RSA and Homomorphic Multiplication

- https://arnaucube.com
- https://github.com/arnaucube
- https://twitter.com/arnaucube

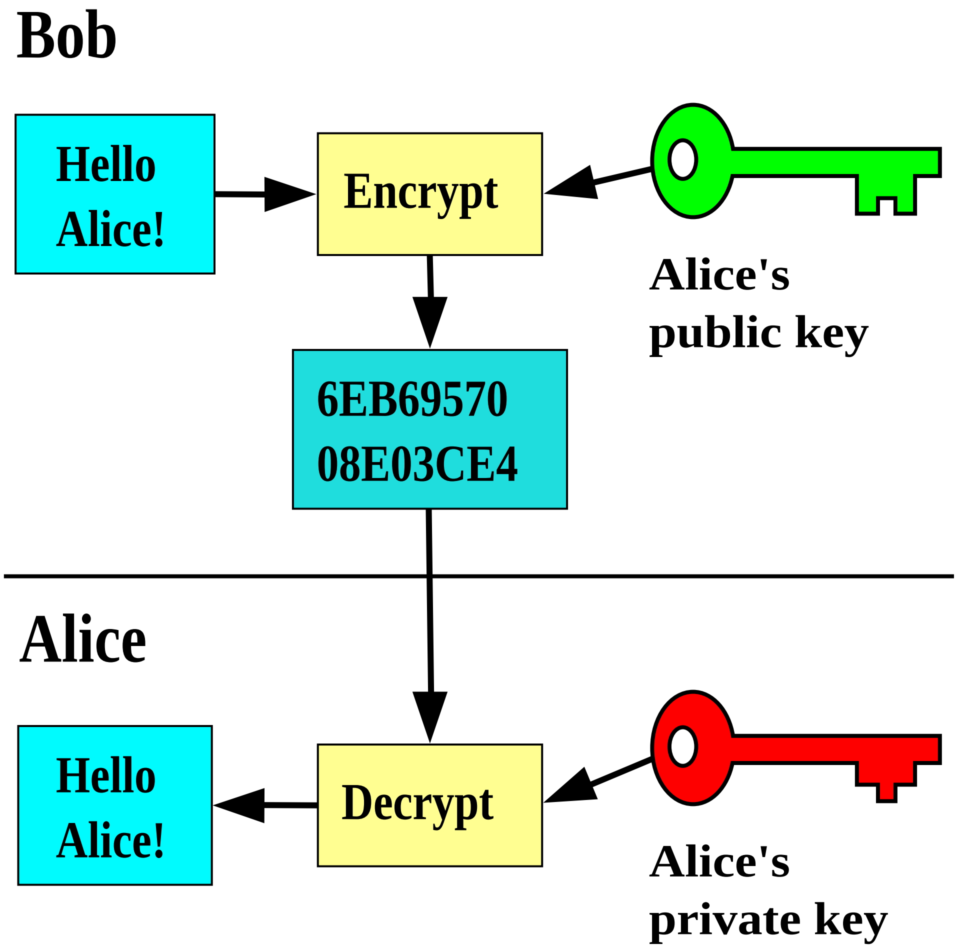
2018-11-30

- Intro
- Public key cryptography
- Basics of modular arithmetic
- Brief history of RSA
- Keys generation
- Prime numbers
- Encryption
- Decryption
- What's going on in encryption and decryption?
- Signature
- Verification of the signature
- Homomorphic Multiplication with RSA
- Resources

Intro

- I'm not an expert on the field, neither a mathematician. Just an engineer with interest for cryptography
- Short talk (15 min), with the objective to make a practical introduction to the RSA cryptosystem
- Is not a talk about mathematical demostrations, is a talk with the objective to get the basic notions to be able to do a practical implementation of the algorithm
- After the talk, we will do a practical workshop to implement the concepts. We can offer support for Go, Rust, Python and Nodejs (you can choose any other language, but we will not be able to help)

Public key cryptography

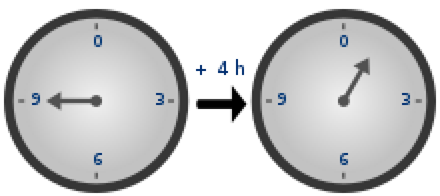


Some examples:

- RSA
- Paillier
- ECC (Corba el·líptica)

Basics of modular arithmetic

- Modulus, `mod`, `%`
- Remainder after division of two numbers




```
5 mod 12 = 5
14 mod 12 = 2
83 mod 10 = 3
```

```
5 + 3 mod 6 = 8 mod 6 = 2
```

Brief history of RSA

- RSA (Rivest–Shamir–Adleman): Ron Rivest, Adi Shamir, Leonard Adleman
- year 1977
- one of the first public key cryptosystems
- based on the difficulty of factorization of the product of two big prime numbers

Prime numbers

- We need an asymmetric key, in a way where we can decrypt a message encrypted with the asymetric key
- Without allowing to find the private key from the public key
- in RSA we resolve this with factorization of prime numbers
- using prime numbers for p and q, it's difficult factorize n to obtain p and q, where $n = p * q$

Example:

If we know n which we need to find the p and q values where $p * q = n$:

```
n = 35
```

To obtain the possible factors, is needed to brute force trying different combinations, until we find:

```
p = 5
q = 7
```

In this case is easy as it's a simple example with small numbers. The idea is to do this with big prime numbers

Another exmample with more bigger prime numbers:

```
n = 272604817800326282194810623604278579733
```

From n, I don't have a 'direct' way to obtain p and q. I need to try by brute force the different values until finding a correct combination.

```
p = 17975460804519255043
q = 15165386899666573831
n = 17975460804519255043 * 15165386899666573831 = 27260481
```

If we do this with non prime numbers:

```
n = 32
We can factorize 32 = 2 * 2 * 2 * 2 * 2
combining that values in two values X * Y
for example (2*2*2) * (2*2) = 8*4 = 32
we can also take 2 * (2*2*2*2) = 2 * 16 = 32
...
```

One example with bigger non prime numbers:

```
n = 272604817800326282227951471308464408608
We can take:
p = 17975460804519255044
q = 15165386899666573832

Or also:
p = 2
q = 136302408900163141113975735654232204304
...
```

In the real world:

- https://en.wikipedia.org/wiki/RSA_numbers
- https://en.wikipedia.org/wiki/RSA_Factoring_Challenge#The_prizes_and_records

So, we are basing this in the fact that is not easy to factorize big numbers composed by big primes.

Keys generation

- PubK: e, n
- PrivK: d, n
- are choosen randomly 2 big prime numbers p and q, that will be secrets
- $n = p * q$
- λ is the Carmichael function
  - λ(n) = (p − 1) * (q − 1)
- Choose a prime number e that satisfies 1 < e < λ(n) and gcd(e, λ(n))=1
  - Usually in examples is used $e = 2^16 + 1 = 65537$
- d such as e * d = 1 mod λ(n)
  - d = e^(-1) mod λ(n) = e modinv λ(n)

Example

- `p = 3`
- `q = 11`
- `e = 7` value choosen between 1 and λ(n)=20, where λ(n) is not divisible by this value
- `n = 3 * 11 = 33`
- `λ(n) = (3-1) * (11-1) = 2 * 10 = 20`
- `d` such as `7 * d = 1 mod 20`
- `d = 3`
- PubK: `e=7, n=33`
- PrivK: `d=3, n=33`

Naive code

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    g, y, x = egcd(b%a,a)
    return (g, x - (b//a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('No modular inverse')
    return x%m
```

```
def newKeys():
    p = number.getPrime(n_length)
    q = number.getPrime(n_length)

    # pubK e, n
    e = 65537
    n = p*q
    pubK = PubK(e, n)

    # privK d, n
    phi = (p-1) * (q-1)
    d = modinv(e, phi)
    privK = PrivK(d, n)

    return({'pubK': pubK, 'privK': privK})
```

Encryption

- Brenna wants to send the message `m` to Alice, so, will use the Public Key from Alice to encrypt `m`
- `m` powered at `e` of the public key from Alice
- evaluate at modulus of `n`

Example

- message to encrypt `m = 5`
- receiver public key: `e=7, n=33`
- `c = 5 ^ 7 mod 33 = 78125 mod 33 = 14`

Naive code

```
def encrypt(pubK, m):
    c = (m ** pubK.e) % pubK.n
    return c
```

Decrypt

- from an encrypted value `c`
- `c` powered at `d` of the private key of the person to who the message was encrypted
- evaluate at modulus of `n`

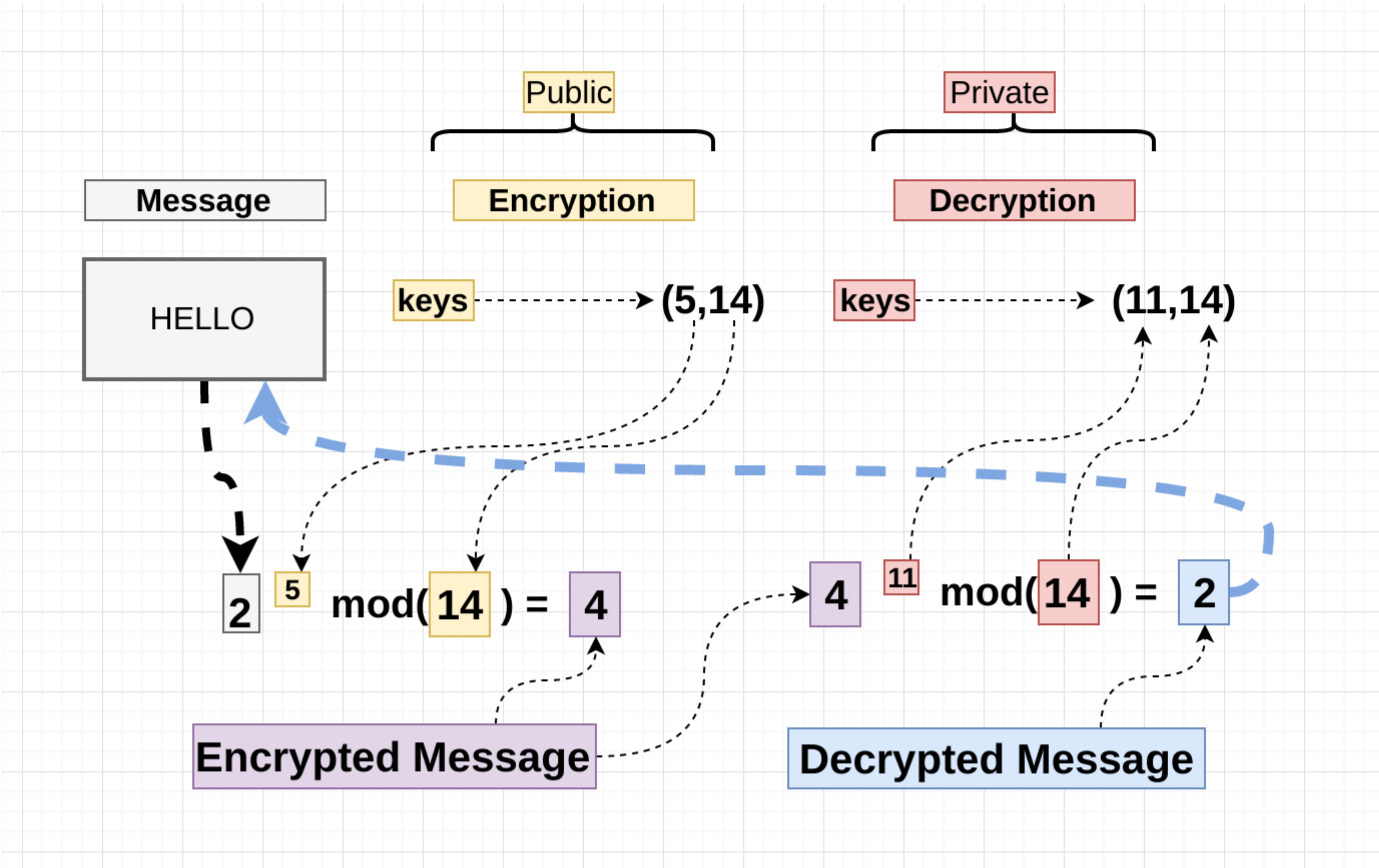
Example

- receiver private key, PrivK: `d=3, n=33`
- `m = 14 ^ 3 mod 33 = 2744 mod 33 = 5`

Naive code

```
def decrypt(privK, c):
    m_d = (c ** privK.d) % privK.n
    return m_d
```

What's going on when encrypting and decrypting?



```
n = pq
e
phi = (p-1)(q-1)
d = e^-1 mod (phi) = e^-1 mod (p-1)(q-1)

# encrypt
c = m^e mod n = m^e mod pq

# decrypt
m' = c^d mod n = c ^(e^-1 mod (p-1)(q-1)) mod pq =
        = (m^e)^(e^-1 mod (p-1)(q-1)) mod pq =
        = m^(e * e^-1 mod (p-1)(q-1)) mod pq =
        = m^(1 mod (p-1)(q-1)) mod pq =
        [theorem in which we're not going into details]
                a ^ (1 mod λ(N)) mod N = a mod N
        [/theorem]
        = m mod pq
```

Signature

- encryption operation but using PrivK instead of PubK, and PubK instead of PrivK
- having a message `m`
- power of `m` at `d` of the private key from the signer person
- evaluated at modulus `n`

Example

- private key of the person emitter of the signature: `d = 3, n = 33`
- message to be signed: `m=5`
- signature: `s = 5 ** 3 % 33 = 26`

Naive code

```
def sign(privK, m):
    s = (m ** privK.d) % privK.n
    return s
```

Verification of the signature

- having message `m` and the signature `s`
- elevate `m` at `e` of the public key from the signer
- evaluate at modulus of `n`

Example

- public key from the singer person `e=7, n=33`
- message `m=5`
- signature `s=26`
- verification `v = 26**7 % 33 = 5`
- check that we have recovered the message (that `m` is equivalent to `v`) `m = 5 = v = 5`

Naive code

```
def verifySign(pubK, s, m):
    v = (s ** pubK.e) % privK.n
    return v==m
```

Homomorphic Multiplication

- from two values a and b
- encrypted are a_{encr} and b_{encr}
- we can compute the multiplication of the two encrypted values, obtaining the result encrypted
- the encrypted result from the multiplication is calculated doing: $c_{encr} = a_{encr} * b_{encr} mod n$
- we can decrypt c_{encr} and we will obtain c, equivalent to $a * b$
- Why:

```
((a^e mod n) * (b^e mod n)) mod n =
= (a^e * b^e mod n) mod n = (a*b)^e mod n
```

Example

- PubK: `e=7, n=33`
- PrivK: `d=3, n=33`
- `a = 5`
- `b = 8`
- `a_encr = 5^7 mod 33 = 78125 mod 33 = 14`
- `b_encr = 8^7 mod 33 = 2097152 mod 33 = 2`
- `c_encr = (14 * 2) mod 33 = 28 mod 33 = 28`
- `c = 28 ^ 3 mod 33 = 21952 mod 33 = 7`
- `c = 7 = a * b % n = 5 * 8 % 33 = 7`, on `5*8 mod 33 = 7`
- take a `n` enough big, if not the result will be cropped by the modulus

Naive code

```
def homomorphic_mul(pubK, a, b):
    c = (a*b) % pubK.n
    return c
```

Small demo

[...]

And now... practical implementation

- full night long
- big ints are your friends