AARHUS UNIVERSITET
INGENIØRHØJSKOLEN

Network Security, E20

Group 20, Aarhus University

Anton Sakarias Rørbæk Sihm	201504954
Peter Marcus Hoveling	201508876
Viktor Gregers Søndergaar	201610466

November 6, 2020

Assignment-2 part I and II, OTP and the Rabin cryptosystem

Date for handin: November 6 2020

Lecturers: Diego F. Aranha

Table of contents

Part I - One Time Pad (OTP) Cihper

1 Introduction

In cryptography, the One-time pad (OTP) is a stream cipher, where the encryption key is of the same length as the plaintext. The OTP is said to be uncrackable, but requires its key only to be used one time, since this may open up for cracking possibilities such as frequency analysis or dictionary-based cribbing.
Each byte or character of the plaintext is encrypted by the corresponding byte from the pad, and therefor impossible to break if the four conditions are met[5]:

- The key must be truly random.
- The key must be at least as long as the plaintext.
- The key must never be reused in whole or in part
- The key must be kept completely secret.

1.1 Assignment setup

For the given assignment, we are tasked with breaking the one-time pad. This is only possible due to the fact that two plaintexts have been encrypted with the same one-time pad. Finding the two plaintexts will be done by a combination of the previously mentioned frequency analysis and dictionary-based cribbing. This in combination with some manual intervention.

Given for the one-time pad is as follows:

Two ciphertexts and one program for encrypting

- `challenge1.txt` (203 Bytes)
- `challenge2.txt` (200 Bytes)
- `otp.c` (1.005 KB)

For verification, the two ciphertexts with the following hash values (calculated with sha256sum):

```
fdb8bb2642cb7c9a1869ab99019e3ee3eaff5160c0cf5c8aec1267742e941eb1 challenge1.txt
```

```
44e79e85e1aaa37ba74a4a77a7fb15f2da43bca494f1b6ff3ab3eb493b68c24f challenge2.txt
```

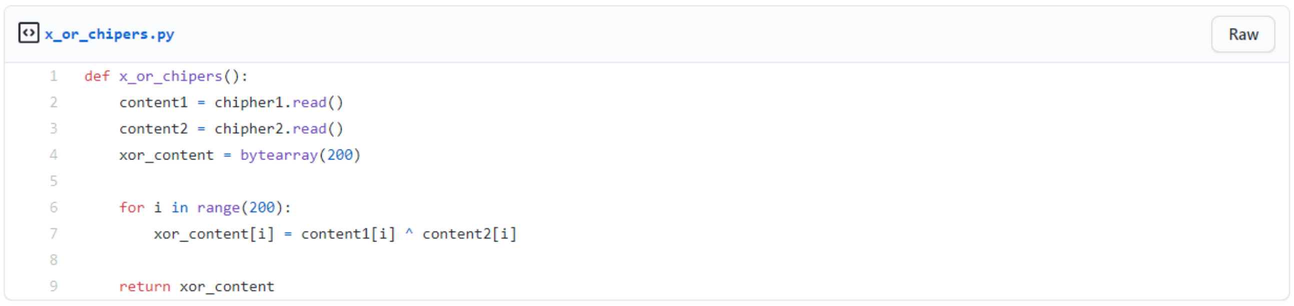
2 Implementation

For the actual cracking of the two ciphers we have chosen to use the "Grib Drag" method, which involves guessing words that may be in the ciphertext via. some manual human interaction. The "Grib Drag" method has the following steps:

1. $Cipher1 \oplus Cipher2 = IntermediateCipher$
2. Guess a word that might be in one of the ciphers
3. $Word \oplus IntermediateCipher$ at index 0
4. Check for occurrences of readable words
5. $Word \oplus IntermediateCipher$ at next index
6. Iterate step 4 and 5 by sliding the word from index 0 to (len(IntermediateCipher)-len(word))
7. Register readable occurrences at their index.

2.1 XOR ciphers - step 1

Given the two given cihpers size: $Cipher1\_size = 200$ and $Cipher2\_size = 203$ we can only calculate the $IntermediateCipher$ with a size of $min\{Cipher1\_size, Cipher2\_size\} = 200$. This also means that we have to guess the last 3 bytes in $Cipher2$. As seen in figure 1, the intermediate cipher is calculated by XOR'ing each byte in $Cipher1$ with each byte in $Cipher2$.



x_or_chipers.py

```python
def x_or_chipers():
    content1 = chipher1.read()
    content2 = chipher2.read()
    xor_content = bytearray(200)

    for i in range(200):
        xor_content[i] = content1[i] ^ content2[i]

    return xor_content
```

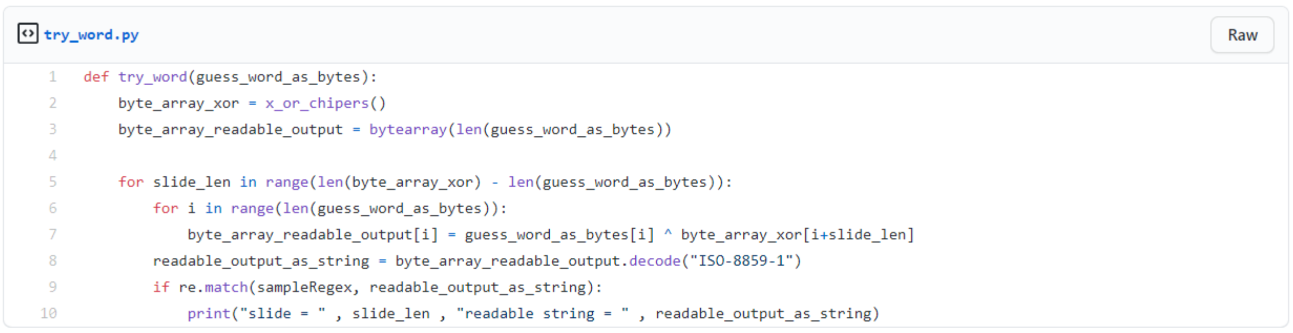
Figure 1: Calculation of intermediate cipher

2.2 English word analysis - step 2

Given the insecure properties of an OTP encryption with key reusage, it is now possible to start attacking the encryption by XOR'ing words into the intermediate cipher. Instead of picking random words from the English vocabulary, statistical data of common used English words [2] can be used for a higher success rate in uncovering words in the ciphers. The most common used word in the English language is the word "The", so there is a high chance that "The" is in the cipher. Probably the word "The" will also have leading and trailing empty spaces " The ".

2.3 Sliding algorithm - step 3

By sliding each guess-word from index 0 to (len(IntermediateCipher)-len(word)), we check each place in the ciphers where this guess word might appear. The sliding algorithm as seen figure 2, slides the chosen guess-word on each index in the intermediate cipher.



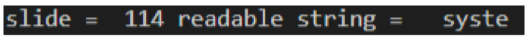
try_word.py

```python
def try_word(guess_word_as_bytes):
    byte_array_xor = x_or_chipers()
    byte_array_readable_output = bytearray(len(guess_word_as_bytes))

    for slide_len in range(len(byte_array_xor) - len(guess_word_as_bytes)):
        for i in range(len(guess_word_as_bytes)):
            byte_array_readable_output[i] = guess_word_as_bytes[i] ^ byte_array_xor[i+slide_len]
        readable_output_as_string = byte_array_readable_output.decode("ISO-8859-1")
        if re.match(sampleRegex, readable_output_as_string):
            print("slide = " , slide_len , "readable string = " , readable_output_as_string)
```

Figure 2: Sliding algorithm

Lets take an example of the sliding algorithm: On the list of common words the word "people" appears. By sliding the word "people" into the intermediate cipher we will get an output size of $output\_size = (len(IntermediateCipher) - len(word))$ which has to be analysed manually for occurrences of words that might be correct. By using the word "people", we can see i figure 3 that the word " syste" appears on index 114, which probably is the word " system" in one of the ciphers.



```
slide =  114 readable string =   syste
```

Figure 3: Sliding the word "people"

2.4 Filter trash - step 4-5-6

Reasoned that each guess-word generates an output size of $output\_size = (len(IntermediateCipher) - len(word))$, it is hard to find viable occurrences of English words in the massive output. To overcome this issue, regex can be used to reduce the output, by filtering random characters that aren't used frequently in the English language. The used regex is ”$\hat{}[a-zA-Z0-9_,.- : ”'] * \$$”

- **No Regex :** Output size with word "people" $= (len(IntermediateCipher) - len(word)) =$ 200-6 $=$ 194
- **With Regex :** Output size with word "people" $=$ 36

Using regex speeds up the manual analysing process.

2.5 Reveal Keys algorithm

When the complete plaintext has been recovered, the keys used to encrypt the two messages are fairly easy to recover as well. We know that each character(r) has been XOR'ed with some key(k) in order to produce the first character in the cihper(c). As seen in the OTP encryption formula 1, the only unknown variable is the key(k) which is used to encrypt the character(r). Since each key is a byte, the key has to be within 0-255. For the greedy approach we can try each key until it matches the correct cipher value.

$$r_i \oplus k_i = c_i \tag{1}$$

By doing so with the plaintext text of the biggest size, we will recover the largest amount of keys which is 203. Note that the $c1$ and $c2$ was encrypted with a key of size 205. The last two keys are unrecoverable due to the lack length of the $c1$ and $c2$.

2.6 Usage instruction

Looking at figure 4, we see the root directory for exercise one. In here lies the two cyphertexts, `challenge1.txt` and `challenge2.txt`, the text file `cracked.txt` which holds the found plaintext words and finally the python program `find_words.py`.

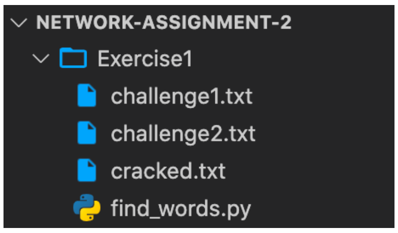
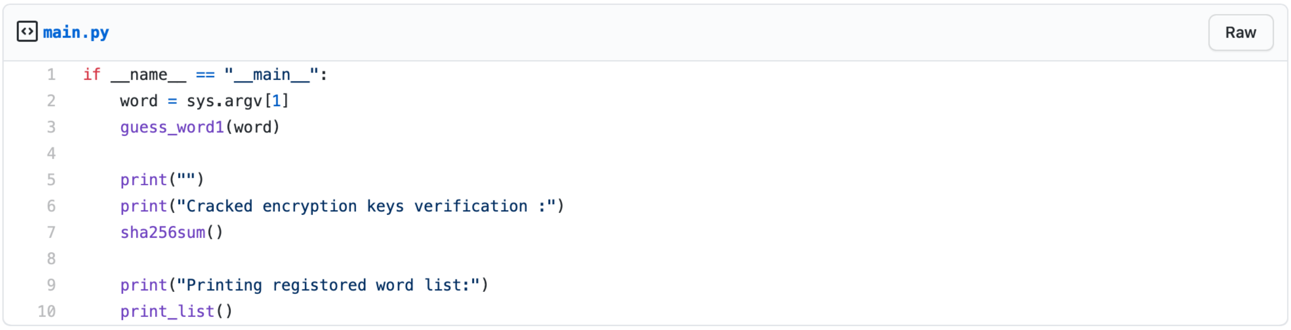


Figure 4: Root director for exercise I

Looking at the programs main function, as seen in figure 5, we see that the programs reads the argument from the command line and calls `guess_word1(word)`. This initiates the program steps as mentioned in section section 2 and via. the use of regular expressions, shows the possible valid words in the console.


main.py

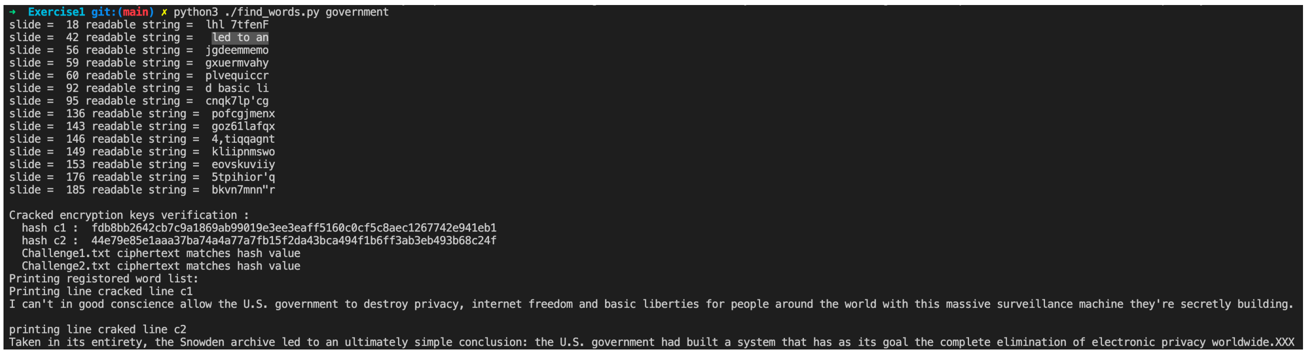
```python
if __name__ == "__main__":
    word = sys.argv[1]
    guess_word1(word)

    print("")
    print("Cracked encryption keys verification :")
    sha256sum()

    print("Printing registored word list:")
    print_list()
```

Figure 5: `main()` function in program

For every valid word found, the user will manually insert the characters in same place of the grib drag slide number. When the entire plaintext is thought to be found the `sha256sum()` will be called to validate the calculated hash with the original. If these match, the plaintext is found!
Looking at figure 6, we can see the user trying the word 'government' and finding a valid word combination ("led to an") at slide 42.



```
➜  Exercise1 git:(main) ✗ python3 ./find_words.py government
slide =  18 readable string =  lhl 7tfenF
slide =  42 readable string =   led to an
slide =  56 readable string =  jgdeemmemo
slide =  59 readable string =  gxuermvahy
slide =  60 readable string =  plvequiccr
slide =  92 readable string =  d basic li
slide =  95 readable string =  cnqk7lp'cg
slide =  136 readable string =  pofcgjmenx
slide =  143 readable string =  goz61lafqx
slide =  146 readable string =  4,tiqqagnt
slide =  149 readable string =  kliipnmswo
slide =  153 readable string =  eovskuviiy
slide =  176 readable string =  5tpihior'q
slide =  185 readable string =  bkvn7mnn"r

Cracked encryption keys verification :
  hash c1 :  fdb8bb2642cb7c9a1869ab99019e3ee3eaff5160c0cf5c8aec1267742e941eb1
  hash c2 :  44e79e85e1aaa37ba74a4a77a7fb15f2da43bca494f1b6ff3ab3eb493b68c24f
  Challenge1.txt ciphertext matches hash value
  Challenge2.txt ciphertext matches hash value
Printing registored word list:
Printing line cracked line c1
I can't in good conscience allow the U.S. government to destroy privacy, internet freedom and basic liberties for people around the world with this massive surveillance machine they're secretly building.

printing line craked line c2
Taken in its entirety, the Snowden archive led to an ultimately simple conclusion: the U.S. government had built a system that has as its goal the complete elimination of electronic privacy worldwide.XXX
```

Figure 6: Output of program

Part II - Rabin cryptosystem

3 Introduction

In cryptography, the Rabin cryptosystem is an asymmetric cryptographic technique. It is related to the difficulty of integer factorization and is therefor valued as being strong. Like all asymmetric cryptosystems, the Rabin system uses a public key for encryption and a private key for decryption. The cryptosystem has the disadvantage, that each output of the Rabin function can be generated by any of four possible inputs. That means extra complexity, in the form of implementation, is needed to identify which of the four possible inputs is the correct output.[6]

3.1 Assignment setup

For the given assignment we were tasked with implementing a Rabin cryptosystem, that could support key generation, encryption and decryption. We should also justify the selection and solution to the disambiguation problem, on how can one can detect what is the right plaintext among the four possibilities produced by the Rabin decryption. We decided to implement the Rabin Cryptosystem in Python.

4 Implementation

4.1 Generate Prime Numbers & Decoding Generating Public Key

To generate the prime numbers used to generate the public key we use the package PyCryptodome[1]. If we go into the source code of PyCryptodome we see that it uses IsPrime that generates a prime with a bias. IsPrime tries to generate a prime then checks for evenness and for pre-calculated Sieves Primes. If the potential prime passes both checks it performs the Rabin-Miller Primality test[7] 10 times and that means the false positive in the worst case is $1e - 6$. An example of this test can be seen on figure 7. To make sure we use carefully selected random numbers we use the PyCryptodome get_random_bytes as a random function as it uses SystemRandom.

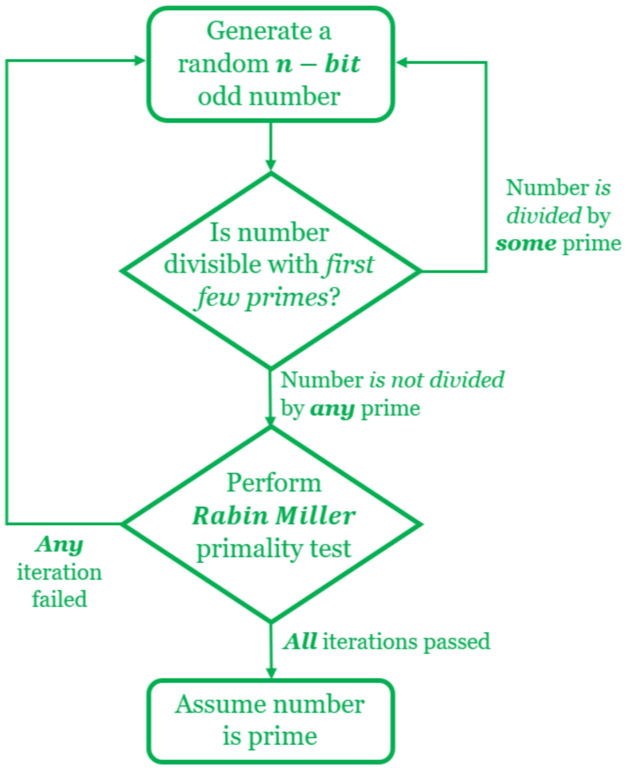

Generate a random *n − bit* odd number
Is number divisible with *first few primes?*
Number *is divided* by **some** prime
Number *is not divided* by **any** prime
Perform **Rabin Miller** primality test
Any iteration failed
All iterations passed
Assume number is prime

Figure 7: Flow diagram for generating a Prime

When we have found the primes p and q we also have to make sure that our primes complies with

$$p \bmod 4 = 3 \text{ and } q \bmod 4 = 3 \tag{2}$$

If thats the case we then generate the public key n by computing

$$n = p * q \tag{3}$$

With this we have the keypair where n is the public key and the pair(p,q) is the private key

4.2 Padding

Before encrypting the desired message, there is a need for some kind of padding to provide a way to automate the possibility to select the correct plaintext after decryption. The decryption, which will be described later, will end up having 4 possible plaintexts. For a human selecting the correct plaintext might be easy, as it's readable. However for a program this might not be the case.
The padding will be simple as seen in formula 4. Concatenate message(m) with itself, makes up a pattern in the 4 possibilities decrypted ciphertexts where 0 to $\frac{n}{2}$ amount of bytes must be equal to the $\frac{n}{2}$ to n amount of bytes. This pattern enables the program to recognise the correct plaintext after decryption.

$$padding(m) = m + m \tag{4}$$

4.3 Encryption method

To encrypt the message we first use our Encoding method to convert it to a number and then we compute.

$$cipher = m^2 \mod n \tag{5}$$

where m is the message, and n our public key

4.4 Decryption Method

To decrypt the message we need to do 3 steps.

1. First we compute the square root of c modulo p and q using these two formulas:

$$m_p = m^{(p+1)/4} \mod p \tag{6}$$

$$m_q = m^{(q+1)/4} \mod q \tag{7}$$

2. Secondly we use the **Extended-euclidean-algorithm**[4] to find y_p and y_q

$$y_p \cdot p + y_q \cdot q = 1 \tag{8}$$

3. Thirdly we use the Chinese remainder theorem[3] to find the four square roots of c modulo n

$$r_1 = (y_p \cdot p \cdot m_q + y_q \cdot q \cdot m_p) \mod n \tag{9}$$

$$r_2 = n - r1 \tag{10}$$

$$r_3 = (y_p \cdot p \cdot m_q - y_q \cdot q \cdot m_p) \mod n \tag{11}$$

$$r_4 = n - r3 \tag{12}$$

After calculating r_1, r_2, r_3 and r_4 we have 4 possible plaintexts which each could be the original plaintext. Because of the padding added, the correct plaintext can be fetched. The correct plaintext is the one there the bytes from 0 to $\frac{n}{2}$ is equal to the $\frac{n}{2}$ to n amount of bytes.

5 Usage instruction

Looking at figure 8, we see the root directory for exercise two. In this directory we only have one file, `rabin.py`. An important detail for this project is the requirement of one packages `pycryptodomex`. This python package is for generating cryptographic primitives used as q and p.

- Link to PyCryptodome: https://pycryptodome.readthedocs.io/en/latest/src/introduction.html

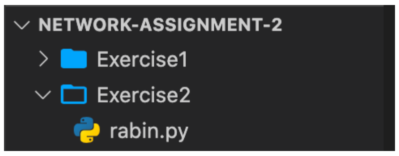
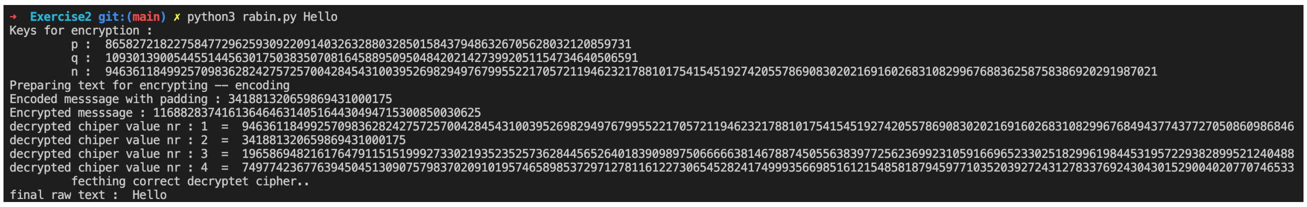


Figure 8: Root directory for exercise II

Running the program is done through the CLI. The user calls the program with the desired word to be encrypted. Looking at the output, as shown in figure 9, the two 1536-bit private-key primes are generated and then after the public key n is calculated.
The plaintext/word is then encoded with padding and then encrypted. The encrypted message is then decrypted as four different byte values and scanned for the correct decrypted cipher.



```
➜  Exercise2 git:(main) ✗ python3 rabin.py Hello
Keys for encryption :
         p :  8658272182275847729625930922091403263288032850158437948632670562803212085973 1
         q :  1093013900544551445630175038350708164588950950484202142739920511547346405065 91
         n :  946361184992570983628242757257004284543100395269829497679955221705721194623217881017541545192742055786908302021691602683108299676883625875838692029 1987021
Preparing text for encrypting -- encoding
Encoded messsage with padding : 34188132065986943100017 5
Encrypted messsage : 11688283741613646463140516443049471530085003062 5
decrypted chiper value nr : 1  =  94636118499257098362824275725700428454310039526982949767995522170572119462321788101754154519274205578690830202169160268310829967684943774377270508609868 46
decrypted chiper value nr : 2  =  34188132065986943100017 5
decrypted chiper value nr : 3  =  196586948216176479115151999273302193523525736284456526401839098975066666381467887450556383977256236992310591669652330251829961984453195722938289952124048 8
decrypted chiper value nr : 4  =  749774236776394504513090757983702091019574658985372971278116122730654528241749993566985161215485818794597710352039272431278337692430430152900402077074653 3
         fecthing correct decryptet cipher..
final raw text :  Hello
```

Figure 9: Output of `rabin.py` program

6 Bibliography

Websites

[1] PyCryptodome. *PyCryptodome Package.* URL: `https://pycryptodome.readthedocs.io/en/latest/src/introduction.html#`.

[2] Rype. *Common English Words.* URL: `https://www.rypeapp.com/most-common-english-words/`.

[3] Wikipedia. *Chinese remainder theorem.* URL: `https://en.wikipedia.org/wiki/Chinese_remainder_theorem`.

[4] Wikipedia. *Extended Euclidean algorithm.* URL: `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`.

[5] Wikipedia. *One-time pad.* URL: `https://en.wikipedia.org/wiki/One-time_pad`.

[6] Wikipedia. *Rabin Cryptosystem.* URL: `https://en.wikipedia.org/wiki/Rabin_cryptosystem`.

[7] Wikipedia. *Rabin Primality Test.* URL: `https://en.wikipedia.org/wiki/Miller-Rabin_primality_test`.