Rabin Cryptosystem

Alberto Boffi

DISCRETE MATHEMATICS

POLITECNICO DI MILANO - JAN. 2024

Contents

1 Introduction **2**

2 Key Generation **2**
2.1 Description and Implementation 2

3 Encryption and Decryption **4**
3.1 Description and Implementation 4
3.2 Example of Usage 8

4 Text Management **9**
4.1 Description and Implementation 9
4.2 Example of Usage 14

5 Conclusion **15**

References **15**

1 Introduction

This project consists of building a Python implementation of the Rabin cryptosystem by exploiting the notions of abstract algebra and number theory learned in the course.
The backbone of the implementation includes two main classes: `KeyGenerator`, that deals with the production of the keys, and `RabinCryptosystem`, which exploits the KeyGenerator to implement the Rabin cryposystem. In addition, the `RabinTextManager` class supports the user by adapting the RabinCryptosystem to messages in string format rather than integers.

Each of these modules is described in detail in the following sections.

The full repository is hosted at: https://github.com/albertoboffi/rabin-cryptosystem

2 Key Generation

2.1 Description and Implementation

For the key generation, the Rabin cryptosystem requires two integers p and q such that the following requirements are fulfilled [1]:

$$\begin{cases} p \equiv 3 \, mod \, 4 & (1) \\ q \equiv 3 \, mod \, 4 & (2) \\ p \text{ and } q \text{ are "large enough"} & (3) \\ p \text{ and } q \text{ are primes} & (4) \\ p \neq q & (5) \end{cases}$$

The public and private (or "secret") key are then [1]:

$$\begin{cases} p_k = p \cdot q \\ s_k = (p, q) \end{cases}$$

The `KeyGenerator` class generates the keys through the following sequence of operations:

- Method `__isValidPrime(p:int):int` ensures (1) and (2) by returning true if and only if $p \equiv 3 \, mod \, 4$.
- Method `__getRandomSeed():int` ensures (3) by generating a random `seed` in the range $[1000, 10000)$, that corresponds to the minimum **rank** of p and q. It does it by exploiting the `secrects` module, which, unlike the popular `random` module, is cryptographically secure [2].
- Method `__generatePrime(q=0):int` ensures (1) (2) (3) (4) (5) by generating prime numbers starting from a rank equal to the seed, until they are valid and distinct. It does so by relying on the `primality` module, specifically designed to support arithmetic on prime numbers [3].

- The public method `generateKeys():dict` generates p and q and constructs the keys accordingly.

```python
#!/usr/bin/env python

import secrets
from primality import primality

__author__ = 'Alberto Boffi'
__deprecated__ = False

class KeyGenerator:

    # Input: Prime p
    # Output: True if p  3 mod 4, False otherwise

    def __isValidPrime(self, p: int) -> bool:

        if (p % 4 == 3): return True
        return False

    # Input: -
    # Output: Random number n, st the generated prime will be at least the n-th prime

    def __getRandomSeed(self) -> int:

        seed_range = range(1000, 10000)
        seed = secrets.choice(seed_range)

        return seed

    # Input: -
    # Output: Random large prime p such that p  3 mod 4

    def __generatePrime(self, q = 0) -> int:

        seed = self.__getRandomSeed()

        p = primality.nthprime(seed)

        i = 0
        while (not(self.__isValidPrime(p) or (p == q))):
           p = primality.nthprime(seed + i)
           i += 1

        return p
```

```python
# Input: -
# Output: Key pair for the Rabin cryptosystem

def generateKeys(self) -> dict:

    q = self.__generatePrime()
    p = self.__generatePrime(q)

    k_pri = (p, q)
    k_pub = p * q

    return {

        "private": k_pri,
        "public": k_pub

    }
```

3 Encryption and Decryption

3.1 Description and Implementation

The `RabinCryptosystem` class, when initialized, exploits the `KeyGenerator` to produce the public and private keys to employ during the communication.

The encryption procedure is developed as follows [1]:

- *Plaintext*: $x \in \mathbb{Z}$ such that $x < p_k$
- x is extended with the same digits composing x itself (e.g. $x_i = 239 \rightarrow x_f = 239239$). This operation is needed in order to recognize the original plaintext once the ciphertext has been decrypted.
- $c = x^2 \ mod \ p_k$
- *Ciphertext*: c

The decryption approach is slightly more elaborate [1]:

- *Ciphertext*: c
- Given the private key s_k, it first computes the roots of c modulo p and q: $s_p = c^{0.25(p+1)} \ mod \ p$, $s_q = c^{0.25(q+1)} \ mod \ q$. To achieve this, I employed the `Decimal` module, enabling the handling of floating-point numbers with extremely high precision [4].

- It exploits the *Extended Euclidean Algorithm*, which I specifically implemented, to find the Bézout coefficient k_p and k_q of p and q, such that $k_p p + k_q q = 1 = GCD(p, q)$.
- It finds the four possible plaintexts as shows in system (6), by means of the *Chinese Remainder Theorem*.
- For each of the possible plaintexts ptx_i, it checks whether $ptx_i = \alpha\alpha$, where α is a generic word in the alphabet $\{0, 1, .., 9\}$. Let ptx_t be the plaintext that verifies this condition.
- *Plaintext*: ptx_t

$$\begin{cases} ptx_0 = (k_p \cdot p \cdot s_q + k_q \cdot q \cdot s_p) \; mod \; p_k \\ ptx_1 = p_k - ptx_0 \\ ptx_2 = (k_p \cdot p \cdot s_q - k_q \cdot q \cdot s_p) \; mod \; p_k \\ ptx_3 = p_k - ptx_2 \end{cases} \tag{6}$$

Please Note: In general, there is nothing preventing the existence of an additional possible plaintext ptx_i that satisfies the condition $ptx_i = \alpha\alpha$, $\alpha = \{0, 1, .., 9\}^+$. However, the probability of this occurring is so low as to be negligible in practical cases.

```python
#!/usr/bin/env python

from .KeyGenerator import KeyGenerator
from .algorithms import *

from decimal import *
getcontext().prec = 1000000 # adjust precision to deal with large numbers

__author__ = 'Alberto Boffi'
__deprecated__ = False

class RabinCryptosystem:

    # Input: -
    # Output: -
    # Behavior: Initialize the key generator and generates the keys

    def __init__(self):

        self.key_generator = KeyGenerator()
        self.generateKeys()

    # Input: Original Plaintext
    # Output: New Plaintext
```

```python
# Behavior: Adds to the original plaintext a prefix equal to the plaintext itself

def __addPrefix(self, plaintext:int) -> int:

    s_pt = str(plaintext)
    pref_pt = int(s_pt + s_pt)

    return pref_pt

# Input: Candidate extended plaintexts
# Output: Correct plaintext
# Behavior: Finds the number that is repeat twice in one of a candidate extended plainte

def __getPlaintext(self, plaintexts: list) -> None:

    for i in range(0, 4):

        s_pt = str(plaintexts[i])
        len_pt = len(s_pt)

        fh = s_pt[: len_pt // 2]
        sh = s_pt[len_pt // 2 :]

        if (fh == sh): return int(fh)

# Input: -
# Output: -
# Behavior: Generates a key pair

def generateKeys(self) -> None:

    keys = self.key_generator.generateKeys()
    self.k_pri = keys["private"]
    self.k_pub = keys["public"]

# Input: Plaintext
# Output: Ciphertext
# Behavior: Encrypts the plaintext using the public key

def encrypt(self, plaintext: int) -> int:

    extended_plaintext = self.__addPrefix(plaintext)

    ciphertext = (extended_plaintext ** 2) % self.k_pub

    return ciphertext
```

```python
# Input: Ciphertext
# Output: All four possible plaintexts
# Behavior: Decrypts the ciphertext using the private and the public key

def decrypt(self, ciphertext: int) -> int:

    # prime numbers composing the private key

    p = self.k_pri[0]
    q = self.k_pri[1]

    # square roots modulo private key

    sq_p = ciphertext ** (Decimal('0.25') * (p + 1)) % p
    sq_q = ciphertext ** (Decimal('0.25') * (q + 1)) % q

    sq_p = int(sq_p)
    sq_q = int(sq_q)

    # bezout coefficients

    coef = extEuclideanAlgorithm(p, q)["coef"]

    coef_p = coef[0]
    coef_q = coef[1]

    # square roots

    plaintext_1 = (coef_p * p * sq_q + coef_q * q * sq_p) % self.k_pub
    plaintext_2 = self.k_pub - plaintext_1
    plaintext_3 = (coef_p * p * sq_q - coef_q * q * sq_p) % self.k_pub
    plaintext_4 = self.k_pub - plaintext_3

    plaintexts = [

        plaintext_1,
        plaintext_2,
        plaintext_3,
        plaintext_4

    ]

    return self.__getPlaintext(plaintexts)
```

Extended Euclidean Algorithm:

```python
#!/usr/bin/env python

__author__ = 'Alberto Boffi'
__deprecated__ = False

# Input: r0 = a, r1 = b such that a > b > 0
# Output: gcd, Bezout coefficients
# Bahvior: Performs the extended version of the Euclidean algorithm

def extEuclideanAlgorithm(r0: int, r1: int, s0 = 1, s1 = 0, t0 = 0, t1 = 1):

    q = r0 // r1

    r_new = r0 - q * r1
    s_new = s0 - q * s1
    t_new = t0 - q * t1

    if (r_new):

        return extEuclideanAlgorithm(r1, r_new, s1, s_new, t1, t_new)

    return {

        "gcd": r1, # greatest common divisor
        "coef": [s1, t1] # bezout coefficients

    }
```

3.2 Example of Usage

```python
from src.RabinCryptosystem import RabinCryptosystem

#---------------------------------------------------#
#--------------- Possible messages -----------------#
#----------- CHANGE WITH YOUR OWN MESAGE -----------#
#---------------------------------------------------#

msg = 741

#---------------------------------------------------#
#--------------- Body of the script ----------------#
#------------------- DO NOT TOUCH ------------------#
#---------------------------------------------------#
```

```python
def main(msg):

    # encryption

    cryptosystem = RabinCryptosystem()

    ciphertext = cryptosystem.encrypt(msg)
    plaintexts = cryptosystem.decrypt(ciphertext)

    # log

    print("Original message >> ", msg)
    print("Encrypted message >> ", ciphertext)
    print("Decrypted message >> ", plaintexts)


if __name__ == '__main__':

    main(msg)
```

Output:

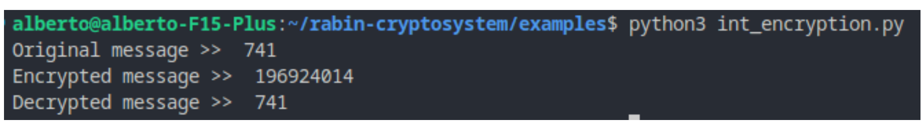

Figure 1: Integer Encryption Output

4 Text Management

4.1 Description and Implementation

The Rabin cryptosystem deals with integers plaintexts and ciphertexts. However, in most applications, it's necessary to encrypt texts, and thus the plaintext and ciphertext need to be generic strings. This kind of management is not straightforward and for this reason I decided to implement it in a specific class, named `RabinTextManager`.

In general, given the set Σ of all possible ASCII characters, the problem can be reduced to finding two functions:

$$f : \Sigma^* \to \mathbb{Z}, \;\; g : \mathbb{Z} \to \Sigma^*$$

such that the following requirements are fulfilled:

$$\begin{cases} f \ is \ bijective & (7) \\ f(\alpha) < p_k \forall \alpha \in \Sigma^* & (8) \\ g \ is \ bijective & (9) \end{cases}$$

Given f and g, the encryption (10) and decryption (11) of a generic string can be implemented as schematized below:

$$str_{ptx} \xrightarrow{f} int_{ptx} \xrightarrow{E} int_{ctx} \xrightarrow{g} str_{ctx} \tag{10}$$

$$str_{ctx} \xrightarrow{g^{-1}} int_{ctx} \xrightarrow{D} int_{ptx} \xrightarrow{f^{-1}} str_{ptx} \tag{11}$$

where E and D are, respectively, the encryption and decryption functions of the Rabin cryptosystem.

About function f, ensuring (8) on a text α of arbitrary length means that, speaking from an information theory perspective, $f(\alpha)$ decrease the amount of information stored in α, preventing (7) from being guaranteed. For this reason, the message is broken down into each individual character that compose it, and each of them is encrypted separately. Then, the ciphertexts are concatenated together. Formally, this means reducing the function f to a function f' where:

$$f' : \Sigma \rightarrow \mathbb{Z}$$

At this point, finding a function f' such that (8) and (7) are guaranteed is quite easy:

$$f(c) = ord(c)$$

where ord returns the ASCII code corresponding to c.

The last step is finding g, whose input is the result of the encryption. Actually, such a result is an integer too large to be interpreted as an ASCII code. This makes sense as we remind that the encryption function extends the plaintext doubling its length. So we have to use g to convert the integer into two different characters.
Unfortunately, there isn't a straightforward way to convert the result into a pair of ASCII characters in a bijective manner. The reason lies in the fact that the ciphertext in the Rabin cryptosystem doesn't have a standard length (specifically, in our case, it's an integer ranging from 8 to 10 digits). To understand it better, let's consider a practical example.

The most practical approach to build g consists in splitting the integer into two parts of equal length (or differing by just one digit if the total digits are 9). Each half is then treated as a separate ASCII character. For instance:

$$g(456398133) = chr(4563) \ . \ chr(98133)$$

where *chr* returns the ASCII character corresponding to the given code.

However, the function is not bijective due to the leading zeros issue in the second half. For instance, let's assume the input is **653200218**. Once divided into the two parts **6532**, **00218**, the corresponding ASCII character for each part is produced in output. Subsequently, g^{-1} should retrieve the integers corresponding to the two ASCII characters, and these would be **6532** and **218**.
Since the ciphertext doesn't have a standard length (it could have either an even or odd number of digits), there is no way to know if the original ciphertext was **65320218** or **653200218**.

Moreover, whatever method is used to translate the integer into a pair of characters, the unknown length always poses a challenge to a reversible translation.

Fortunately, the structure of the Rabin cryptosystem comes to the rescue. In fact, if the ciphertext has been incorrectly reconstructed (in the previous case, **65320218** instead of **653200218**), the decryption fails and triggers an error. Therefore, both possible ciphertexts are considered, and in case an error is generated, it is intercepted and the other ciphertext is used.

```python
#!/usr/bin/env python

from .RabinCryptosystem import RabinCryptosystem

__author__ = 'Alberto Boffi'
__deprecated__ = False

class RabinTextManager:

    # Input: -
    # Output: -
    # Behavior: Initializes the Rabin cryptosystem

    def __init__(self):

        self.crypsys = RabinCryptosystem()

    # Input: Text
    # Output: List of characters of the strings
    # Behavior: Splits the text in each character composing it

    def __splitText(self, text:str, split_size:int) -> list:

        text_split = [

            text[i:i + split_size]
```

```
            for i in range(0, len(text), split_size)
        ]

        return text_split

    # Input: Integer
    # Output: Two-characters string
    # Behavior: Transforms the ciphertext into a text by using ascii encoding

    def __getTextFromCiphertext(self, ciphertext:int) -> str:

        s_ciphertext = str(ciphertext)

        fh = s_ciphertext[:len(s_ciphertext) // 2]
        sh = s_ciphertext[len(s_ciphertext) // 2 :]

        fchar = chr(int(fh))
        schar = chr(int(sh))

        text = fchar + schar

        return text

    # Input: Two-character string
    # Output: Integer
    # Behavior: Converts the text into the original ciphertext using ascii decoding

    def __getCiphertextFromText(self, text:str) -> tuple[int, int]:

        fciph = ord(text[0])
        sciph = ord(text[1])

        s_fciph = str(fciph)
        s_sciph = str(sciph)

        s_sciph = ("0" * (len(s_fciph) - len(s_sciph))) + s_sciph # adds possible leading ze

        ciphertext = int(s_fciph + s_sciph)
        alt_ciphertext = int(s_fciph + "0" + s_sciph) # considering additional possible lea

        return ciphertext, alt_ciphertext

    # Input: Text
    # Output: Encrypted text
```

```python
# Behavior: Encrypts the text using the Rabin cryptosystem

def encrypt(self, text:str) -> str:

    chars = self.__splitText(text, 1) # extracts each character from the text

    enc_text = ""

    for i in range(len(chars)):

        plaintext = ord(chars[i]) # converts each character in the corresponding ascii c
        ciphertext = self.crypsys.encrypt(plaintext)

        enc_text += self.__getTextFromCiphertext(ciphertext) # converts the integer ciph

    return enc_text

# Input: Text
# Output: Decrypted text
# Behavior: Decrypts the text using the Rabin cryptosystem

def decrypt(self, text:str) -> str:

    chars = self.__splitText(text, 2) # splits the text into groups of two characters

    dec_text = ""

    for i in range(len(chars)):

        ciphertext, alt_ciphertext = self.__getCiphertextFromText(chars[i]) # converts t
        plaintext = self.crypsys.decrypt(ciphertext)

        try:

            dec_text += chr(plaintext) # converts the integer-format plaintext into the

        except TypeError: # the wrong ciphertext has been considered, switch

            plaintext = self.crypsys.decrypt(alt_ciphertext)
            dec_text += chr(plaintext)

    return dec_text
```

4.2 Example of Usage

```python
from src.RabinTextManager import RabinTextManager

#---------------------------------------------------#
#--------------- Possible messages -----------------#
#----------- CHANGE WITH YOUR OWN MESAGE -----------#
#---------------------------------------------------#

msg = "Hello world! What a beautiful day :)"

#---------------------------------------------------#
#--------------- Body of the script ----------------#
#------------------ DO NOT TOUCH -------------------#
#---------------------------------------------------#

def main(msg):

    # encryption

    text_manager = RabinTextManager()

    ciphertext = text_manager.encrypt(msg)
    plaintexts = text_manager.decrypt(ciphertext)

    # log

    print("Original message >> ", msg)
    print("Encrypted message >> ", ciphertext)
    print("Decrypted message >> ", plaintexts)


if __name__ == '__main__':

    main(msg)
```

Output:

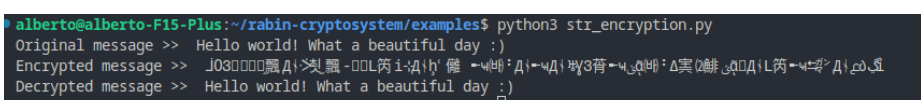

Figure 2: String Encryption Output

5 Conclusion

The successful implementation of the Rabin cryptosystem not only fortified my understanding of cryptographic principles, but also demonstrated the real-world relevance of the theoretical knowledge acquired throughout the course, providing a valuable hands-on experience in the realm of discrete mathematics and its applications in information security.

References

[1] J. Rothe. *Rabin's Public-Key Cryptosystem*. https://ccc.cs.uni-duesseldorf.de/~rothe/CRYPTOCOMPLEXITY2/folien-4-rabin.pdf. Accessed: Dec. 2023.

[2] *random — Generate pseudo-random numbers — docs.python.org*. https://docs.python.org/3/library/random.html. [Accessed 02-01-2024].

[3] *primality — pypi.org*. https://pypi.org/project/primality/. [Accessed 02-01-2024].

[4] *decimal — Decimal fixed point and floating point arithmetic — docs.python.org*. https://docs.python.org/3/library/decimal.html. [Accessed 02-01-2024].