Laboratory 4

Mos Daniela, 935

The Rabin cryptosystem

The Rabin cryptosystem is an asymetric cryptosystem which is based on the problem of finding modular square roots and is usually compared with RSA's integer factorization problem. The drawback of this algorithm is the fact the decryption will generate 4 solutions which need to be analyzed in order to get the correct message. This search can be improved by adding a number of redundant bits to the original message and then operate on that version.

Prerequisites

The alphabet used is the 26 letters from the English alphabet and the space, as follows below. The plaintext will be split in smaller blocks of size k, mapped to the corresponding values in base 27, and the resulting ciphertext will be of blocks of lenght l. The splitting is done in order to protect the data from plaintext attacks, since the public key is accesible. Also, the boundaries for the public key are between 27^k and 27^l.

```
<<Constants>>=

import random
import math
import sys

alphabet = {
    ' ': 0, 'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5, 'F': 6, 'G': 7,
    'H': 8, 'I': 9, 'J': 10, 'K': 11, 'L': 12, 'M': 13, 'N': 14,
    'O': 15, 'P': 16, 'Q': 17, 'R': 18, 'S': 19, 'T': 20, 'U': 21,
    'V': 22, 'W': 23, 'X': 24, 'Y': 25, 'Z': 26,
}
alphabet_len = len(alphabet.keys())
alphabet_keys = list(alphabet.keys())
k = 11
l = 15
dup = 11
boundaries = (alphabet_len ** k, alphabet_len ** l)
```

```
primes_boundary = (int(math.sqrt(boundaries[0])),
                   int(math.sqrt(boundaries[1])))

@
```

Key generation

The algorithm needs two types of keys: the private ones, which are two large primes p and q, with the property $p = q = 3 \bmod 4$ (to simplify the process of decryption), and a public key n = p * q.

```
<<Keys>>=

def check_prime(a):
    a = abs(a)
    if a == 0 or a == 1:
        return False
    for i in range(2, int(math.sqrt(a)) + 1):
        if a % i == 0:
            return False
    return True

def random_primes():
    nr = random.randint(primes_boundary[0], primes_boundary[1])
    while not check_prime(nr) or nr % 4 != 3:
        nr = random.randint(primes_boundary[0], primes_boundary[1])
    return nr

def get_private_key():
    a = random_primes()
    b = random_primes()
    return a, b

def get_public_key(priv):
    p, q = priv
    return p*q
@
```

Encryption

The encryption function is $f(m) = m^2 \bmod n$, $f : Z_n \to Z_n$ and the result is the ciphertext c. This function is applied on each block of text, splitted before. If the text is shorter than the expected block, add as many empty spaces as needed. And as mentioned before, the result of the decryption will generate 4 solutions which need to be analyzed, so the redundant bits are added before encrypting the plaintext.

```
<<Encryption>>=

def encrypt(plaintext, public_key):
    ciphertext = ""
    blocks = split_text(plaintext, k)
    for block in blocks:
        number = text_to_number(block)

        bin_text = number_to_binary(number)
        dup_number = binary_to_number(bin_text + bin_text[-dup:])

        c = (dup_number ** 2) % public_key

        ciphertext += number_to_text(c, l)

    return ciphertext

@
```

Decryption

The decryption function is $f^{-1} : Z_n \to Z_n$ and $f^{-1}(c)$ is one of the 4 solutions given by the Chinese remainder theorem. After splitting, each block is then converted to its numerical value and checked if it is a quadratic residue mod q or p, using the Law of Quadratic Reciprocity. Afterwards, the solutions are obtained using the modular square root algorithm but the private keys where chosen in such a way that they will follow the branch with x $= 3 \bmod 4$, $x \in \{p, q\}$. Then the final four solutions are computed using the Chinese remainder theorem and the last bits are checked. The accepted solutions are the ones that have the last and second to last *dup* bits duplicated.

```
<<Decryption>>=

def decrypt(ciphertext, private_keys):
    p, q = private_keys
    blocks = split_text(ciphertext, l)
    message = ""
    for block in blocks:
        current_block = text_to_number(block)
        reduce_p = current_block % p
        reduce_q = current_block % q

        check1 = is_quadratic_residue(reduce_p, p)
        check2 = is_quadratic_residue(reduce_q, q)

        if check1 == -1 and check2 == -1:
```

```
            break
        if check1 == -1:
            sol2 = mod_sqr_root_p(reduce_q, q)
            solutions = [sol2 % pub, -sol2 % pub]
        if check2 == -1:
            sol1 = mod_sqr_root_p(reduce_p, p)
            solutions = [sol1 % pub, -sol1 % pub]
        else:
            sol1 = mod_sqr_root_p(reduce_p, p)
            sol2 = mod_sqr_root_p(reduce_q, q)
            solutions = chinese_rem_thr(pub, pub//p, pub//q, sol1, sol2)

        for solution in solutions:
            bintext = number_to_binary(solution)
            if bintext[-(2 * dup):-dup] == bintext[-dup:]:
                number = binary_to_number(bintext[:-dup])
                text = number_to_text(number, k)
                message += text

    return message

@
```

Mathematical and helper functions

Modular square root

Finding a modular square root means finding the values of x where $x^2 = a$ mod p, where $a, p \in Z$. This equation has no solution if a is a quadratic residue mod p. Normally, there are 3 cases:
1. p = 1 mod 8
2. p = 3 mod 4
3. p = 5 mod 8

but since we stated from the beginning that the private keys p and q are chosen to verify the second case, the function reduces to the following:

```
<<Modular Sqr Root>>=
def mod_sqr_root_p(a, p):
    return repeated_sqr(a, (p + 1) // 4, p)
@
```

Quadratic residue modulo p

A quadratic residue modulo p is number a such that it verifies the following $x^2 = a$ mod p, where $a, p \in Z$. In other words, it is a number that has a square

root modulo p. In order to determine if a number is a quadratic residue modulo p, one can use the Law of Quadratic Reciprocity and the Legendre symbol.

Legendre symbol

Let p be an odd prime number and $a \in Z_p$. The Legendre symbol is used to determine whther a is a quadratic residue modulo p and it is defined as follows:

$$(\frac{a}{p}) = a^{\frac{p-1}{2}} \quad mod \quad p = \begin{cases} 1, & a \quad is \quad a \quad quadratic \quad residue \quad mod \quad p \quad and \quad a \neq mod \quad p \\ -1, a & is \quad a \quad non-quadratic \quad residue \quad mod \quad p \\ 0, if & a = 0 \quad mod \quad p \end{cases}$$

Properties

Let $a, b \in Z$ and p, q odd primes.
1. $(\frac{ab}{p}) = (\frac{a}{p})(\frac{b}{p}) \rightarrow (\frac{b^2}{p}) = 1$
2. $(\frac{1}{p}) = 1$ and $(\frac{-1}{p}) = (-1)^{\frac{p-1}{2}}$
3. $$(\frac{2}{p}) = (-1)^{\frac{p-1}{8}} = \begin{cases} 1, p = \pm 1 & mod \quad 8 \\ -1, p = \pm 3 & mod \quad 8 \end{cases}$$

Law of Quadratic Reciprocity

$$(\frac{q}{p}) = (-1)^{\frac{(p-1)(q-1)}{4}} (\frac{p}{q}) = \begin{cases} -\frac{p}{q}, p = q = 3 & mod \quad 4 \\ \frac{p}{q}, otherwise \end{cases}$$

Jacobi symbol

In order to find out if a composed number is a quadratic residue mod p, one needs to factor out that number and apply Legendre symbol and the Law of Quadratic Reciprocity on the factors. The Jacobi symbol is a generalization of the Legendre symbol and is defined as follows:
Let $a, n \in Z$, n postitive odd, then $(\frac{a}{n}) = \prod (\frac{a}{p_i})_i^k$, i=1, r and $n = \prod p_i^{k_i}$

The properties are similar with the ones from the Legendre symbol, only that instead of the primes p and q, we have the odd integers m and n. In this way, the factorization is reduced to factoring out powers of two and applying the properties and the Law of Quadratic Reciprocity to find if an integer is a quadratic residue mod n.

```
<<Jacobi>>=

def is_quadratic_residue(a, n):
    result = 1
    while a != 0:
        while a % 2 == 0:
            if n % 8 in [3, 5]:
                result = -result
            a /= 2
        a, n = n, a
        if a % 4 == 3 and n % 4 == 3:
            result = -result
        a %= n
    if n == 1:
        return result
    else:
        return 0

@
```

Chinese remainder theorem

This is a simplified version of the algorithm that will result in 4 solutions, since we know that there are 2 moduli and 4 integers. It also uses the Extendend Euclidean algorithm to obtain the modular inverses.

```
<<Chinese Rem>>=

def extended_euclid(x, m):
    a, b, u = 0, m, 1
    while x > 0:
        q = b // x
        x, a, b, u = b % x, u, x, a - q * u
    if b == 1:
        return a % m

def chinese_rem_thr(n, n1, n2, a1, a2):
    k1 = extended_euclid(n1, n2)
    k2 = extended_euclid(n2, n1)
    x1 = (a1 * n1 * k1)
    x2 = (a2 * n2 * k2)
    solutions = [
        x1 + x2,
        x1 - x2,
        -x1 + x2,
        -x1 - x2
```

```
    ]
    return [x % n for x in solutions]

@
```

Repeated squaring modular exponentiation

```
<<Repeated squaring>>=

def repeated_sqr(nr, pow, n):
    numbers = [nr % n]
    powers = [1]
    i = 2
    while i < pow + 1:
        new_nr = numbers[-1] ** 2 % n
        numbers.append(new_nr)
        powers.append(i)
        i *= 2
    return compose_number(pow, numbers, powers) % n


def compose_number(power, numbers, results):
    results.reverse()
    numbers.reverse()
    result = []
    for p, n in zip(results, numbers):
        if power - p > 0:
            result.append(n)
            power -= p
        if p == 1:
            result.append(n)
            break

    prod = 1
    for x in result:
        prod *= x
    return prod

@
```

Helper functions

Various helper functions used for converting between text, numbers and binary strings and for reading and splitting the text.

```
<<Helpers>>=

def open_file(filename):
    text = ""
    with open(filename, "r") as f:
        lines = f.readlines()
    for line in lines:
        text += line.strip()
    return text.upper()

def text_to_number(block):
    letter_sum = 0
    for index, letter in enumerate(block):
        letter_sum += alphabet[letter] * (alphabet_len ** (len(block) - index - 1))
    return letter_sum

def number_to_text(number, size):
    letters = ""
    index = 1
    try:
        while number > 0:
            position = number // (alphabet_len ** (size - index))
            number -= position * (alphabet_len ** (size - index))
            index += 1
            letters += alphabet_keys[position]
    except IndexError:
        print(f"Index error {position}")
    return letters

def number_to_binary(block):
    return "{0:b}".format(block)

def binary_to_number(bintext):
    return int(bintext, 2)

def split_text(text, size):
    blocks = []
    if len(text) % size != 0:
        text += " " * (len(text) % size)
    for i in range(0, len(text), size):
        blocks.append(text[i:(i + size)])
    return blocks

@
```

Putting everything together

The application accepts a file as argument and prints the public key, the ciphertext and the decrypted message. The file must respect the specified alphabet and the text will be transformed to uppercase at runtime. To run the application, the command should be of the form `py lab4.py filename`.

```
<<*>>=

<<Constants>>
<<Helpers>>
<<Modular Sqr Root>>
<<Jacobi>>
<<Chinese Rem>>
<<Repeated squaring>>

<<Keys>>
<<Encryption>>
<<Decryption>>

if __name__ == "__main__":

    try:
        priv = get_private_key()
        pub = get_public_key(priv)
        print("Public key {}\n".format(pub))
        message = open_file(sys.argv[1])
        print("Message {}\n".format(message))
        ciphertext = encrypt(message, pub)
        print("Ciphertext {}\n".format(ciphertext))
        plaintext = decrypt(ciphertext, priv)
        print("Best choice {}\n".format(plaintext))

    except Exception as e:
        print(e)

@
```