Homework 3: Playing with PK schemes
RSA Implementation

CNS Course Sapienza

Riccardo Salvalaggio 1750157

5/11/2020

1 Introduction to Asymmetric Encryption

One of the types of encryption is the Asymmetric one, also known as public-key encryption, that is a form of data encryption where the encryption key (also called the public key) and the corresponding decryption key (also called the private key) are different. A message encrypted with the public key can be decrypted only with the corresponding private key. The public key and the private key are related mathematically, but it is computationally infeasible to derive the private key from the public key. Therefore, a recipient could distribute the public key widely. Anyone can use the public key to encrypt messages for the recipient and only the recipient can decrypt them.

2 Asymmetric vs. Symmetric Encryption

Asymmetric cryptography typically gets used when increased security is the priority over speed and when identity verification is required, as the latter is not something symmetric cryptography supports. Asymmetric encryption provides extremely secure key exchanging procedure using huge computations that implies slow executions.
Some of the most common use cases for asymmetric cryptography include:
- Digital signatures: Confirming identity for someone to sign a document.
- Blockchain: Confirming identity to authorize transactions for cryptocurrency.
- Public key infrastructure (PKI): Governing encryption keys through the issuance and management of digital certificates.

Symmetric cryptography typically gets used when speed is the priority over increased security.
Some of the most common use cases for symmetric cryptography include:
- Banking: Encrypting credit card information or other personally identifiable information (PII) required for transactions.
- Data storage: Encrypting data stored on a device when that data is not being transferred.

Due to the drawbacks of both encryption methods, in most of the real-world cases is good habit to use Hybrid encryption: the symmetric one for the encryption and the other for the key exchanging.

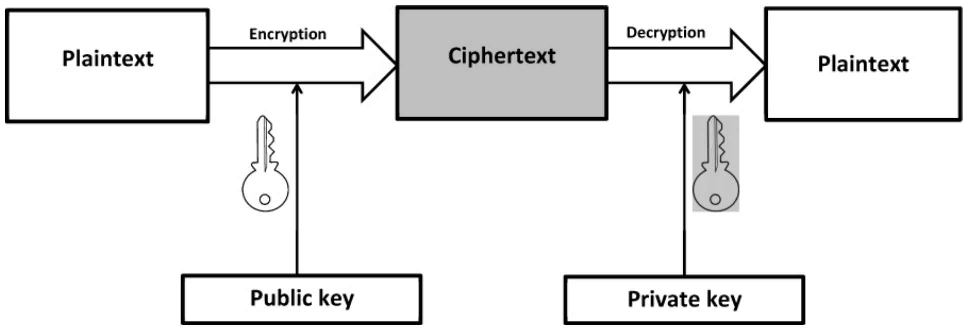


Figure 1: Asymmetric Encryption scheme

3 RSA (Ron Rivest, Adi Shamir, Leonard Adleman)

3.1 Scheme

The basic scheme of RSA is based on huge computational complexity and factorization of prime numbers.
1. We choose randomly two huge prime numbers: p and q (often choose 1024 bit).
2. Calculate $n = p * q$ and $\Phi(n) = (p - 1)(q - 1)$
3. We consider factorization of n as secret.
4. Then, we choose "e" co-prime with $\Phi(n)$ and in $[1,\Phi(n) - 1]$
5. We calculate d such that e*d = 1mod$\Phi(n)$

Public key = (n,e).
Private key = (n,d).
The power of the algorithm is that is based on computational complexity of modular mathematics and on the difficulty for a machine to solve it.

3.2 A simple example

We have two users that want to communicate in a secure way using RSA.
Bob:

$p = 3$
$q = 11$
$n = 3 * 11$

$\Phi(n) = (p-1)(q-1) = 2 * 10 = 20$
$e = 3$
Sends $k_{pub} = (n, e)$ to Alice.

Alice:

$x = 4(plaintext)$
$y = x^e = 4^3 = 31 mod 33$
Sends y, the ciphertext, to Bob.

Bob:

$d = e^{-1} = 7 mod 20$ such that: $d * e = 1 mod 20$
$x = y^d = 31^7 = 4 mod 33$

3.3 Possible problems

1. x larger than N:
In this case, after decryption, we'll have a different plaintext from the original, because x becomes the remainder of the division.
2. Exponentiation for very large exponent is not very efficient: even when using square-and-multiply, RSA is significantly slower than AES.
3. Avoid "corner case" messages such as x = 0, x = 1, x = n-1.
4. Using same p multiple times, an attacker can easily recover q1 and q2.
5. If x and e are small then: y = $x^e < n$. Adversary can easily get the plaintext by computing: $sqrt^e y$.

3.4 Implementation in python

In figures at the end of the document, there is the code of my implementation.
Here I have done some implementations decisions due to computational complexity of some operation (especially power and prime numbers search).
1. randchoose():
my initial implementation used a file with a huge list of prime numbers accessed in this way:

fo = open('primes-to-1000k.txt', 'r')

```
lines = fo.read().splitlines()
fo.close()
p = int(lines[random.randint(100, 300)])
q = int(lines[random.randint(100, 300)])
```

Anyway file reading is such slower than functions of sympy library and requires a great work to create that file for huge prime numbers.

2. gcd(): is the classic recursive implementation for gcd calculation.
3. pubkey(): do some calculations already cited in section 3.1. We also use randchoose() to calculate p and q and gcd() to verify that is equal to 1.
4. encrypt(): here we do the encrypting calculation. In the first implementation I used basic formula: (x**e)mod n, anyway that function do the real calculation that is really huge due to the number we choose; instead, pow() use low-level procedures such as shifts.

In main function (figure 3) we group all the calculation in order to execute the algorithm.

In figure 4 we have a real execution.
As you can see this implementation permits the encryption of short strings due to implementation requirements but it can be simply expanded using operational modes to divide text in blocks and encrypt one per time. I've also tried encrypting char per char to expand the encryption to long documents, anyway this leads to security problem because each char has a corresponding code so it would be simply decrypted using letter frequency analysis.

4 Comparison with real-world implementations

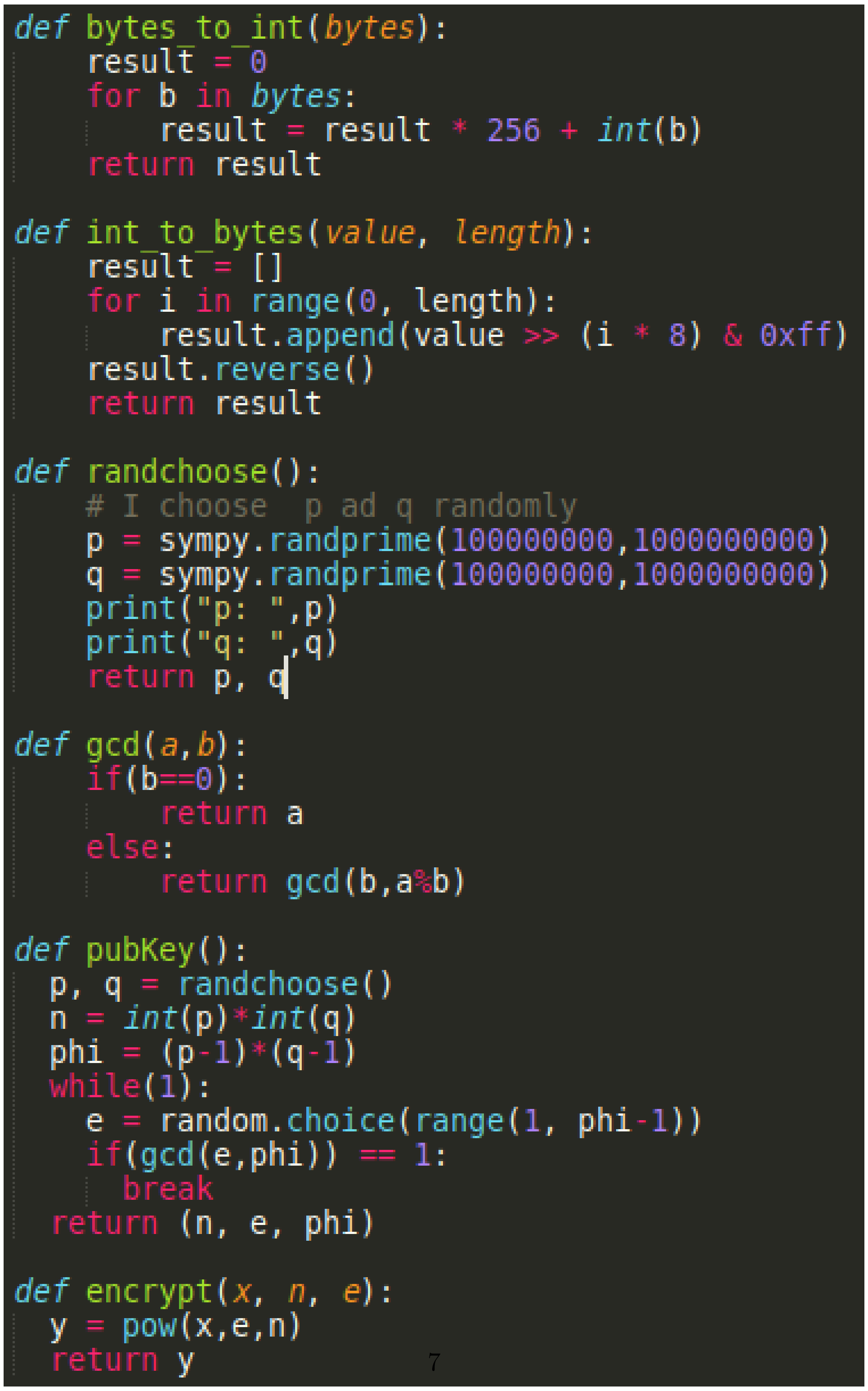
I've also compared execution time of my implementation with two well-known implementation based on standard and extremely efficient libraries. One is an RSA implementation, the other an AES CTR Block Mode. You can see comparison in figure 4,5,6 at the end of the document.

References

[1] ”RSA” page on wikipedia.com

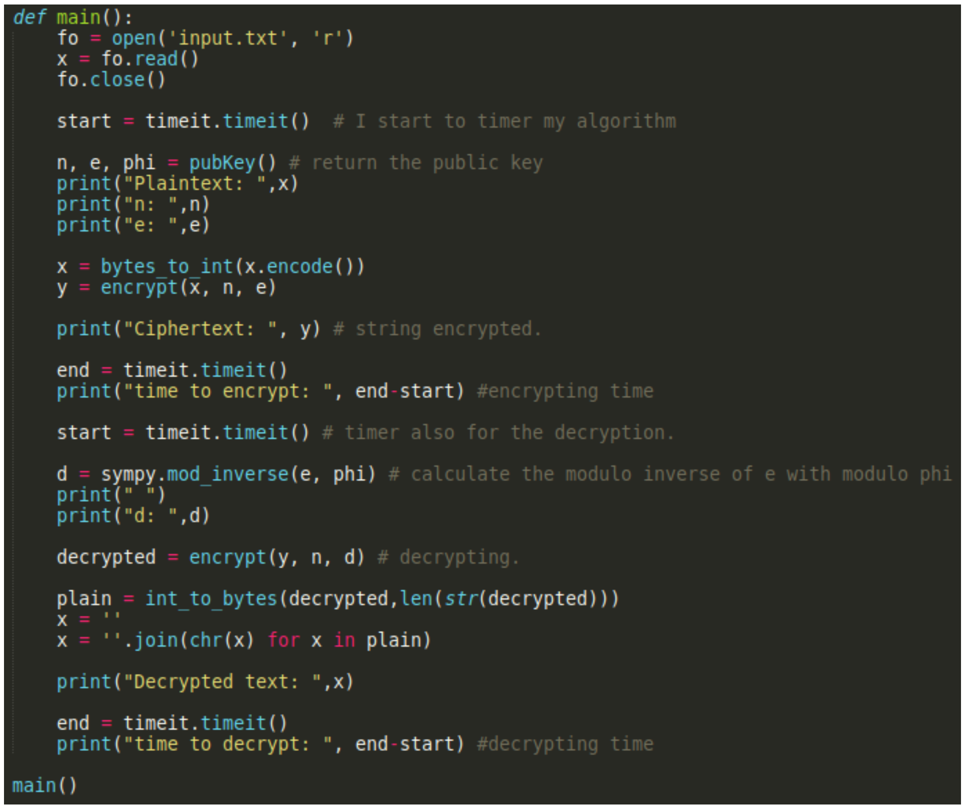
[2] securityboulevard.com

[3] https://link.springer.com/referenceworkentry/



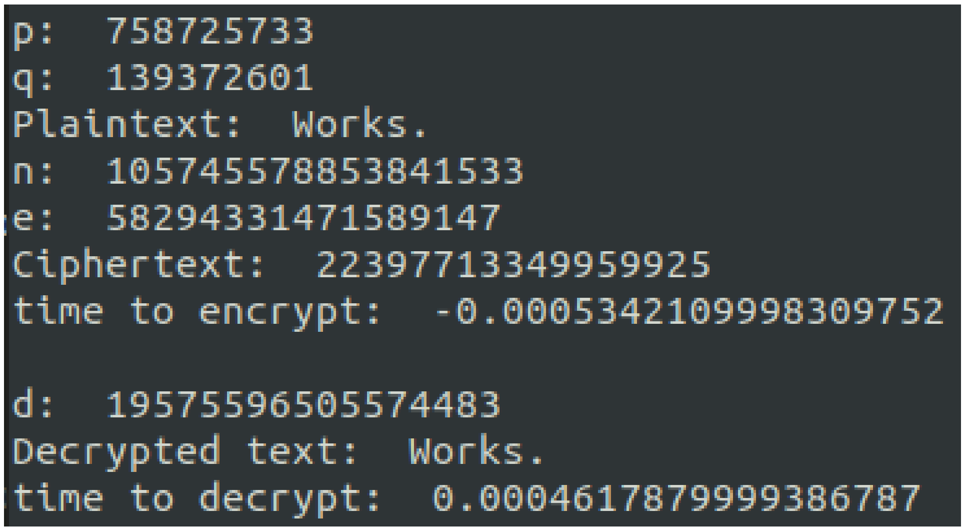
```
def bytes_to_int(bytes):
    result = 0
    for b in bytes:
        result = result * 256 + int(b)
    return result

def int_to_bytes(value, length):
    result = []
    for i in range(0, length):
        result.append(value >> (i * 8) & 0xff)
    result.reverse()
    return result

def randchoose():
    # I choose  p ad q randomly
    p = sympy.randprime(100000000,1000000000)
    q = sympy.randprime(100000000,1000000000)
    print("p: ",p)
    print("q: ",q)
    return p, q

def gcd(a,b):
    if(b==0):
        return a
    else:
        return gcd(b,a%b)

def pubKey():
  p, q = randchoose()
  n = int(p)*int(q)
  phi = (p-1)*(q-1)
  while(1):
    e = random.choice(range(1, phi-1))
    if(gcd(e,phi)) == 1:
      break
  return (n, e, phi)

def encrypt(x, n, e):
  y = pow(x,e,n)
  return y
```

Figure 2: Implementation functions



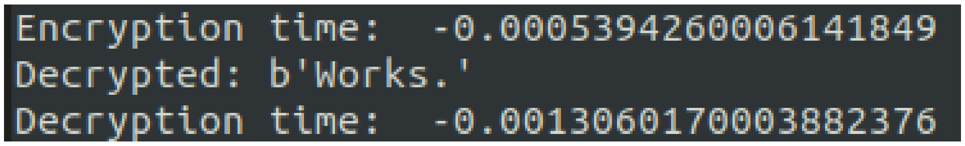
```
def main():
    fo = open('input.txt', 'r')
    x = fo.read()
    fo.close()

    start = timeit.timeit()  # I start to timer my algorithm

    n, e, phi = pubKey() # return the public key
    print("Plaintext: ",x)
    print("n: ",n)
    print("e: ",e)

    x = bytes_to_int(x.encode())
    y = encrypt(x, n, e)

    print("Ciphertext: ", y) # string encrypted.

    end = timeit.timeit()
    print("time to encrypt: ", end-start) #encrypting time

    start = timeit.timeit() # timer also for the decryption.

    d = sympy.mod_inverse(e, phi) # calculate the modulo inverse of e with modulo phi
    print(" ")
    print("d: ",d)

    decrypted = encrypt(y, n, d) # decrypting.

    plain = int_to_bytes(decrypted,len(str(decrypted)))
    x = ''
    x = ''.join(chr(x) for x in plain)

    print("Decrypted text: ",x)

    end = timeit.timeit()
    print("time to decrypt: ", end-start) #decrypting time

main()
```

Figure 3: Implementation main



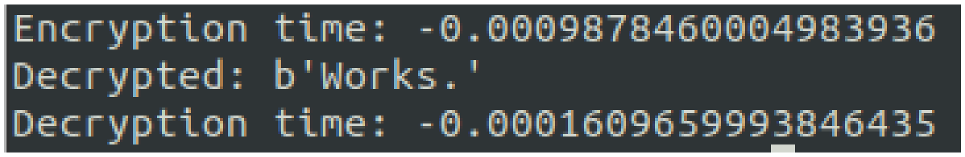
```
p:  758725733
q:  139372601
Plaintext:  Works.
n:  105745578853841533
e:  58294331471589147
Ciphertext:  22397713349959925
time to encrypt:  -0.0005342109998309752

d:  19575596505574483
Decrypted text:  Works.
time to decrypt:  0.0004617879999386787
```

Figure 4: My execution



```
Encryption time:  -0.0005394260006141849
Decrypted: b'Works.'
Decryption time:  -0.0013060170003882376
```

Figure 5: RSA execution



```
Encryption time: -0.0009878460004983936
Decrypted: b'Works.'
Decryption time: -0.0001609659993846435
```

Figure 6: AES execution