Criptosistema Rabin

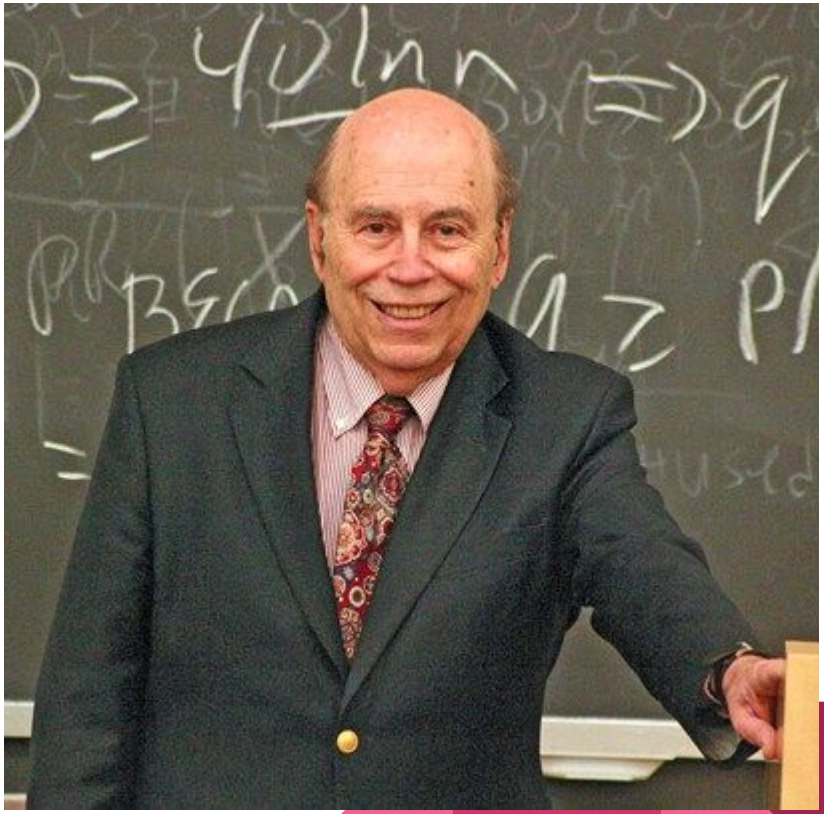
Canek García (kaan.ek@ciencias.unam.mx)

Agenda

- Breve historia
- Criptosistema Rabin
- Implementación

Breve historia...

- Técnica criptográfica asimétrica, cuya seguridad (al igual que RSA) se basa en la complejidad de la factorización.
- Se ha demostrado que la complejidad del problema en la que se basa es tan duro como la factorización de enteros.
- Cada salida de la función de Rabin puede ser generado por 4 posibles entradas.
- El algoritmo se publicó en enero de 1979 por Michael O. Rabin.



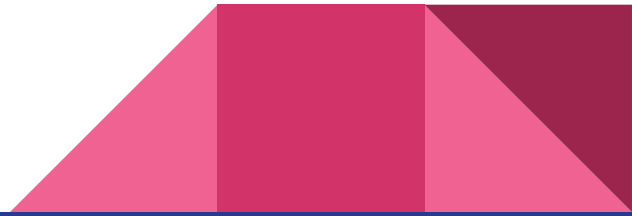
Criptosistema

Notas previas

En álgebra habitual, un número tiene 2 raíces reales, pero cuando n es el producto de 2 primos, existen valores que tienen 4 raíces distintas.

Tendrán 4 raíces todos los números que no tengan como divisor a uno de los dos primos con los que se generó n, en cuyo caso tendrán solo 2.

Si se realizan los cálculos se observará que existen del orden de O(p+q) valores con 2 raíces y del orden de O(p*q) valores con 4 raíces distintas.

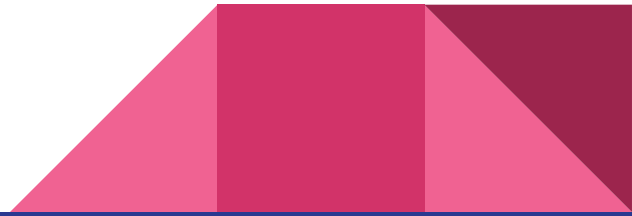


1. Generación de llaves

1. Cada usuario elige una pareja de primos **p** y **q**, cada uno igual a **3 mod 4**, ie, ambos **congruentes** con **3 mod 4**
2. Calcular el producto **n = pq**

   La **clave pública** será: **n**

   La **clave privada** será: **p** y **q**



2. Cifrar mensaje

Cabe aclarar que para cifrar el mensaje, este debe estar en forma de cadena de bits con una longitud de 16 bits, en el caso de que falten se cogerán las últimas cifras de la cadena para completar los 16 bits.

Se realiza una conversión de binario a decimal de la cadena ingresada **m**

Para codificar un mensaje, simplemente se calcula:

$$\mathbf{C = E(m) = m^2 \ (mod \ n)}$$

3. Descifrar el mensaje

Para desencriptar el mensaje solo se deben obtener las 4 raíces de:

$$C \ (\mathbf{mod}\ n)$$

Una de las raíces convertidas a binario hará referencia al mensaje cifrado.

Implementación

Generación de llaves

```java
public static BigInteger blumPrime(int bitLength) {
    BigInteger p;

    do {
        p = BigInteger.probablePrime(bitLength,r);
    }
    while( !p.mod(FOUR).equals(THREE) ); // congruent 3 (mod 4)

    return p;
}
```

```java
public static BigInteger[] genKey(int bitLength) {
    BigInteger p = blumPrime(bitLength/2);
    BigInteger q = blumPrime(bitLength/2);
    BigInteger n = p.multiply(q);

    return new BigInteger[]{n,p,q};
}
```

Cifrando

```
public static BigInteger encrypt(BigInteger m, BigInteger n) {
    // E(m)=m^2(mod n)
    return m.modPow(TWO, n);
}
```

Descifrando

```
public static BigInteger[] decrypt(BigInteger c, BigInteger p , BigInteger q) {
    BigInteger N = p.multiply(q);

    BigInteger m_p1 = c.modPow(p.add(BigInteger.ONE).divide(FOUR), p);
    BigInteger m_p2 = p.subtract(m_p1);
    BigInteger m_q1 = c.modPow(q.add(BigInteger.ONE).divide(FOUR), q);
    BigInteger m_q2 = q.subtract(m_q1);

    BigInteger[] ext = ext_gcd(p,q); // Greatest common divisor

    BigInteger y_p = ext[1];
    BigInteger y_q = ext[2];

    //y_p*p*m_q + y_q*q*m_p (mod n)
    BigInteger d1 = y_p.multiply(p).multiply(m_q1).add(y_q.multiply(q).multiply(m_p1)).mod(N);
    BigInteger d2 = y_p.multiply(p).multiply(m_q2).add(y_q.multiply(q).multiply(m_p1)).mod(N);
    BigInteger d3 = y_p.multiply(p).multiply(m_q1).add(y_q.multiply(q).multiply(m_p2)).mod(N);
    BigInteger d4 = y_p.multiply(p).multiply(m_q2).add(y_q.multiply(q).multiply(m_p2)).mod(N);

    return new BigInteger[]{d1,d2,d3,d4};
}
```