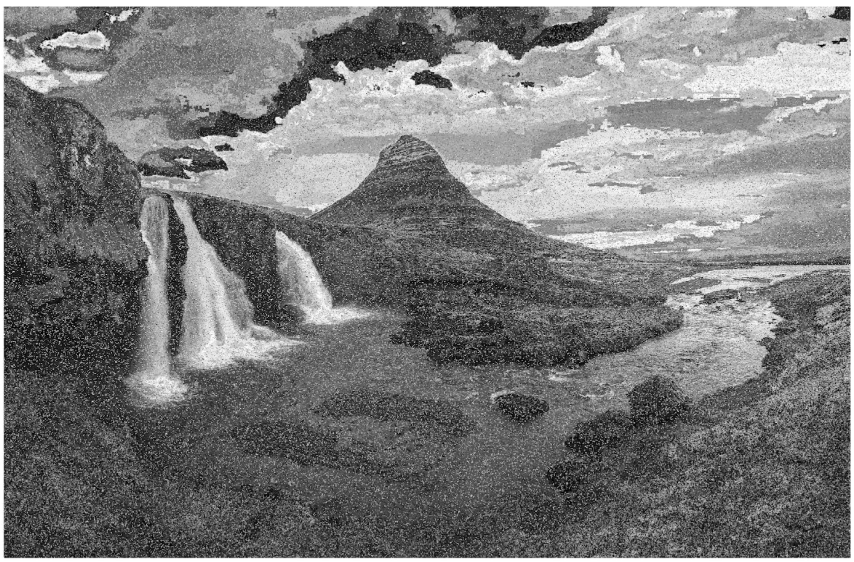

Figure 18. The three channels recombined by XORing each pixel

A.3.2. Converting the Matrix to a Square

With the three two-dimensional matrices now reduced to one two-dimensional matrix, the matrix is reshaped into a square matrix. This serves two purposes. For one, it is significantly easier to do the following matrix manipulations with a square matrix. Secondly, this again serves to alter the input image.

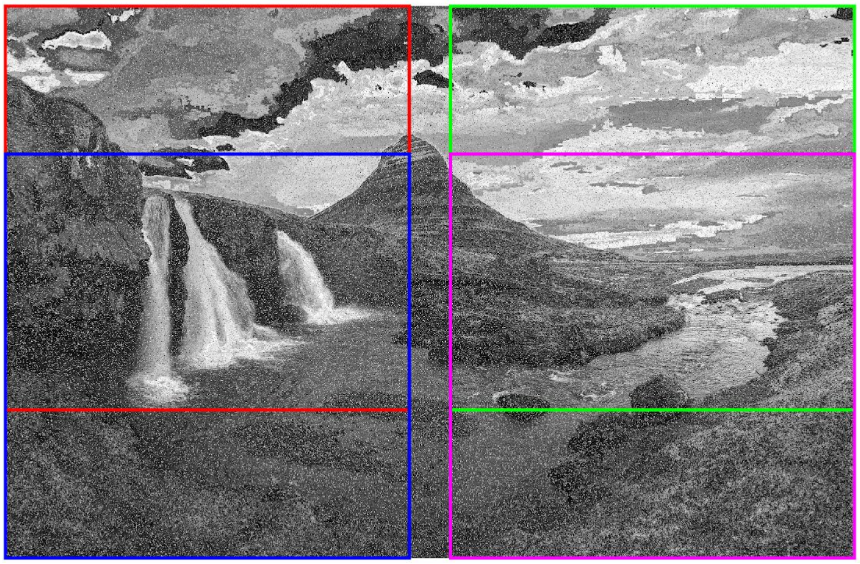

Figure 19. The four squares the algorithm could randomly chose to crop to

Figure 20. Image reformatted into a square after picking the red square in Figure 19

Initially testing was done to reshape the image into a square. However, the noise introduced in this reshaping proved to degrade the quality of the RNG instead of improve it. With this being the case it was much easier to simply crop the image into a square. While this does throw out a good section of the image this limitation was designed around in later steps of the algorithm. To reintroduce some randomness back into this stage one of four squares is randomly chosen to crop to. As seen in Figure 19. The process for randomly choosing an index is described below in Figure 23.

Landscape:

$reshape\,factor \;=\; floor(\;(rows \;-\; columns)\,/\,2)$ (2)

Portrait:

$reshape\,factor \;=\; floor(\;(columns \;-\; rows)\,/\,2)$ (3)

The squared image will be (*rows + reshape factor) x* (*rows + reshape factor)* for a landscape image and (*columns + reshape factor) x* (*columns + reshape factor)* for a portrait image.

A.3.3. Converting the Square Matrix into a One-Dimensional Array

Similar to above this served the purpose of further altering the image and making logic easier. The goal of this section was to find the best way to divide up the image so that an even spread of intensity values was found in each part of the one-dimensional array. Numerous methods were tested and the one that ultimately worked the best was what we referred to as "jumping" through the matrix. First the pixel in the top left corner was chosen. Then the pixel in the bottom right corner was. Then the algorithm jumped back up to the top and repeated this process of alternating between choosing a pixel from the top and bottom of the matrix until it had worked its way to the center of the image.

For a NxN matrix A:

$Order\ is\ A[0,0],\ A[N,N],\ A[1,0],\ A[N-1,N],\ A[0,1],\ A[N,N-1],\ A[0,2],\ A[N,N-2],\ A[1,1],\ ...$

B. Bit Manipulation

B.1. Generating Bit Strings

With the image now sufficiently scrambled and reordered into a one-dimensional array the intensity values of the pixels can now be converted into binary for bit manipulation. Through testing it was determined that the best way to generate strings of bits that were equally 0 and 1 was to take the least significant bit (LSB) of each pixel. These bits were then appended into a collection of 256 long bit strings that will be further altered in the next couple of steps. The size of the array of bit strings is as follows.

For a NxN matrix A:

$Length\ of\ 1D-Array \;=\; N^2\ x\ 1$ (4)

$Number\ of\ usable\ bit\ strings \;=\; N^2\,/\,256$ (5)

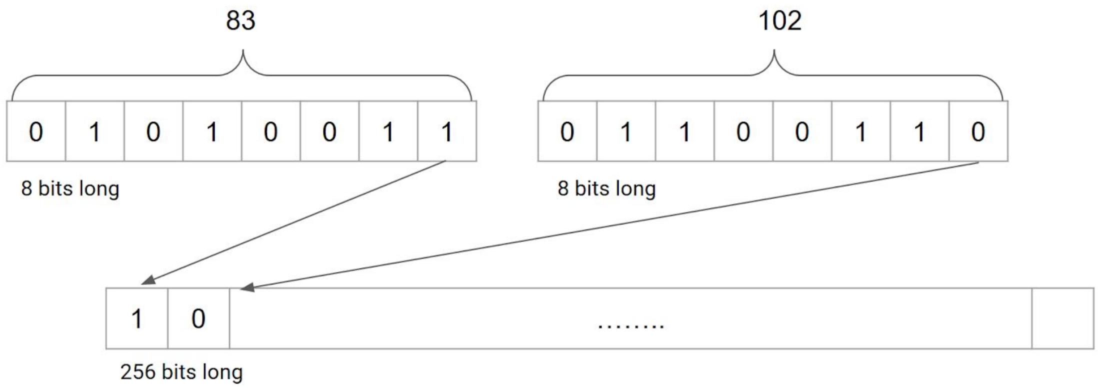

Figure 21. Illustration of how one of the N^2 / 256 bit strings was created by taking the LSB of 256 intensity values

B.2. Extractors

When dealing with a weakly random source such as images saved in the JPEG format it is necessary to introduce an extractor to help with equal distribution of 1s & 0s in the bit string. The most common one is the Von Neumann extractor which takes two successive bits and takes the first bit if and only if the two bits are different [11]. We originally used this and it did help significantly with equalizing the distribution of 1s & 0s. However, when combining this with the LSB implementation of generating bit strings—which was also working to normalize the distribution—the runs of 1s was squashed too much. We instead chose to design an XOR extractor which works similar in concept to the Von Neumann extractor except it just XOR's two successive bits and takes the output. This proved to work better at normalizing the distribution without squashing the runs of 1s.

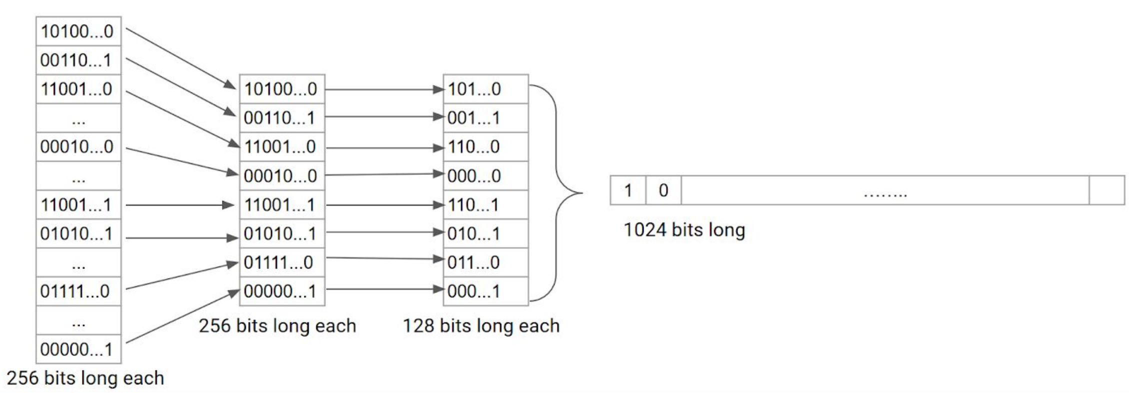

Figure 22. Illustration of how eight of the N^2 / 256 bit strings were randomly chosen and fed through the XOR extractor to generate a 1024 long bit string

For the purpose of feeding into the below hashing function the extractor functions were designed to run until they had generated a 1024 long bit string. This means that 8 of the $N^2/256$ bit strings are needed for the XOR extractor to work. Instead of taking the first 8 bit strings of the array of bit strings each time a new bit string was needed a random one was chosen. The algorithm for selecting a bit string is shown below in Figure 20. The algorithm relies on the execution time of a function call to select the bit string that will be used. Through testing, enough variation was found in the least significant bits of the elapsed time to produce variation in the bit string selected.

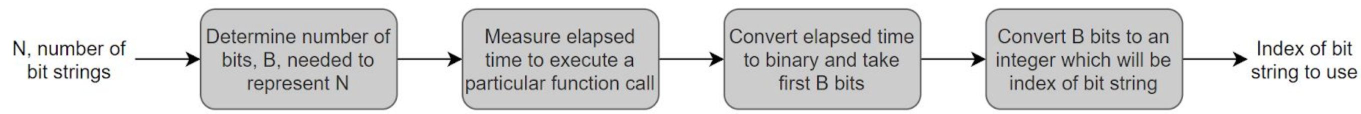

Figure 23. Algorithm for selecting a bit string to be passed into hash function

C. SHA-2

C.1. Introduction

Although the techniques implemented above had a great deal of success in approaching a TRNG they were ultimately not enough to pass the testing suite outlined in a section below. To remedy this a hashing algorithm was used. Hashing algorithms have a property called the avalanche effect. This property means that if a single bit is changed in the input to the hashing algorithm all of the bits in the output hash have a 50% chance of changing [12]. This adds a great deal of entropy to our algorithm and was the final push to pass the statistical tests outlined below.

SHA-2 was chosen as the hashing algorithm as it is one of if not the most common hashing algorithm and has been rigorously tested. Since OpenSSL requires a 256 bit input seed the SHA-256 of the SHA-2 family was used. [13]

C.2. Input

The SHA-2 hashing function can accept an input of any length but the optimal length found through testing is 1024 bits. At this length we have only taken a randomly chosen small part of an already heavily scrambled image.

C.3. Converting its output into a seed

The implementation of the SHA-2 hashing function used in this project outputs a 256 bit hexadecimal string. Since the OpenSSL PRNG uses a binary seed we simply convert this hexadecimal value back into binary before seeding the PRNG.

IV. Windows Form Application

As a proof of concept a simple windows form application was developed to display how the algorithm could be applied in an app based solution. The application has two main functionalities.

- A user can select an input image from their computer and the app will output the seed that was created, a pair of RSA keys, and the AES key.
- The four initial statistical tests—outlined below—can be run to test the generator for a library of about 1000 images. The app will output whether these tests passed for the image set as well as further details on the results of the tests.

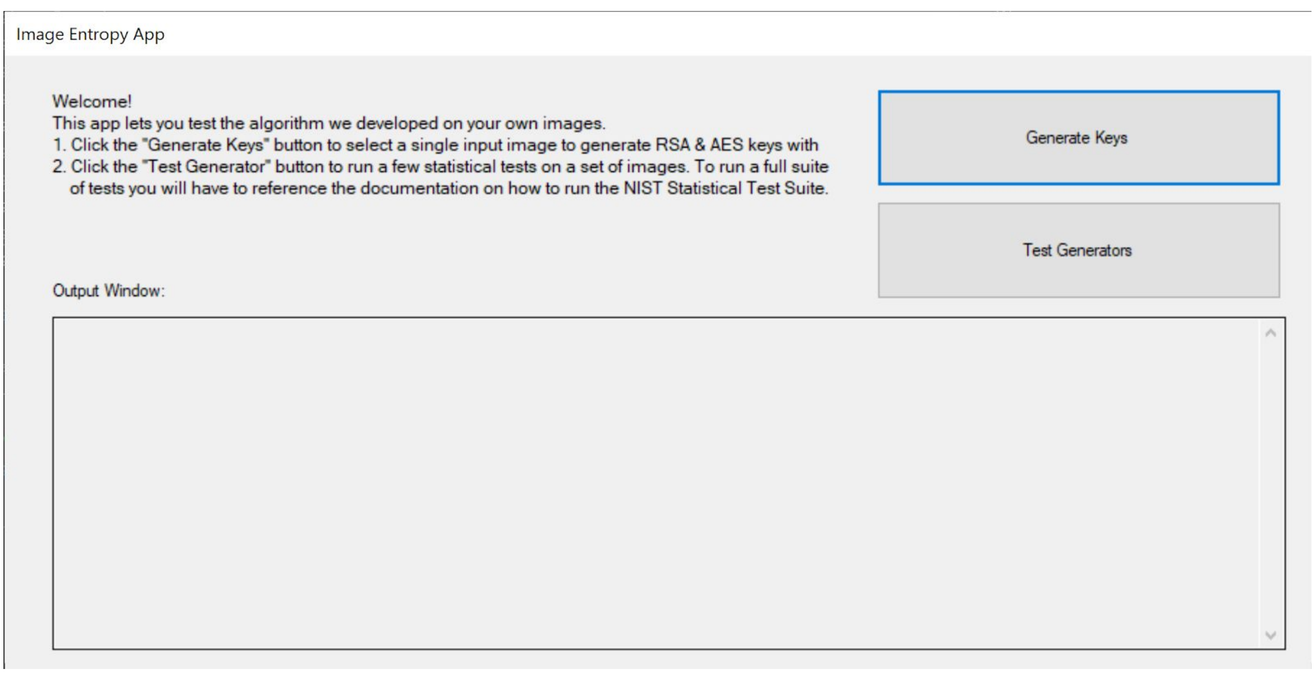

Figure 24. The Windows Form App developed to display the functionality of the algorithm.

In most applications of this algorithm the encryption keys would not need to be visible to the user. This app is simply designed to show that our algorithm could generate a seed and the corresponding encryption keys. To understand how testing the randomness of the generator is done see the section below. This app also takes no consideration of secure data storage as it is simply to show the algorithm works and not meant for any real world applications.

V. Testing

A. Introduction to the National Institute of Standard and Technology Statistical Test Suite (NIST STS)

In 2010 NIST developed a STS that expanded upon and standardized existing statistical tests for random number generators. This test suite is the recommended suite by OpenSSL to test a RNG that is being used to seed their PRNG [1].

B. Summary of the Tests Used

The entire suite consists of fifteen tests. For a detailed description of each of these tests reference the NIST paper that outlines their suite [14]. OpenSSL recommends the first tests of this suite that should be run are the frequency test, the runs test, and the longest runs of ones in a block test [1]. When testing the algorithm these three tests—along with the block frequency test—were the main tests we were trying to pass.

C. Results

Periodically while testing the initial four tests we ran the algorithm against the full suite. Of the fifteen tests we tested our algorithm against twelve. Due to insufficient input length the Maurer's "Universal Statistical" test, random excursions test, and random excursions variant test were excluded.

Approximately 1,000 images or 256,000 bits were tested against the suite. Prior to implementing the SHA-2 hashing algorithm four of the twelve tests were passed. The non-overlapping test was also mostly passed. To pass the remaining seven the hashing algorithm was needed.

Summary		
	No Hashing	Hashing
Approximate Entropy	FAILURE	SUCCESS
Block Frequency	FAILURE	SUCCESS
Cumulative Sums	SUCCESS	SUCCESS
FFT	FAILURE	SUCCESS
Frequency	SUCCESS	SUCCESS

Linear Complexity	SUCCESS	SUCCESS
Longest Run of Ones	SUCCESS	SUCCESS
Non-Overlapping Template	MOSTLY SUCCESS	SUCCESS
Overlapping Template	FAILURE	SUCCESS
Rank	FAILURE	SUCCESS
Runs	FAILURE	SUCCESS
Serial	FAILURE	SUCCESS

Table 1. Summary of tests passed by algorithm with hashing excluded and included

In Table 1 and all the tables below a red cell or the word FAILURE indicates the algorithm failed that test. A green cell or the word SUCCESS indicates the test passed that test. The value which indicates whether a test passed or failed is referred to as the p value. This value must be greater than or equal to 0.01. Which indicates a 99% confidence level or 1 in 100 bits sequences would be expected to fail [14].

C.1. Approximate Entropy

	No Hashing	Hashing
m (block length)	10	10
n (sequence length)	260864	260864
Chi^2	1644.374903	1060.361043
Phi(m)	-6.927875	-6.929324
Phi(m+1)	-7.617871	-7.620439
ApEn	0.689995	0.691115
Log(2)	0.693147	0.693147
p value	0.000000	0.209302

Table 2. Results of Approximate Entropy Test

C.2. Block Frequency

	No Hashing	Hashing
Chi^2	199.412577	107.283742
Number of Substrings	100	100
Block Length	2608	2608
Bits Discarded	64	64
p value	0.000000	0.291186

Table 3. Results of Block Frequency Test

C.3. Cumulative Sums

Forward Test		
	No Hashing	Hashing
Max Partial Sum	611	838
p value	0.462508	0.201706
Reverse Test		
	No Hashing	Hashing
Max Partial Sum	605	910
p value	0.471642	0.149597

Table 4. Results of Cumulative Sums Test

C.4. FFT Test

	No Hashing	Hashing
Percentile	93.934004	94.955992
N_l	122520.000000	123853.000000
N_o	123910.400000	123910.400000
d	-24.981348	-1.031307
p value	0.000000	0.302397

Table 5. Results of FFT Test

C.5. Frequency Test

	No Hashing	Hashing
Nth Partial Sum	-6	-702
S_n/n	-0.000023	-0.002691
p value	0.990627	0.169301

Table 6. Results of Frequency Test

C.6. Linear Complexity

No Hashing								
M (substring length) = 500, N (number of substrings) = 521, Bits Discarded = 364								
C0	C1	C2	C3	C4	C5	C6	Chi^2	p value
6	13	53	267	141	26	15	6.928896	0.327473

Table 7. Results of Linear Complexity Test with no hashing algorithm included

Hashing								
M (substring length) = 500, N (number of substrings) = 521, Bits Discarded = 364								
C0	C1	C2	C3	C4	C5	C6	Chi^2	p value
5	7	72	253	129	36	19	12.758999	0.047027

Table 8. Results of Linear Complexity Test with hashing algorithm included

C.7. Longest Runs of Ones

No Hashing						
M (substring length) = 128, N (number of substrings) = 2038, Chi^2 = 9.511930						
<= 4	5	6	7	8	>= 9	p value
263	475	490	331	227	252	0.090306

Table 9. Results of Longest Runs of Ones Test with no hashing algorithm included

Hashing						
M (substring length) = 128, N (number of substrings) = 2038, Chi^2 = 5.582931						
<= 4	5	6	7	8	>= 9	p value
266	463	512	366	202	229	0.348938

Table 10. Results of Longest Runs of Ones Test with hashing algorithm included

C.8. Non-Overlapping Template

LAMBDA = 63.671875 M = 32608 N = 8 m = 9 n = 260864

Template	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	Chi^2	P_value	Assignment	Index
	F R E Q U E N C Y											
000000001	84	61	82	56	89	69	69	72	25.706105	0.001179	FAILURE	0
000000011	76	68	65	53	78	63	66	61	8.196366	0.414528	SUCCESS	1
000000101	74	52	66	63	73	82	75	48	16.982195	0.030295	SUCCESS	2
000000111	70	57	69	60	74	72	67	67	5.271986	0.728150	SUCCESS	3
000001001	59	58	64	72	58	55	76	77	9.102305	0.333739	SUCCESS	4
000001011	100	56	68	67	68	59	66	58	24.142917	0.002169	FAILURE	5
000001101	74	72	51	74	75	67	46	49	18.031257	0.020993	SUCCESS	6
000001111	70	55	75	65	71	67	71	66	5.996737	0.647597	SUCCESS	7
000010001	72	59	72	64	61	57	69	63	3.916377	0.864588	SUCCESS	8
000010011	69	66	69	92	65	61	75	81	21.148498	0.006763	FAILURE	9
000010101	54	62	64	58	68	62	70	68	3.393115	0.907325	SUCCESS	10
000010111	87	58	72	72	62	59	65	58	12.564665	0.127732	SUCCESS	11
000011001	66	79	75	67	68	72	71	84	15.181991	0.055702	SUCCESS	12
000011011	71	68	58	85	78	59	53	73	16.038192	0.041837	SUCCESS	13
000011101	57	52	59	66	75	47	57	69	11.160333	0.192779	SUCCESS	14
000011111	58	59	82	55	59	59	62	59	8.662785	0.371536	SUCCESS	15
000100011	74	52	70	78	54	56	66	60	10.711678	0.218577	SUCCESS	16
000100101	47	61	78	53	66	52	67	56	13.250337	0.103522	SUCCESS	17
000100111	59	64	62	55	77	74	70	78	10.224958	0.249591	SUCCESS	18
000101001	57	48	57	56	80	80	49	65	18.575327	0.017303	SUCCESS	19
000101011	57	65	62	56	74	67	78	63	7.006719	0.535908	SUCCESS	20
000101101	72	56	74	59	73	69	51	62	8.696789	0.368516	SUCCESS	21
000101111	63	66	58	62	68	63	67	67	1.334578	0.995126	SUCCESS	22
000110011	57	95	98	73	79	68	74	70	43.717283	0.000001	FAILURE	23
000110101	69	81	58	57	62	67	58	48	11.319697	0.184234	SUCCESS	24
000110111	77	66	54	76	63	81	50	79	18.696119	0.016572	SUCCESS	25
000111001	76	54	60	75	45	61	60	62	12.333232	0.136944	SUCCESS	26
000111011	59	55	55	58	69	63	57	71	5.383135	0.715950	SUCCESS	27
000111101	60	63	68	66	57	67	70	55	3.393115	0.907325	SUCCESS	28
000111111	50	56	71	60	67	69	65	67	5.932281	0.654817	SUCCESS	29
001000011	76	53	60	65	50	51	73	80	15.952419	0.043066	SUCCESS	30
001000101	50	53	63	54	79	72	81	56	17.186729	0.028222	SUCCESS	31
001000111	64	46	59	69	56	73	59	52	10.825365	0.211792	SUCCESS	32
001001011	56	68	69	53	53	51	64	52	10.242722	0.248402	SUCCESS	33
001001101	62	77	56	80	58	61	54	76	12.842283	0.117393	SUCCESS	34
001001111	56	61	59	52	54	73	67	72	7.877638	0.445513	SUCCESS	35
001010011	62	55	57	65	71	69	52	66	5.652125	0.686134	SUCCESS	36
001010101	56	60	78	52	67	69	75	56	10.402593	0.237898	SUCCESS	37
001010111	62	75	48	54	72	61	77	59	12.119563	0.145948	SUCCESS	38
001011011	59	63	57	70	72	74	58	63	5.123787	0.744267	SUCCESS	39
001011101	74	46	57	56	66	67	52	55	12.185034	0.143137	SUCCESS	40
001011111	73	71	61	70	65	66	55	59	4.744156	0.784538	SUCCESS	41
001100101	81	45	61	65	41	64	57	51	22.364029	0.004284	FAILURE	42
001100111	70	98	78	73	74	59	75	75	30.791543	0.000153	FAILURE	43
001101011	68	62	62	55	58	69	52	63	4.819778	0.776653	SUCCESS	44
001101101	62	66	61	77	62	54	62	70	5.394808	0.714664	SUCCESS	45
001101111	67	72	57	54	59	75	58	75	8.593761	0.377715	SUCCESS	46
001110101	46	59	60	53	67	61	51	70	11.049185	0.198928	SUCCESS	47
001110111	53	72	69	85	51	52	63	66	15.740780	0.046244	SUCCESS	48
001111011	51	73	75	67	58	82	76	66	14.819615	0.062749	SUCCESS	49
001111101	58	58	52	59	63	52	64	58	6.356067	0.607413	SUCCESS	50
001111111	53	60	77	73	60	69	68	78	10.685287	0.220177	SUCCESS	51
010000011	58	62	53	71	71	57	57	57	6.330691	0.610242	SUCCESS	52
010000111	65	56	61	58	42	59	68	64	9.911305	0.271306	SUCCESS	53
010001011	49	59	71	61	80	55	63	56	11.353194	0.182478	SUCCESS	54
010001111	69	52	55	76	62	70	53	55	10.130050	0.256017	SUCCESS	55
010010011	62	73	51	75	58	47	59	69	12.002831	0.151078	SUCCESS	56
010010111	66	66	66	64	70	58	47	41	14.300921	0.074251	SUCCESS	57
010011011	60	64	60	69	53	69	70	63	3.869177	0.868730	SUCCESS	58
010011111	54	61	63	63	63	75	55	83	11.029899	0.200011	SUCCESS	59
010100011	60	68	66	69	67	78	49	73	9.495639	0.302223	SUCCESS	60
010100111	50	52	58	61	70	84	54	68	15.071857	0.057763	SUCCESS	61
010101011	52	59	72	50	64	64	64	50	9.770212	0.281526	SUCCESS	62
010101111	61	72	48	47	66	55	70	59	12.059674	0.148561	SUCCESS	63
010110011	56	73	58	68	73	61	57	59	5.802353	0.669360	SUCCESS	64
010110111	51	63	63	56	71	69	69	57	6.095705	0.636512	SUCCESS	65
010111011	58	58	69	53	69	71	48	63	8.685116	0.369551	SUCCESS	66
010111111	76	68	74	68	58	47	73	66	11.347104	0.182797	SUCCESS	67
011000111	63	46	54	59	56	60	60	70	8.997246	0.342528	SUCCESS	68
011001111	61	85	67	50	70	63	67	63	11.564326	0.171727	SUCCESS	69
011010111	72	58	62	54	64	59	70	76	6.688498	0.570584	SUCCESS	70
011011111	52	76	48	67	62	80	65	63	13.260995	0.103180	SUCCESS	71
011101111	63	55	79	67	54	58	72	82	13.848205	0.085809	SUCCESS	72
011111111	66	64	94	75	72	68	66	91	30.760076	0.000155	FAILURE	73
100000000	84	61	82	56	89	69	69	72	25.706105	0.001179	FAILURE	74
100010000	61	63	77	67	57	63	65	65	3.975758	0.859304	SUCCESS	75

100100000	83	64	66	70	66	63	54	56	9.377893	0.311427	SUCCESS	76
100101000	52	64	62	61	94	68	72	62	18.790012	0.016024	SUCCESS	77
100110000	59	62	67	71	65	60	72	75	4.910118	0.767138	SUCCESS	78
100111000	65	60	72	58	79	82	73	77	15.466207	0.050690	SUCCESS	79
101000000	69	43	58	65	65	62	60	54	9.764630	0.281936	SUCCESS	80
101000100	51	65	65	40	61	57	61	63	12.728090	0.121554	SUCCESS	81
101001000	62	61	39	61	76	79	65	65	16.504610	0.035701	SUCCESS	82
101001100	69	61	57	64	60	48	52	56	8.678011	0.370182	SUCCESS	83
101010000	72	65	50	57	64	78	65	64	8.280108	0.406598	SUCCESS	84
101010100	60	70	73	55	59	70	74	60	6.460110	0.595836	SUCCESS	85
101011000	67	59	59	55	60	78	70	57	7.036663	0.532682	SUCCESS	86
101011100	63	62	62	60	65	59	84	63	7.418832	0.492190	SUCCESS	87
101100000	75	66	72	64	60	66	69	56	5.024312	0.754975	SUCCESS	88
101100100	61	62	55	57	59	71	82	51	11.395827	0.180264	SUCCESS	89
101101000	58	76	65	55	69	55	48	51	12.520003	0.129467	SUCCESS	90
101101100	55	55	72	59	61	56	57	60	5.937356	0.654249	SUCCESS	91
101110000	73	40	62	57	69	55	67	50	16.180300	0.039871	SUCCESS	92
101110100	72	59	63	72	69	69	57	64	4.261496	0.832793	SUCCESS	93
101111000	55	60	48	70	70	51	76	71	12.678352	0.123406	SUCCESS	94
101111100	71	68	54	58	69	85	62	72	12.238832	0.140862	SUCCESS	95
110000000	80	62	65	47	67	71	79	82	19.241713	0.013619	SUCCESS	96
110000010	70	60	59	63	59	63	77	58	5.000458	0.757527	SUCCESS	97
110000100	75	63	83	86	77	71	67	56	21.148498	0.006763	FAILURE	98
110001000	66	44	91	74	50	78	77	68	29.793741	0.000230	FAILURE	99
110001010	50	69	53	74	65	66	73	68	8.912997	0.349689	SUCCESS	100
110010000	91	53	64	83	67	48	55	45	31.100120	0.000135	FAILURE	101
110010010	53	69	63	55	73	54	71	54	8.863259	0.353962	SUCCESS	102
110010100	67	54	54	65	62	64	68	69	4.059500	0.851716	SUCCESS	103
110011000	57	72	70	58	70	57	68	71	5.571935	0.695058	SUCCESS	104
110011010	53	58	86	62	68	63	48	65	14.843469	0.062262	SUCCESS	105
110100000	65	55	57	60	56	38	48	61	17.956142	0.021558	SUCCESS	106
110100010	47	69	63	51	72	67	50	62	11.977962	0.152190	SUCCESS	107
110100100	53	77	57	73	72	69	53	61	10.423909	0.236524	SUCCESS	108
110101000	73	69	49	45	57	60	53	72	14.950558	0.060114	SUCCESS	109
110101010	56	69	67	53	56	65	63	54	5.957657	0.651975	SUCCESS	110
110101100	63	66	60	61	66	68	64	70	1.474656	0.993121	SUCCESS	111
110110000	56	58	77	73	75	62	72	80	13.362500	0.099971	SUCCESS	112
110110010	58	57	53	64	60	67	68	62	3.845323	0.870802	SUCCESS	113
110110100	54	61	50	63	69	45	55	44	18.307860	0.019033	SUCCESS	114
110111000	67	73	62	61	64	66	63	44	8.136478	0.420253	SUCCESS	115
110111010	53	73	60	77	60	61	60	68	7.224956	0.512568	SUCCESS	116
110111100	65	76	59	65	52	58	69	70	6.726563	0.566401	SUCCESS	117
111000000	84	44	75	61	80	75	75	68	23.998779	0.002293	FAILURE	118
111000010	74	61	77	67	67	64	52	62	7.352853	0.499084	SUCCESS	119
111000100	69	74	65	59	56	61	70	76	6.767165	0.561950	SUCCESS	120
111000110	76	59	66	75	50	75	70	59	11.119731	0.195007	SUCCESS	121
111001000	72	46	57	70	59	52	47	52	16.865463	0.031541	SUCCESS	122
111001010	79	56	74	67	71	61	75	59	10.110764	0.257338	SUCCESS	123
111001100	77	65	79	55	57	68	63	64	8.987096	0.343386	SUCCESS	124
111010000	66	64	56	59	64	53	68	62	3.601201	0.891195	SUCCESS	125
111010010	54	73	54	70	76	55	55	60	10.232064	0.249114	SUCCESS	126
111010100	76	52	36	59	66	48	46	78	29.954628	0.000215	FAILURE	127
111010110	55	74	58	56	63	64	62	60	4.705584	0.788530	SUCCESS	128
111011000	56	54	69	63	78	64	55	85	14.888639	0.061347	SUCCESS	129
111011010	56	63	43	57	84	79	69	57	20.337467	0.009132	FAILURE	130
111011100	55	60	50	63	56	51	72	50	12.209395	0.142103	SUCCESS	131
111100000	72	66	67	67	68	69	86	73	11.849557	0.158044	SUCCESS	132
111100010	60	63	64	71	59	80	73	75	9.281970	0.319071	SUCCESS	133
111100100	54	50	52	57	74	71	58	59	10.972040	0.203288	SUCCESS	134
111100110	65	62	81	50	59	50	66	53	13.314285	0.101484	SUCCESS	135
111101000	57	60	50	58	64	65	65	48	8.548083	0.381839	SUCCESS	136
111101010	89	59	45	54	69	61	54	68	20.355230	0.009072	FAILURE	137
111101100	58	53	68	50	75	74	68	78	13.167102	0.106228	SUCCESS	138
111101110	59	62	53	50	52	52	88	64	19.324440	0.013218	SUCCESS	139
111110000	74	60	87	72	61	74	77	70	17.299908	0.027133	SUCCESS	140
111110010	64	50	58	65	74	66	68	61	5.829252	0.666351	SUCCESS	141
111110100	48	77	54	66	62	63	67	63	8.721151	0.366363	SUCCESS	142
111110110	52	67	65	60	72	82	72	68	10.652805	0.222158	SUCCESS	143
111111000	61	63	93	73	45	67	66	64	21.437789	0.006071	FAILURE	144
111111010	71	82	46	69	60	70	65	78	16.093005	0.041068	SUCCESS	145
111111100	54	53	97	68	63	62	69	66	22.314799	0.004365	FAILURE	146
111111110	66	64	94	75	72	68	66	91	30.760076	0.000155	FAILURE	147

Figure 25. Results of Non-Overlapping Template test with no hashing algorithm included

LAMBDA = 63.671875 M = 32608 N = 8 m = 9 n = 260864

Template	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	Chi^2	P_value	Assignment	Index
		F R E Q U E N C Y										
000000001	60	68	67	63	63	55	70	61	2.705413	0.951464	SUCCESS	0
000000011	65	60	74	63	60	56	74	72	6.021098	0.644868	SUCCESS	1
000000101	58	60	70	61	50	57	58	69	6.249994	0.619251	SUCCESS	2
000000111	66	58	70	56	48	63	79	81	14.905388	0.061011	SUCCESS	3
000001001	63	60	69	64	66	63	62	67	1.009760	0.998186	SUCCESS	4
000001011	53	69	64	68	61	60	59	62	3.351497	0.910404	SUCCESS	5
000001101	76	64	61	74	69	71	52	53	9.713877	0.285685	SUCCESS	6
000001111	73	64	62	70	63	72	79	72	8.186723	0.415446	SUCCESS	7
000010001	63	68	80	70	65	74	52	65	9.294151	0.318093	SUCCESS	8
000010011	57	67	67	68	79	57	71	72	7.924331	0.440896	SUCCESS	9
000010101	64	61	68	72	73	65	60	58	3.731636	0.880488	SUCCESS	10
000010111	67	76	58	66	62	58	69	63	4.294993	0.829576	SUCCESS	11
000011001	52	72	67	67	51	59	59	53	8.865289	0.353787	SUCCESS	12
000011011	51	70	78	89	83	81	56	58	29.433396	0.000266	FAILURE	13
000011101	58	78	65	67	45	60	61	73	11.475509	0.176184	SUCCESS	14
000011111	58	59	61	70	65	73	70	64	3.737219	0.880020	SUCCESS	15
000100011	67	54	82	65	73	60	52	72	12.154582	0.144439	SUCCESS	16
000100101	56	77	65	59	75	77	60	64	9.413927	0.308589	SUCCESS	17
000100111	60	69	56	64	76	63	71	77	7.870533	0.446218	SUCCESS	18
000101001	59	59	62	56	61	62	74	73	5.017206	0.755736	SUCCESS	19
000101011	66	43	65	68	71	64	69	51	11.303964	0.185064	SUCCESS	20
000101101	65	70	66	58	51	61	67	70	4.843632	0.774151	SUCCESS	21
000101111	52	69	58	59	59	68	63	53	6.066268	0.639809	SUCCESS	22
000110011	67	61	54	66	55	68	53	60	5.497329	0.703336	SUCCESS	23
000110101	68	73	64	53	69	67	79	66	8.113639	0.422449	SUCCESS	24
000110111	61	59	64	86	69	66	56	72	11.200428	0.190599	SUCCESS	25
000111001	54	57	74	49	52	54	65	52	13.443705	0.097468	SUCCESS	26
000111011	81	76	67	63	51	67	61	56	11.391767	0.180474	SUCCESS	27
000111101	67	68	68	68	66	67	69	75	3.905719	0.865528	SUCCESS	28
000111111	65	63	75	77	62	57	65	59	6.156608	0.629694	SUCCESS	29
001000011	63	66	65	68	62	72	65	62	1.674115	0.989426	SUCCESS	30
001000101	56	60	60	60	61	72	63	55	4.083861	0.849479	SUCCESS	31
001000111	76	49	71	55	62	65	67	58	8.834330	0.356464	SUCCESS	32
001001011	56	82	46	56	64	54	62	56	14.961723	0.059894	SUCCESS	33
001001101	61	55	72	56	63	58	67	69	4.590375	0.800325	SUCCESS	34
001001111	79	72	55	58	78	59	63	73	11.795252	0.160576	SUCCESS	35
001010011	57	66	59	59	63	65	78	69	5.351160	0.719468	SUCCESS	36
001010101	61	75	63	62	57	65	61	65	3.148993	0.924662	SUCCESS	37
001010111	78	54	67	52	64	62	70	52	10.155934	0.254252	SUCCESS	38
001011011	69	54	55	59	50	58	58	76	10.105181	0.257721	SUCCESS	39
001011101	59	70	65	64	60	65	74	59	3.369768	0.909059	SUCCESS	40
001011111	64	59	58	61	56	47	67	54	8.163884	0.417627	SUCCESS	41
001100101	62	61	60	49	62	59	62	55	5.543006	0.698271	SUCCESS	42
001100111	66	68	53	69	65	74	49	67	8.140030	0.419913	SUCCESS	43
001101011	66	75	76	58	57	67	71	68	7.242212	0.510740	SUCCESS	44
001101101	69	62	64	72	61	61	59	67	2.400896	0.966192	SUCCESS	45
001101111	55	58	75	73	70	79	53	71	12.429155	0.133059	SUCCESS	46
001110101	55	67	74	63	60	59	66	76	6.270802	0.616927	SUCCESS	47
001110111	75	76	62	64	61	66	64	65	4.833989	0.775163	SUCCESS	48
001111011	73	73	58	61	62	58	64	69	4.495467	0.809886	SUCCESS	49
001111101	69	53	59	57	50	64	64	53	8.277063	0.406884	SUCCESS	50
001111111	64	63	62	74	65	67	65	59	2.378565	0.967153	SUCCESS	51
010000011	65	69	69	78	54	65	54	59	7.706601	0.462642	SUCCESS	52
010000111	60	75	63	58	61	65	67	58	3.679868	0.884791	SUCCESS	53
010001011	51	68	67	67	51	74	66	56	8.656187	0.372124	SUCCESS	54
010001111	61	50	64	71	56	55	61	56	7.274694	0.507306	SUCCESS	55
010010011	66	60	41	63	61	55	66	63	10.095031	0.258419	SUCCESS	56
010010111	57	66	52	52	73	42	71	60	15.368254	0.052368	SUCCESS	57
010011011	72	63	78	71	59	70	66	82	11.888637	0.156242	SUCCESS	58
010011111	80	62	51	59	66	62	68	72	8.901831	0.350645	SUCCESS	59
010100011	54	61	62	67	63	62	65	71	2.814024	0.945481	SUCCESS	60
010100111	65	62	57	64	74	62	62	61	2.737895	0.949714	SUCCESS	61
010101011	68	82	65	59	61	50	55	59	10.870535	0.209143	SUCCESS	62
010101111	77	66	76	60	68	63	56	55	8.149166	0.419037	SUCCESS	63
010110011	63	72	66	73	70	74	78	59	8.706432	0.367663	SUCCESS	64
010110111	64	50	55	56	55	56	58	62	7.959858	0.437401	SUCCESS	65
010111011	58	64	64	55	51	59	59	66	5.152209	0.741191	SUCCESS	66
010111111	74	53	65	61	60	50	55	59	8.557219	0.381012	SUCCESS	67
011000111	62	69	74	61	55	66	68	64	3.970175	0.859804	SUCCESS	68
011001111	84	55	49	80	69	66	45	72	23.096393	0.003243	FAILURE	69
011010111	59	62	60	71	60	61	54	62	3.390577	0.907514	SUCCESS	70
011011111	49	49	66	63	62	64	53	62	9.029728	0.339794	SUCCESS	71
011101111	67	63	70	64	62	59	66	64	1.328996	0.995197	SUCCESS	72
011111111	63	48	61	58	62	68	63	72	6.118036	0.634012	SUCCESS	73
100000000	60	68	67	63	63	55	70	61	2.705413	0.951464	SUCCESS	74
100010000	62	65	62	70	63	68	51	59	4.043766	0.853153	SUCCESS	75

100100000	61	79	60	59	64	65	63	77	7.427968	0.491238	SUCCESS	76
100101000	60	66	57	44	72	73	55	68	11.380093	0.181078	SUCCESS	77
100110000	59	71	70	72	68	68	80	64	7.943617	0.438997	SUCCESS	78
100111000	66	60	58	74	51	68	66	61	5.678009	0.683248	SUCCESS	79
101000000	67	55	62	64	60	54	67	73	4.779683	0.780843	SUCCESS	80
101000100	58	59	68	55	72	66	43	72	11.683596	0.165889	SUCCESS	81
101001000	79	60	65	66	66	87	51	70	16.336111	0.037814	SUCCESS	82
101001100	52	84	58	65	49	72	74	65	15.858526	0.044450	SUCCESS	83
101010000	58	66	73	67	57	56	64	67	4.064068	0.851297	SUCCESS	84
101010100	54	67	62	80	66	70	66	65	6.929574	0.544250	SUCCESS	85
101011000	60	72	69	68	65	57	75	53	6.796094	0.558784	SUCCESS	86
101011100	60	53	56	71	68	65	79	52	10.257947	0.247386	SUCCESS	87
101100000	55	60	56	67	62	74	65	63	4.389901	0.820344	SUCCESS	88
101100100	63	67	59	58	73	65	67	69	3.146963	0.924799	SUCCESS	89
101101000	62	55	68	74	65	70	70	65	4.661429	0.793074	SUCCESS	90
101101100	67	68	62	58	58	68	64	74	3.612875	0.890255	SUCCESS	91
101110000	60	61	57	81	77	61	61	72	10.177758	0.252771	SUCCESS	92
101110100	70	78	57	46	71	66	47	63	15.261165	0.054262	SUCCESS	93
101111000	56	65	78	71	68	55	71	51	10.196537	0.251502	SUCCESS	94
101111100	58	54	66	53	60	58	57	54	6.963071	0.540622	SUCCESS	95
110000000	48	62	81	66	67	63	65	64	9.216499	0.324363	SUCCESS	96
110000010	62	65	50	73	58	72	59	65	6.555018	0.585312	SUCCESS	97
110000100	71	78	65	76	70	59	66	62	7.841603	0.449094	SUCCESS	98
110001000	62	59	69	63	74	63	61	47	7.238152	0.511170	SUCCESS	99
110001010	64	60	73	72	60	63	64	59	3.342869	0.911037	SUCCESS	100
110010000	53	75	62	48	63	54	70	68	10.449286	0.234896	SUCCESS	101
110010010	59	51	63	54	64	56	79	60	9.481429	0.303323	SUCCESS	102
110010100	57	63	59	48	80	62	65	58	10.000123	0.265017	SUCCESS	103
110011000	57	60	70	73	57	72	81	58	10.253887	0.247657	SUCCESS	104
110011010	65	61	62	75	65	66	75	74	6.207361	0.624017	SUCCESS	105
110100000	73	53	69	77	52	59	49	61	12.787978	0.119356	SUCCESS	106
110100010	53	64	65	60	64	68	66	56	3.448943	0.903116	SUCCESS	107
110100100	82	69	48	55	61	66	55	70	13.202629	0.105065	SUCCESS	108
110101000	47	63	58	69	62	50	64	79	12.403779	0.134077	SUCCESS	109
110101010	61	74	64	77	70	55	60	59	7.180294	0.517311	SUCCESS	110
110101100	61	85	59	74	56	70	79	61	15.128700	0.056691	SUCCESS	111
110110000	73	73	60	75	68	76	78	64	11.237985	0.188576	SUCCESS	112
110110010	59	70	53	62	75	71	66	70	6.594606	0.580934	SUCCESS	113
110110100	56	72	56	78	67	77	68	70	10.391935	0.238587	SUCCESS	114
110111000	59	61	63	87	65	62	55	67	10.791360	0.213804	SUCCESS	115
110111010	75	80	62	48	59	66	36	64	23.328841	0.002967	FAILURE	116
110111100	65	58	70	64	68	65	80	66	5.954104	0.652373	SUCCESS	117

111000000	59	64	73	60	54	61	61	67	3.919422	0.864319	SUCCESS	118
111000010	59	71	67	75	73	56	75	66	8.031927	0.430358	SUCCESS	119
111000100	62	63	71	66	77	56	66	48	8.930760	0.348171	SUCCESS	120
111000110	72	52	60	64	62	66	68	61	4.113298	0.846759	SUCCESS	121
111001000	52	63	51	61	59	53	69	78	10.943111	0.204943	SUCCESS	122
111001010	69	56	66	56	66	68	71	64	3.727069	0.880871	SUCCESS	123
111001100	69	54	65	53	61	60	73	66	5.694757	0.681380	SUCCESS	124
111010000	79	68	71	67	55	58	57	64	7.640622	0.469338	SUCCESS	125
111010010	76	58	60	57	67	62	55	63	5.386687	0.715559	SUCCESS	126
111010100	54	73	63	60	68	59	59	79	7.987772	0.434666	SUCCESS	127
111010110	56	60	61	73	67	72	70	61	4.776638	0.781161	SUCCESS	128
111011000	64	63	78	61	67	70	70	64	4.941585	0.763801	SUCCESS	129
111011010	63	62	49	74	64	69	64	71	6.617952	0.578356	SUCCESS	130
111011100	63	71	62	69	62	62	64	63	1.485822	0.992941	SUCCESS	131
111100000	71	68	67	49	44	58	56	54	14.134959	0.078315	SUCCESS	132
111100010	51	56	80	59	84	63	59	57	16.044282	0.041751	SUCCESS	133
111100100	70	57	48	74	58	52	71	64	10.703558	0.219068	SUCCESS	134
111100110	70	54	75	64	78	64	64	74	9.325617	0.315576	SUCCESS	135
111101000	67	66	70	60	55	60	73	60	4.209728	0.837723	SUCCESS	136
111101010	64	72	59	70	75	52	55	75	9.735193	0.284106	SUCCESS	137
111101100	69	62	75	64	56	66	59	68	4.294993	0.829576	SUCCESS	138
111101110	72	76	61	66	68	64	54	65	5.652632	0.686077	SUCCESS	139
111110000	68	63	59	51	45	68	46	64	14.314117	0.073937	SUCCESS	140
111110010	77	54	63	70	53	67	68	63	7.403099	0.493830	SUCCESS	141
111110100	69	53	69	56	53	48	60	59	10.139693	0.255358	SUCCESS	142
111110110	52	56	62	75	60	62	61	83	11.745514	0.162925	SUCCESS	143
111111000	74	58	73	54	66	61	51	78	11.333401	0.183514	SUCCESS	144
111111010	58	63	54	66	67	50	65	58	5.903859	0.658000	SUCCESS	145
111111100	67	56	67	62	62	75	54	71	5.882035	0.660444	SUCCESS	146
111111110	63	48	61	58	62	68	63	72	6.118036	0.634012	SUCCESS	147

Figure 26. Results of Non-Overlapping Template test with hashing algorithm included

C.9. Overlapping Template

No Hashing							
n (sequence length) = 260864, m (block length of 1s) = 9, M (length of substring) = 1032 N (number of substrings) = 252, lambda [(M-m+1)/2^{m}] = 2, eta = 1							
0	1	2	3	4	>= 5	Chi^2	p value
110	27	49	15	20	40	17.069506	0.004370

Table 10. Results of Overlapping Template Test with no hashing algorithm included

Hashing							
n (sequence length) = 260864, m (block length of 1s) = 9, M (length of substring) = 1032 N (number of substrings) = 252, lambda [(M-m+1)/2^{m}] = 2, eta = 1							
0	1	2	3	4	>= 5	Chi^2	p value
91	52	29	22	25	33	5.315932	0.378552

Table 11. Results of Overlapping Template Test with hashing algorithm included

C.10. Rank

	No Hashing	Hashing
Probability: P32	0.288788	0.288788
Probability: P31	0.577576	0.577576

Probability: P30	0.133636	0.133636
Frequency: F32	4	75
Frequency: F31	4	153
Frequency: F30	246	26
Number of Matrices	254	254
Chi^2	1529.173706	2.166124
Number of Bits Discarded	768	768
p value	0.000000	0.338557

Table 12. Results of Rank Test

C.11. Runs

	No Hashing	Hashing
Pi	0.499988	0.498654
V_n_obs (Total # of Runs)	129342	130462
(V_n_obs - 2*n*pi*(1-pi)) / (2*sqrt(2*n)*pi*(1-pi))	3.018105	0.085683
p value	0.000020	0.903553

Table 13. Results of Runs Test

C.12. Serial

	No Hashing	Hashing
Block length (m)	16	16
Sequence length (n)	260864	260864
Psi_m	100708.993131	66330.630029
Psi_m-1	50662.374877	33427.721295
Psi_m-2	25627.006869	16803.674190
Del_1	50046.618253	32902.908734
Del_2	25011.250245	16278.861629
p value 1	0.000000	0.298448
p value 2	0.000000	0.718495

Table 14. Results of Serial Test

D. Replication of Testing Results

The approximately1000 images the algorithm was tested against are provided in the deliverables of this project. The first four tests were rewritten in C++ as a separate library to allow for easier understanding of how to pass those tests.

In the GUI made for this project there is a test generator button which will run these four tests on the approximately thousand provided images to show that the generator passes those.The text file of approximately 256,000 bits generated when clicking the test generator button in the GUI can be inputted into the full test suite.

To use the full test suite it must first be downloaded from NIST's website [15]. Because the suite was built in Linux it is easiest to download Cygwin to run the suite in the terminal [16]. With Cywgin installed, reference the NIST STS paper on how to build and run the suite [14].

VI. Applications of This Project

A. File Transfer App

The original end goal of this project was to create a file transfer app that leveraged the TRNG algorithm we designed. This intuitively made a lot of sense in that a user could take a picture with their phone before they wanted to send a file and that image would be used to encrypt the file they wanted to send. This could also be scaled back to only asking the user for a new image after a set time period or number of sends.

Ultimately, due to time and cost constraints this portion of the project was not fully realized. Designing an app capable of transferring files of even a relatively decent size would require hosting the encrypted data on the cloud. The security and cost concerns of this were enough that we chose not to pursue the app.

B. Other Applications

The algorithm is fairly flexible and can be easily tweaked to generate a seed of any length. Because of this any application that could afford to ask users for an input image to use for RNG purposes would be able to leverage this application.

It should be stressed that the more security that is needed the more consideration will have to go into how each image is inputted into the algorithm and what the image actually is. The algorithm has been designed in such a way that the same image inputted twice will not output the same seed. There is also—as discussed above—bad image detection that will ensure the user is not inputting an image that will not generate a strong seed. However, even with these two components in mind the greatest security is for each input image to be sufficiently colorly diverse and different from any of the previous input images by the user.

VII. Expansions of the concept

A. Video

Early on into the project it was decided that taking even a short video as an input instead of a single image would produce more data than may be necessary to create a TRNG. The results of this project show that that was in fact true and the overwhelming amount of data supplied from a video was not required. Because video introduces multiple frames, sound, motion detection, and countless other sources of random data it could in theory produce a more secure TRNG. This would however introduce significantly more computational complexity to the algorithm at potentially only minor increases to the randomness of the TRNG.

B. Accelerometer or Gyroscope

If this algorithm was applied to its original goal of being for mobile devices access to a mobile devices accelerometer or gyroscope would provide yet another source of random data. This would be limited in that this data from these sensors would only be captured while the app is running. Further testing would need to be done on how much data is provided by these sensors and what their data would be best used for.

C. RAW Image data

Due to the photoelectric effect an image in its RAW format would be a significantly more random data source [17]. This would require access to the image when it is taken by then sensor and prior to any compression [18]. Depending on the camera in use this is not always a data source that is readily available. This is also a data source that is significantly larger than the compressed image in the JPEG format. There is not a standard format for storing RAW image data and this would add extensive work to ensure the algorithm could utilize as many of these formats as required [18].

VIII. Conclusion

The algorithm developed for this project provides a fantastic foundation for truly random number generation. The algorithm in its current form could quickly be set up for any application that could take JPEGs as an input. With the current design of the project there are plenty of areas that could utilize a new source of random data.

The goals of this project are not particularly unique. However, the image processing approach introduces a different method of randomization that allows for more modularity when it comes to different applications in which random number generation may be required.

IX. References

[1] https://wiki.openssl.org/index.php/Random_Numbers#Generation
[2] https://en.wikipedia.org/wiki/Lavarand
[3] https://www.cloudflare.com/learning/ssl/lava-lamp-encryption/
[4] https://www.visolve.com/ssl.html
[5] https://simple.wikipedia.org/wiki/RSA_algorithm
[6] https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm
[7] https://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard
[8] https://www.tutorialspoint.com/cryptography/advanced_encryption_standard.htm
[9] https://en.wikipedia.org/wiki/JPEG
[10] https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html
[11] https://en.wikipedia.org/wiki/Randomness_extractor
[12] https://link.springer.com/chapter/10.1007%2F3-540-39799-X_41#page-1
[13] https://en.wikipedia.org/wiki/SHA-2
[14] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf
[15] https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software
[16] https://www.cygwin.com/
[17] https://en.wikipedia.org/wiki/Photoelectric_effect
[18] https://en.wikipedia.org/wiki/Raw_image_format
[19] http://www.cs.tufts.edu/comp/116/archive/fall2013/pnixon.pdf
[20] https://link.springer.com/chapter/10.1007/978-3-642-25541-0_58
[21] https://wakespace.lib.wfu.edu/bitstream/handle/10339/62642/Li_wfu_0248M_10943.pdf