Manuscript

Security for Internet of Things

Submitted by:

Joshua SHAKYA
Ishan GAUTAM

Submitted to:

Professor Christophe GUYEUX

Written on : 11/2020

Version : v1.0

1. PREFACE

1.1 Description of the User

This document is prepared for Professor Cristophe Guyeux as a supplementary document for the project of Security for the internet of Things. This manuscript will comment on algorithms of random generators, exponential calculators, prime number generators & testers, key exchanging and asymmetric encryption algorithms prepared during the course.

1.2 Conventions Used in This Manual

The following style conventions are used in this document:

- **Bold**

Names of product elements, commands, options, programs, processes, services, and utilities
Names of interface elements (such windows, dialog boxes, buttons, fields, and menus)

Interface elements the user selects, clicks, presses, or types

- *Italic*

Publication titles referenced in text

Emphasis (for example a new term)

Variables

- `Courier`

System output, such as an error message or script

URLs, complete paths, filenames, prompts, and syntax

User input variables
< > Angle brackets surround user-supplied values
[ ] Square brackets surround optional items
| Vertical bar indicates alternate selections - the bar means “or”

1.3 Project Repository and Version Control

The source code and technical information regarding the project can be found in the github repository given below:

https://github.com/iamishan9/IOT-Security

1. DESCRIPTION OF THE PROJECT

2.1 Objective of the Project

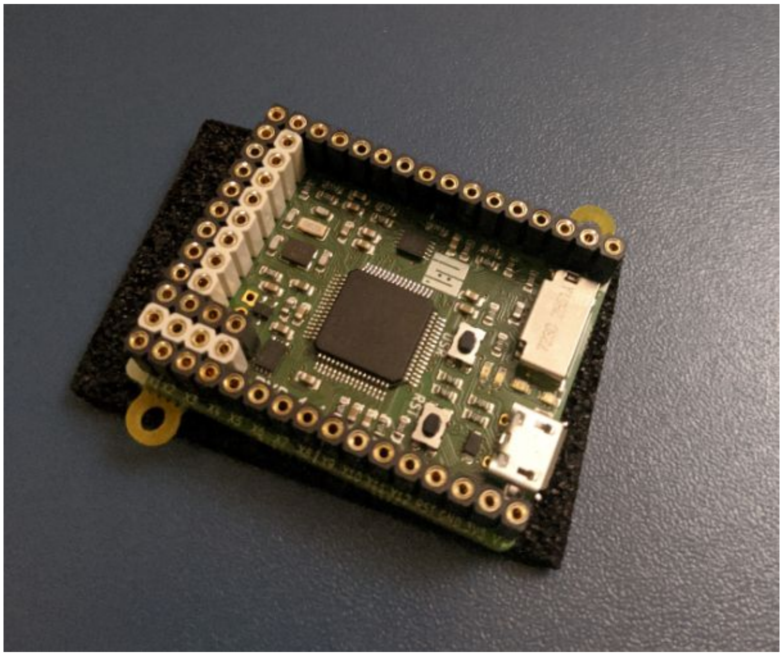
The objective of the project is to build a small security library, compatible with the resources of the Internet of Things.

2.2 Programming Language used

We used ‘Python’ programming language to develop the library.

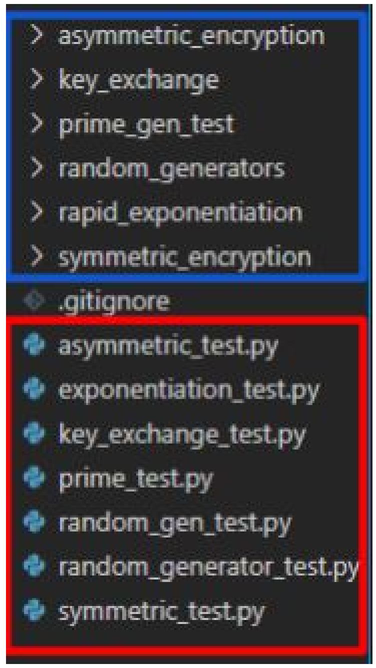
2.3 Hardware Used

We had been provided with the PyBoard to test the different algorithms that we have developed to see how efficiently they work in IOT devices.



2.4 Final Result

The final product is a Python Library with sub-libraries for each of the topics given in the website https://cours-info.iut-bm.univ-fcomte.fr/pmwiki-2.2.131/pmwiki.php/MonWiki/IotGenerateurAleaEn. We also created test files to test each algorithm belonging to each topic as shown in the screenshot below. The blue box contains each topic in the library and the red box contains the test files that we have used to verify the algorithms and respond to the questions in the document.



2. PREPARATION [HOW TO INSTALL THE PROJECT]

3.1 Downloading the project

The source code for the simulator can either be downloaded from the zip file from the Google Drive or from the git repository which was used to coordinate between the members of the project as well as to maintain version control.

3.1.1 From Zip File

The source code has been added to Drive and the link has been sent in the email. The file can be downloaded through the platform by clicking on the file and saving in one of your local directories. Then, the file can be unzipped.

3.1.2 From Git Repository

The source code can be downloaded by cloning the git repository. The command for cloning the repository is git clone <url>. So, in order to clone the simulator repository, go to the folder in which you want the source code to be saved and then use the command given below:

```
git clone https://github.com/iamishan9/IOT-Security
```

3.2 Running the project

From the folder, choose any of the test files and run using the command `python filename.py` to see the validation done for the given topic.

3. SECTIONS

4.1 SECTION 1: Symmetric encryption

For symmetric encryption, we have generated the key first by generating random bits of the same length. Then, we use LCG to generate the key and encrypt the message. The results we obtained using each of the method is shown below:

Encrypting message without using random generator (Generating random bits for the key):

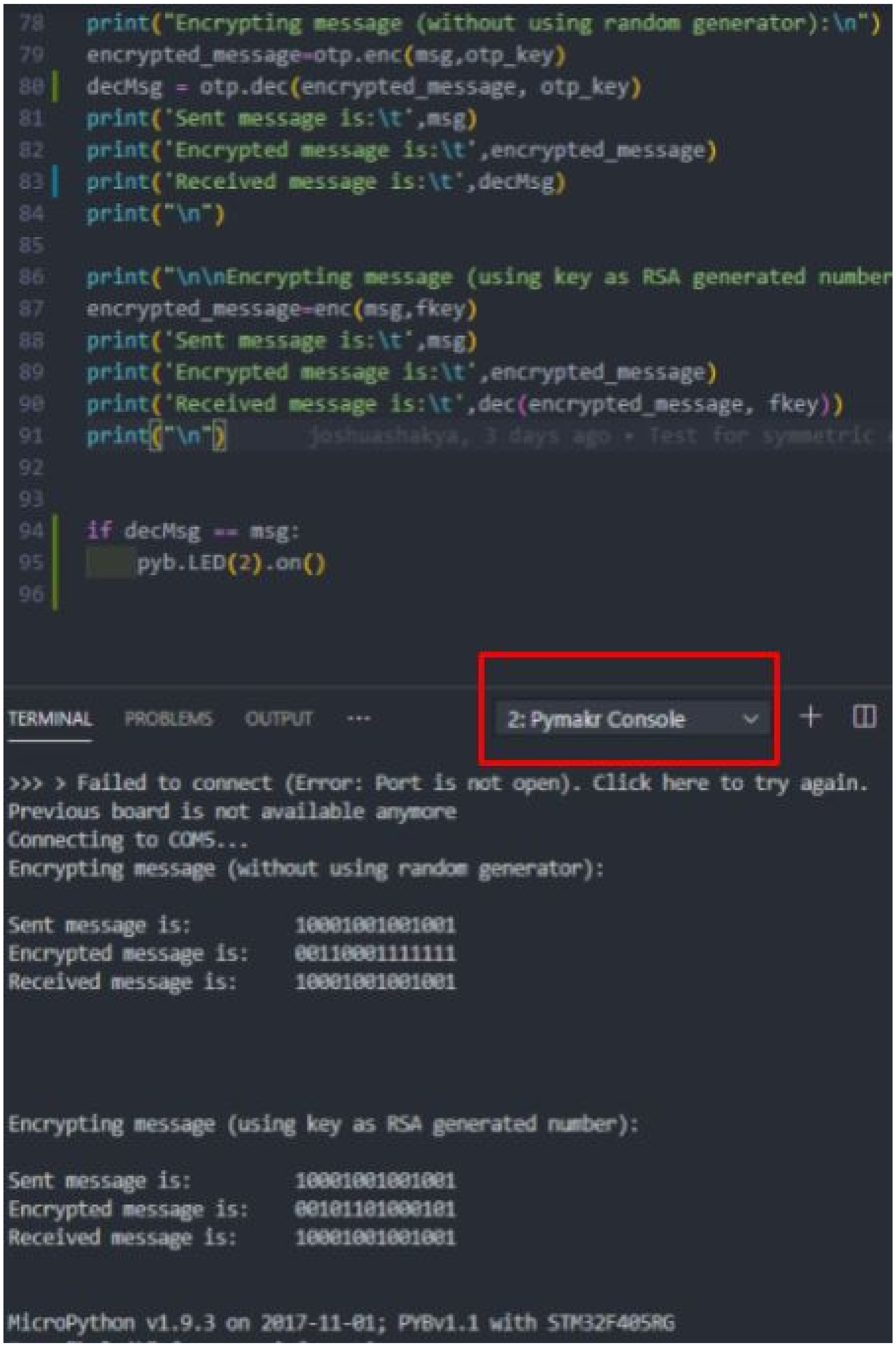
```
Encrypting message (without using random generator):

Sent message is:         10001001001001
Encrypted message is:    10010101011101
Received message is:     10001001001001
```

Encrypting message using key as random number generated by RSA algorithm.

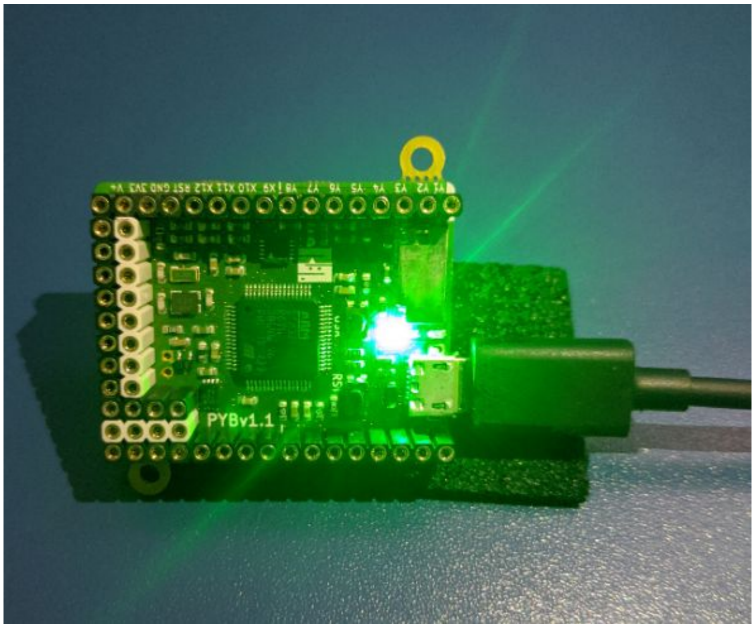
```
Encrypting message (using key as RSA generated number):

Sent message is:         10001001001001
Encrypted message is:    01000011011101
Received message is:     10001001001001
```

We configured the PyBoard to turn on GREEN LED light if the encryption and decryption is successful as shown in the photo below:

Pymakr Console



PyBoard emitting green LED light



4.2 SECTION 2: Random generators

Linear Congruential Generators

Linear Congruential Generators are the least complex random number generators in this list. We implemented it using the milliseconds part of the time as the seed and got random numbers.

```
Generating 10 random numbers using Linear Congruetional Generator (LCG):
1260999
15668882
7042437
8451452
10925268
10354320
3026373
14364492
14569357
10736344
```

- **Program this generator on the pyboard.**

  Done.

- **Find good parameters a,c,m.**

  We created a function that verifies all the conditions to be good parameters for a,c and M for LCG. We tested with several parameters and chose the best one.

```
Checking if the parameters we used for LCG are good and satisfy all requirements

With values a=1140671485, c=128201163, m=2**24
The parameters are good.

With values a=567, c=992, m=2**15
The parameters are not good.

After checking with different values, we chose the first one.
```

- **Check that the numbers produced look random.**

  To check if the numbers generated are random, we calculated the percentage of even and odd numbers, percentage of numbers below median and above or equal to median as shown in the screenshot below:

Algorithm	EVEN(%)	ODD(%)	<MEDIAN(%)	>=MEDIAN(%)
LCG	50.0	50.0	50.0	50.0
ISAAC	50.0	50.0	50.0	50.0
XOR-Shift	70.0	30.0	50.0	50.0
LSFR	60.0	40.0	50.0	50.0
Mersenne	50.0	50.0	50.0	50.0
BBS	30.0	70.0	50.0	50.0

- **Integrate this random generator tool to the previously programmed one-time pad.**

  Done in Section 4.1 (Look above).

XORShift

XorShift, also called shift-register generators were discovered by George Marsaglia. They generate numbers by repeatedly taking the exclusive or of a number with a bit shifted version of itself. We generated 10 random numbers using this algorithm as follows:

```
Generating 10 random numbers using XORshift:
703056
272506
784478
640433
546296
21235
959074
959074
959074
421281
```

- **Make this generator on PyBoard**

  Done.

- **Compare the time required for random generation using the two techniques.**

  We check the execution time of the two algorithms and confirm that LCG is faster and hence more efficient than XorShift.

```
Comparing the time required for random generation using LCG and XORshift

Algorithm       Execution time
LCG             1.2799999999923983e-05
XORshift        2.4600000000152278e-05

Hence, it is proved that LCG is faster.
```

Linear feedback shift registers

Linear-feedback shift register is a shift register whose input bit is a linear function of its previous state. In our project, we implemented **Galois LFSR** to get random numbers as shown below:

```
Generating 10 random numbers using Linear feedback shift registers(LFSR):
524299
854874
130046
352456
195240
574814
404251
306000
161111
801817
```

Mersenne-twister

Mersenne twister is the most commonly used general purpose pseudo random number generator. It is used in a lot of software systems.

```
Generating 10 random numbers using Mersenne-twister:
3521569528
1101990581
1076301704
2948418163
3792022443
2697495705
2002445460
502890592
3431775349
1040222146
```

Blum Blum Shub

Blum Blum Shub is a pseudorandom number generator that takes the form:

$$x_{n+1} = x_n^2 \bmod M,$$

We implemented it and generated random numbers as follows:

```
Generating 10 random numbers using Blum Blum Shub:
1609
2902
4890
3933
9991
8166
306
5681
4957
6133
```

ISAAC

ISAAC (indirection, shift, accumulate, add and count) is a cryptographically secure pseudorandom generator. We have implemented it in our project and got the result as in the screenshot below:

```
Generating 10 random numbers using ISAAC:
4132496584
3829983597
2564501428
4126856527
1128209147
3991672138
3627484295
1187663770
1575187518
2285375639
```

4.3 SECTION 3: Efficient power calculation

To calculate the power of a number efficiently, we developed two algorithms, one using recursion and another following the steps in the WikiMath document which does not use recursion making it very efficient for IOT devices. The results are shown below:

Power calculation with recursion for 2^500

```
Performing exponetiation with recursion:
 32733906078961418700131896968275991522166420460430647894832913680961337964046745548832700923252325
904157150886684127560071009217256545885393053328527589376
```

Power calculation without recursion for 2^500

As you can see from the screenshot below, the number of steps for the exponentiation is only **13**, if we performed naive exponentiation, it would have been 500.

```
Performing exponentiation without recursion:
 32733906078961418700131896968275991522166420460430647894832913680961337964046745548832700923252325
904157150886684127560071009217256545885393053328527589376
Number of steps for non-recursive exponentiation:13
```

Comparing the two methods

We see that for higher powers like 2^500, the method without recursion is more efficient.

```
Algorithm                  Time of execution
exp with recursion:     1.0799999999998311e-05
exp without recursion:  7.000000000000062e-06
```

4.4 SECTION 4: Prime number generators and checkers

Eratosthenes Sieve

Eratosthenes Sieve obtains prime numbers below the given number N. We implemented it in Python which prints prime numbers below 500 as shown below:

```
 Generating prime numbers between 500 and 600 using Eratosthenes Seive:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 19
7, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311,
 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 4
33, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499]
```

Some questions:

- **Why do we get the list of prime numbers below N?**

  Since it works by adding all numbers to a list and then iteratively marking as composites the multiples of each prime, it removes all the composite numbers until N, which leaves us with the list of prime numbers below N.

- **Why do we stop at N/2?**

  Since we remove all the multiples of prime numbers iteratively, at N/2 we will have already removed all the multiples(composite numbers) and left with only prime numbers.

- **Is it a good method to obtain large prime numbers in the context of the Internet of Things or even in Computer Security?**

  No, it is not a very good method to obtain large prime numbers because this method generates prime numbers from 0 to N. If we have to find a very large prime number, we will get all the prime numbers below the target prime number which is very burdensome on the memory, especially for IOT devices which have limited energy and memory.

  For example, if we are looking for 97, we will get all the other unnecessary prime numbers too (2,3,5,7,11,13,........,83,89,97).

1	**2**	**3**	4	**5**	6	**7**	8	9	10
11	12	**13**	14	15	16	**17**	18	**19**	20
21	22	**23**	24	25	26	27	28	**29**	30
31	32	33	34	35	36	**37**	38	39	40
41	42	**43**	44	45	46	**47**	48	49	50
51	52	**53**	54	55	56	57	58	**59**	60
61	62	63	64	65	66	**67**	68	69	70
71	72	**73**	74	75	76	77	78	**79**	80
81	82	**83**	84	85	86	87	88	**89**	90
91	92	93	94	95	96	**97**	98	99	100

Fermat's Primality Test and Miller Rabin Primality Test

We also implemented Fermat's primality test and Miller Rabin primality test to check if the numbers are prime or not. We generated 10 random numbers using one of the random generators (LCG) in the previous section and checked if our primality testers gave the correct results. They were 100% accurate so far.

Rand no	Fermat check	Miller Rabin check	isPrime check
9527135	False	False	False
3711273	False	False	False
14858619	False	False	False
13962236	False	False	False
536650	False	False	False
14313682	False	False	False
6733598	False	False	False
5563370	False	False	False
10935632	False	False	False
4661113	True	True	True

4.5 SECTION 5: Key exchanging

Diffie–Hellman key exchange

Diffie–Hellman key exchange is a method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols as conceived by Ralph Merkle and named after Whitfield Diffie and Martin Hellman.

The cryptographic steps are:

1. Alice and Bob publicly agree to use a modulus $p = 23$ and base $g = 5$ (which is a primitive root modulo 23).
2. Alice chooses a secret integer $\boldsymbol{a} = 4$, then sends Bob $A = g^a \bmod p$
   - $A = 5^4 \bmod 23 = 4$
3. Bob chooses a secret integer $\boldsymbol{b} = 3$, then sends Alice $B = g^b \bmod p$
   - $B = 5^3 \bmod 23 = 10$
4. Alice computes $\boldsymbol{s} = B^a \bmod p$
   - $\boldsymbol{s} = 10^4 \bmod 23 = 18$
5. Bob computes $\boldsymbol{s} = A^b \bmod p$
   - $\boldsymbol{s} = 4^3 \bmod 23 = 18$
6. Alice and Bob now share a secret (the number 18).

Both Alice and Bob have arrived at the same values because under mod p,

$$A^b \bmod p = g^{ab} \bmod p = g^{ba} \bmod p = B^a \bmod p$$

More specifically,

$$(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$$

Functions to generate keys and encrypt/decrypt the message

```python
def generate_partial_key(self):
    partial_key = self.public_key1**self.private_key
    partial_key = partial_key%self.public_key2
    return partial_key
        You, a month ago • Add most programs
def generate_full_key(self, partial_key_r):
    full_key = partial_key_r**self.private_key
    full_key = full_key%self.public_key2
    self.full_key = full_key
    return full_key
```

```python
def encrypt_message(self, message):
    encrypted_message = ""
    key = self.full_key
    for c in message:
        encrypted_message += chr(ord(c)+key)
    return encrypted_message

def decrypt_message(self, encrypted_message):
    decrypted_message = ""
    key = self.full_key
    for c in encrypted_message:
        decrypted_message += chr(ord(c)-key)
    return decrypted_message
```

Result

```
Partial keys are:
Sender:  147
Receiver:  66
Full keys generated with partial keys are:
Receiver:  75
Receiver:  75
Encrypted text sent by sender is:
 ´¾k´¾k¬kÁ°½Äk¾°®½°¿k¸°¾¾¬²°lll
Decrypted text received by receiver is:
 This is a very secret message!!!
```

4.6 SECTION 6: Asymmetric encryption

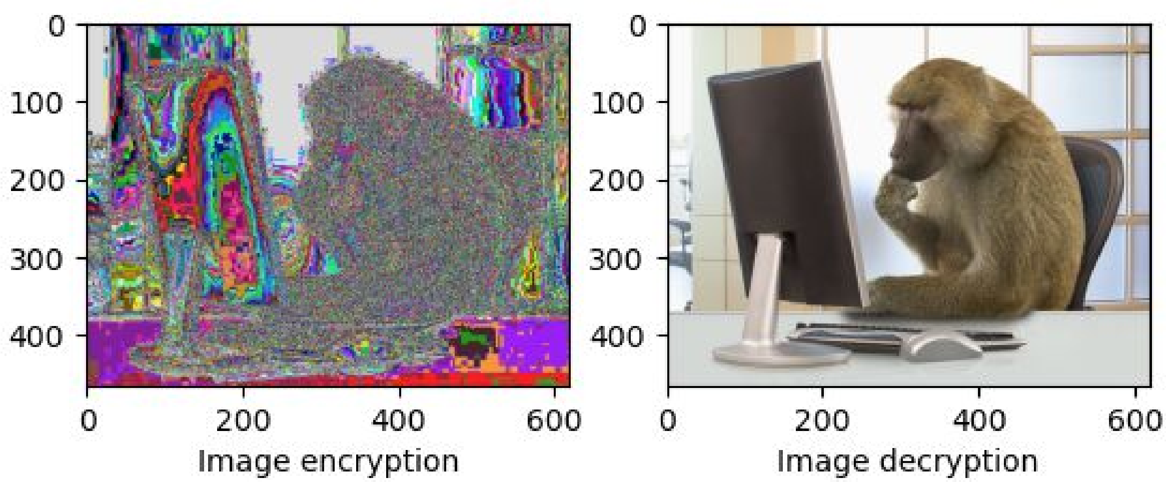
RSA text encryption and decryption

RSA is an asymmetric encryption algorithm using two big prime numbers. We generated these numbers ourselves using the algorithms in Section 4.4 and got the results as follows:

```
RSA for text

Parameters used

p1 is 29 and p2 is 23
n, c, d is  667 13 237
Encrypted Message ĜÛɜʒţʒ¯sKŝqʒqƃʒUsKq
Decrypted Message RSA by Joshua and Ishan
```

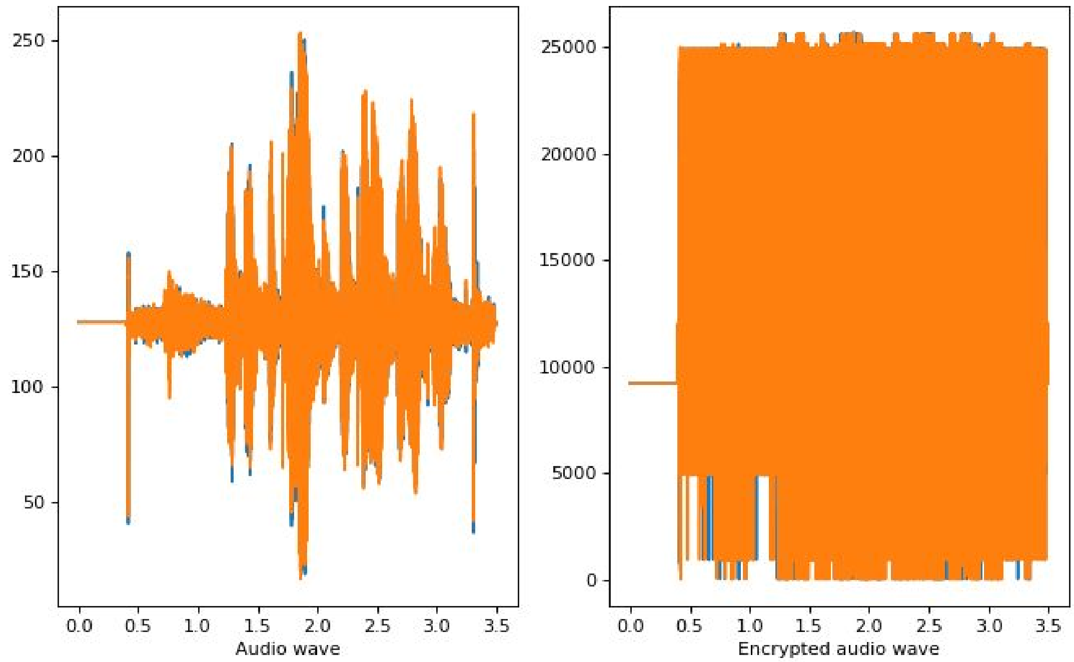
RSA image encryption and decryption

We have also extended the RSA algorithm to encrypt and decrypt image files. An example is shown in the image below:

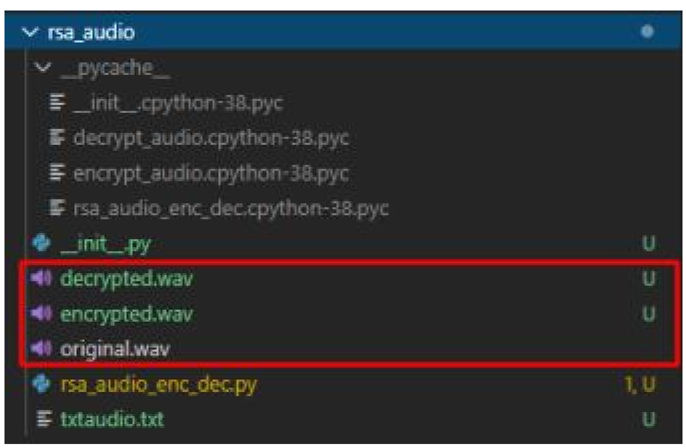


RSA audio encryption and decryption

We have encrypted an audio file, which will read a file and encrypt it so that the file is not comprehensible. Then, you can call the decrypt function to get the actual content of the file.



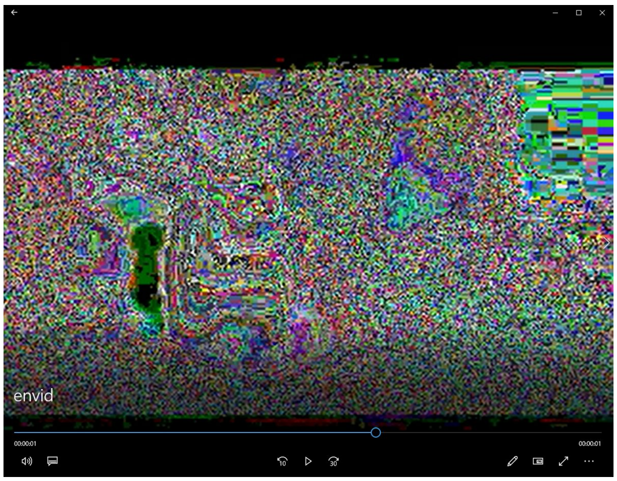
The audio files are in the folder rsa_audio. It can be listened to and verified that it has been properly encrypted.

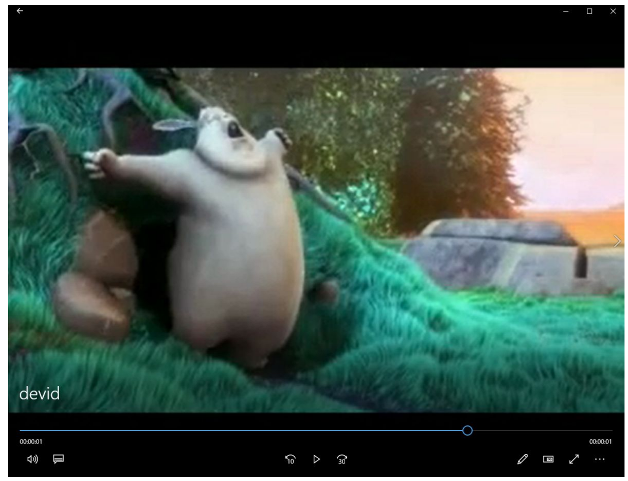


RSA video encryption and decryption

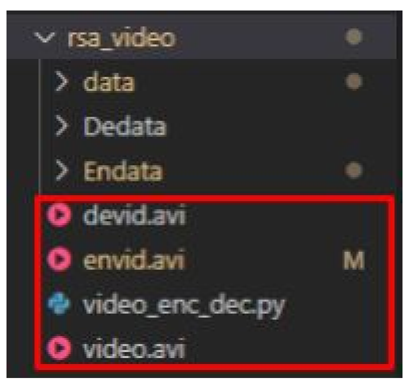
We have also encrypted a video file, which will read a video file, separate it into frames and then encrypt them. The encrypted frames will be combined to make the encrypted video that will be sent to the receiver. The receiver will follow the same steps: he will read the encrypted file, separate into frames and decrypt them and then combine the decrypted frames to make the decrypted video.

The encrypted and decrypted videos are shown below:





The video files are in the folder rsa_video. It can be watched and verified that it has been properly encrypted.



NOTE: The functions that call image, audio and video encryption are commented because they are time consuming. Uncomment them if you want to test them at the end of the file asymmetric_encryption.py.

```
check_rsa_text()
# rsa_image.start()
# rsa_audio_enc_dec.encrypt()
# rsa_audio_enc_dec.decrypt()
check_goldwasser()
check_el_gamal()
el_gamal_user_prompt()
```

If the subject interests you, you can refine it a little :

1. **by cutting the numbers to be encrypted into packets such that the n modulo does not cause any loss of information,**

   Done.

2. **by creating a function to transform text into a large number, and vice versa,**

   Done.

3. **by generalizing to other media (music, image, film, etc.)**

   Done (See above).

4. **by making an effective exponentiation,**

   Done using the functions in section 4.3.

5. **by obtaining large original prime numbers yourself.**

   Done using the functions in section 4.4.

El Gamal

El Gamal contains three main algorithms:

- Key generation

```
>>> def keys(p,g,a):
...     A = g**a%p
... public = (p,g,A)
... private = a
... return (public, private)
...
```

- Encryption

```
>>> def digit(message, public, b):
...     (p,g,A) = public
...     B = g**b%p
... c = message*A**b%p
... return (B, c)
...
```

- Decryption

```
>>> def decipher(cryptogram, public, private):
...     (p,g,A), a = public, private
...     (B,c) = cryptogram
... return B**(p-1-a)*c%p
...
```

By following the instructions, we generated a public key and a private key. Then we encrypted a message and tested if the decryption worked perfectly.

The result is shown below:

```
EL GAMAL
Generating keys first:
p: 66559259428217210332514756107100425578650502803847317526510651757086155295107
g: 42964469302472924162795175047054573471394213047237252314465601884314963424867
A: 52238536134001890261659104599763293322172616806978903909336882699077487746954
The cipher is:   21076083839738632383471803729538325014834938168492097594592255004962077
569120 54175147247526965943853039026197956510413110285365987320047841087061403324764 488071487930
59601601437872047044774415686305777861565157811974985471976513610 15029132383075367332939579046620
1632721722915232126903515769059675458428442154
Decrypted message:   We are testing El Gamal
```

The user will be prompted to generate the key, encrypt a text or decrypt it.

As per the instruction, we also asked the user to generate a key, encrypt a message or decrypt it as shown :

```
Choose 1 for generating key, 2 for encrypting message, 3 for decrypting message, 4 for quittin
he program.:
    2
Generate key first.


Choose 1 for generating key, 2 for encrypting message, 3 for decrypting message, 4 for quittin
he program.:
    1
Key generated.


Choose 1 for generating key, 2 for encrypting message, 3 for decrypting message, 4 for quittin
he program.:
    2
Enter message:Hello, this is ElGamal user prompt.
Message encrypted


Choose 1 for generating key, 2 for encrypting message, 3 for decrypting message, 4 for quitting
 the program.:
3
The encrypted message is:  2928944369454642855676243043463677575519032411444235817830571
713210704277633503 97737768024839860555132999829068922329345868670950518787337840784405910956262
 25374078558082768918470124077259843250958304824693888661048606178774869995277 5250716650778434
343973979781562756462670578083109628926159941093677626177570 54771886439373443352169311338237135
14012213386592371709269694940405707889180754 0836605337122368336087902566742115356594486640941
846676443547973210070963268

The message received is:  Hello, this is ElGamal user prompt.
```

Blum-Goldwasser cryptosystem

Blum-Goldwasser mainly uses three algorithms:

- Key generation
- Encryption
- Decryption

Key generation [edit]

The public and private keys are generated as follows:

1. Choose two large distinct prime numbers p and q such that $p \equiv 3 \bmod 4$ and $q \equiv 3 \bmod 4$.
2. Compute $n = pq$.[1]

Then n is the public key and the pair (p, q) is the private key.

Encryption [edit]

A message M is encrypted with the public key n as follows:

1. Compute the block size in bits, $h = \lfloor log_2(log_2(n)) \rfloor$.
2. Convert M to a sequence of t blocks $m_1, m_2, \ldots, m_t$, where each block is h bits in length.
3. Select a random integer $r < n$.
4. Compute $x_0 = r^2 \bmod n$.
5. For i from 1 to t
    1. Compute $x_i = x_{i-1}^2 \bmod n$.
    2. Compute $p_i =$ the least significant h bits of x_i.
    3. Compute $c_i = m_i \oplus p_i$.
6. Finally, compute $x_{t+1} = x_t^2 \bmod n$.

The encryption of the message M is then all the c_i values plus the final x_{t+1} value: $(c_1, c_2, \ldots, c_t, x_{t+1})$.

Decryption [edit]

An encrypted message $(c_1, c_2, \ldots, c_t, x)$ can be decrypted with the private key (p, q) as follows:

1. Compute $d_p = ((p+1)/4)^{t+1} \bmod (p-1)$.
2. Compute $d_q = ((q+1)/4)^{t+1} \bmod (q-1)$.
3. Compute $u_p = x^{d_p} \bmod p$.
4. Compute $u_q = x^{d_q} \bmod q$.
5. Using the Extended Euclidean Algorithm, compute r_p and r_q such that $r_p p + r_q q = 1$.
6. Compute $x_0 = u_q r_p p + u_p r_q q \bmod n$. This will be the same value which was used in encryption (see proof below). x_0 can then used to compute the same sequence of x_i values as were used in encryption to decrypt the message, as follows.
7. For i from 1 to t
    1. Compute $x_i = x_{i-1}^2 \bmod n$.
    2. Compute $p_i =$ the least significant h bits of x_i.
    3. Compute $m_i = c_i \oplus p_i$.
8. Finally, reassemble the values $(m_1, m_2, \ldots, m_t)$ into the message M.

For key generation, we needed two prime numbers which we generated using Eratosthenes sieve and Miller Rabin which we prepared in the previous section. The parameters that we chose were:

```
Parameters chosen:
p= 499
q= 491
a= -184
b= 187
r= 365
x0= 145575
ap+bq: 1
```

After choosing the parameters, we encrypted the message with the public key and then performed the decryption using the private key by following the steps in https://en.wikipedia.org/wiki/Blum%E2%80%93Goldwasser_cryptosystem.

```
Ciphertext:  01101000000110001010000100100100100011111100100011001001110011101101010010110111
10000011010000000110110110011011001011011001011001000000101000011011101001111010101110001111110
00000100100101101001000111001011101110000010010100011100110000001001110000000001111000100100011111
01111010000010100110010110000111101110001001100011010110011101010011111011101110011100000111000101
1101100000101
Decrypted message:  Hello, we are testing Blum Goldwasser encryption
```

4.RELATED DOCUMENTATION

#	Document Title	Link
1	Blum Goldwasser cryptosystem	https://en.wikipedia.org/wiki/Blum%E2%80%93Goldwasser_cryptosystem
2	Diffie-Hellman Key exchange	https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange
3	ElGamal Encryption	https://en.wikipedia.org/wiki/ElGamal_encryption