

UNIVERSIDAD NACIONAL SAN ANTONIO ABAD DEL CUSCO

Facultad de Ingeniería Eléctrica, Electrónica, Informática y Mecánica

Escuela profesional de Ingeniería Informática y de Sistemas

DESCIFRAR RSA CON COMPUTACIÓN CUÁNTICA

Integrantes

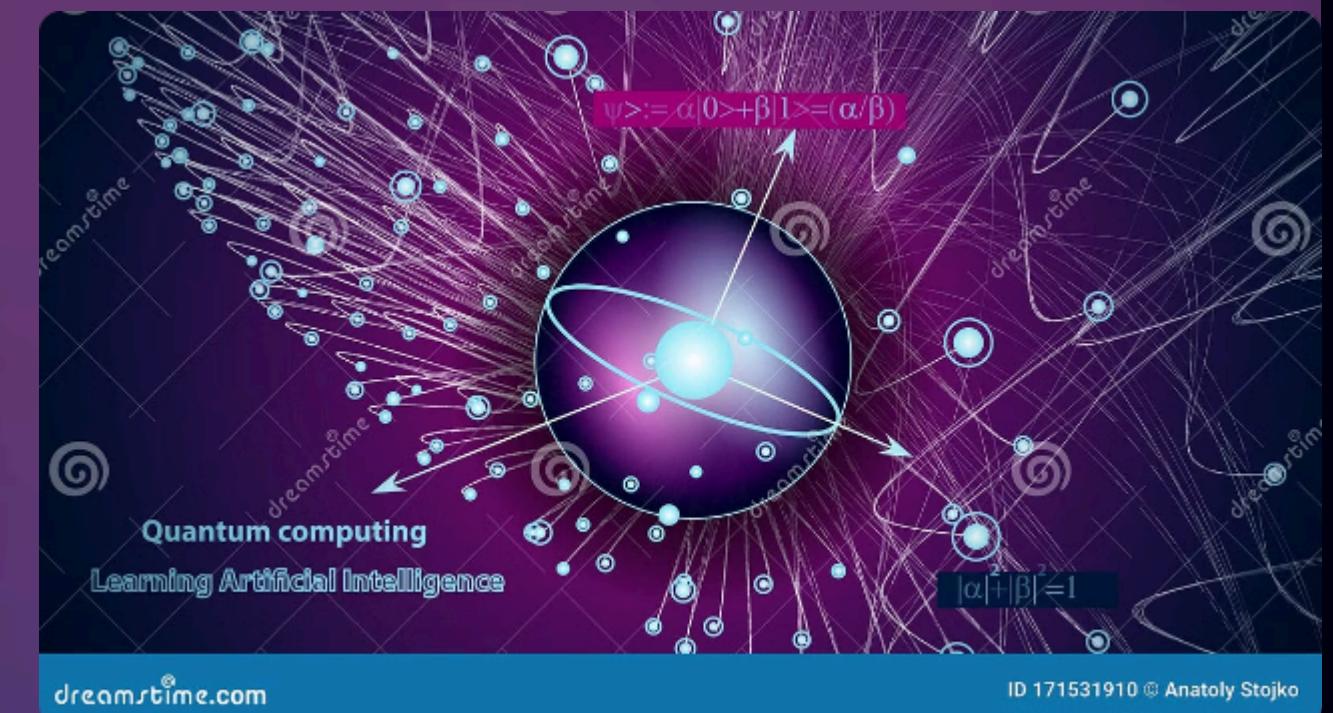
- Llasa Yucra, Ruth Margot
- Montes Huillca, Franklin Jesus
- Cusi Fuentes, Gonzalo
- Mendoza Huayllapyuma, Elisban
- Condori Huaychay, Cesar Aparicio



INTRODUCCIÓN

La criptografía ha sido crucial para proteger la comunicación e información a lo largo de la historia, adaptándose a los avances tecnológicos y nuevas amenazas. En este contexto, la criptografía cuántica emerge como una disciplina prometedora que podría revolucionar la seguridad de la información.

En este proyecto exploraremos el impacto de la computación cuántica en la criptografía, con un enfoque en el cifrado RSA, uno de los algoritmos criptográficos más utilizados hoy en día.





HISTORIA Y DESARROLLO

Criptografía Clásica

La criptografía nació como el arte de esconder información. A través de los siglos, se desarrollaron técnicas y métodos para proteger la comunicación.

Objetivo Común

Tanto la criptografía clásica como la moderna buscan cifrar la información para que solo el receptor autorizado pueda descifrarla.



Criptografía Moderna

En 1948, la Teoría de la Información de Claude Shannon convirtió a la criptografía en una ciencia basada en las matemáticas. Esto dio paso a algoritmos criptográficos como DES, RSA y AES.

COMPUTACIÓN CUÁNTICA

La computación cuántica utiliza principios de la mecánica cuántica, como la superposición y el entrelazamiento, para realizar cálculos mucho más rápidos y eficientes que las computadoras clásicas. Esta tecnología tiene el potencial de revolucionar campos como la criptografía, la inteligencia artificial y la simulación de sistemas físicos complejos, permitiendo resolver problemas actualmente inabordables.

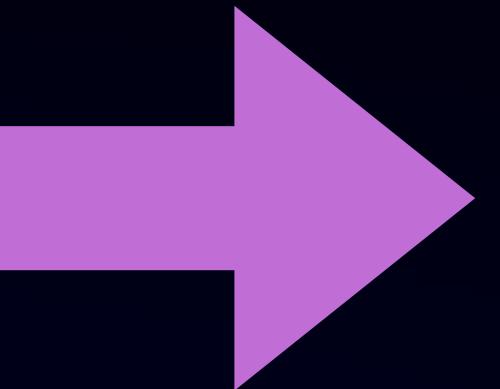


Principios de la mecánica cuántica

La mecánica cuántica estudia el comportamiento de las partículas a escalas subatómicas, revelando fenómenos como la superposición, donde partículas pueden existir en múltiples estados simultáneamente, y el entrelazamiento, donde partículas separadas están instantáneamente conectadas, afectando el estado de una a la otra sin importar la distancia. Estos principios, confirmados por experimentos, desafían la intuición de la física clásica.

Qubits, Superposición y Entrelazamiento

En la computación cuántica, el qubit puede representar simultáneamente un 0 y un 1 gracias a la superposición y puede entrelazarse con otros qubits, permitiendo operaciones complejas. Esto da a las computadoras cuánticas el potencial de resolver problemas difíciles para las computadoras clásicas, como la factorización de números grandes y la simulación de moléculas.

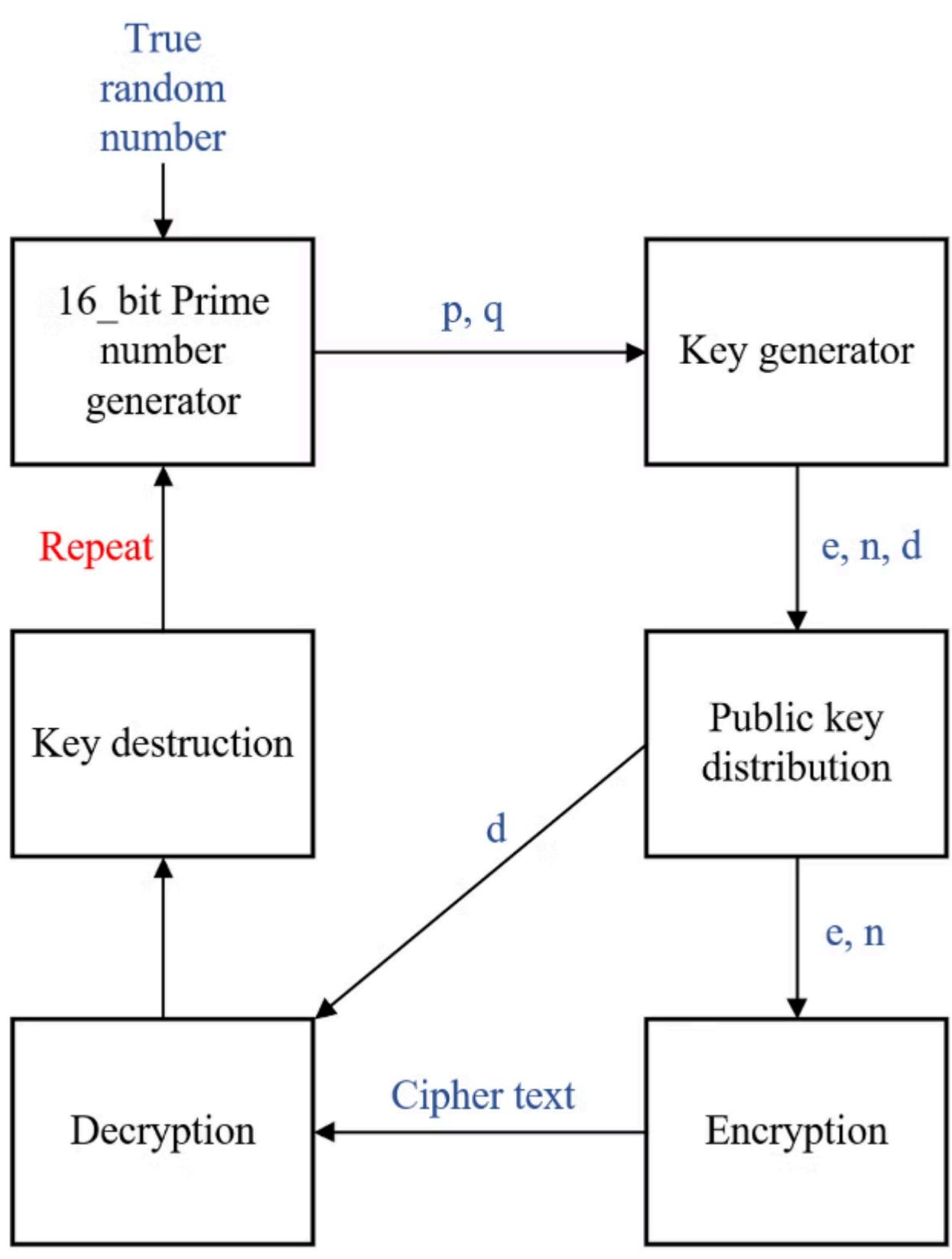


RSA Y LA COMPUTACIÓN CUÁNTICA

RSA, un sistema de criptografía de clave pública, se basa en la dificultad de factorizar números grandes. La computación cuántica, con el algoritmo de Shor, podría romper esta seguridad al factorizar números grandes eficientemente.

Aunque las computadoras cuánticas necesarias aún no existen, su desarrollo rápido sugiere que pronto serán una realidad, poniendo en riesgo la seguridad basada en RSA.

The RSA logo, consisting of the letters "RSA" in a bold, white, sans-serif font, with a registered trademark symbol (®) at the top right corner.



Algoritmo RSA

Cifrado Asimétrico

RSA se basa en el uso de dos números primos grandes para generar una llave pública y una privada.

Seguridad en Factorización

La seguridad de RSA radica en la dificultad de factorizar el producto de los dos números primos.

Amplio Uso

RSA es utilizado en protocolos como SSL, VPN e infraestructura de llave pública (PKI).



FUNCIONAMIENTO DEL ALGORITMO RSA

Generación de Claves

Cifrado del Mensaje

Descifrado del Mensaje

$$M = C^j \bmod n$$

Generación de claves	
<ol style="list-style-type: none">1. Seleccionar dos números primos: p, q2. Calcular: $n = p * q$3. Calcular: $z = (p - 1) * (q - 1)$4. Seleccionar un entero k que cumpla: $\gcd(z, k) = 1; 1 < k < z$ <small>gcd: greatest common divisor (máximo común divisor)</small>5. Elegir j de modo que cumpla: $k * j = 1 \pmod{z}$ En la práctica: elegir un j entero que verifique $j = (1+x*z)/k$ para algún valor entero de k	<p>Clave Pública: (n , k)</p> <p>Clave Privada: (j)</p>

Cifrado y descifrado	
<p>Texto cifrado: C, que verifica: $M^k = C \pmod{n}$</p> <p>Que puede calcularse así: $C = M^k \% n$ (donde '%' calcula el módulo)</p>	<p>Texto plano: M, que verifica: $C^j = M \pmod{n}$</p> <p>Que puede calcularse así: $M = C^j \% n$ (donde '%' calcula el módulo)</p>

APLICACIONES DEL ALGORITMO RSA

Aplicaciones en el entorno

Firmas Digitales

Autenticación y
Verificación de Identidad

Distribución de Claves

Sistemas de Mensajería
Segura

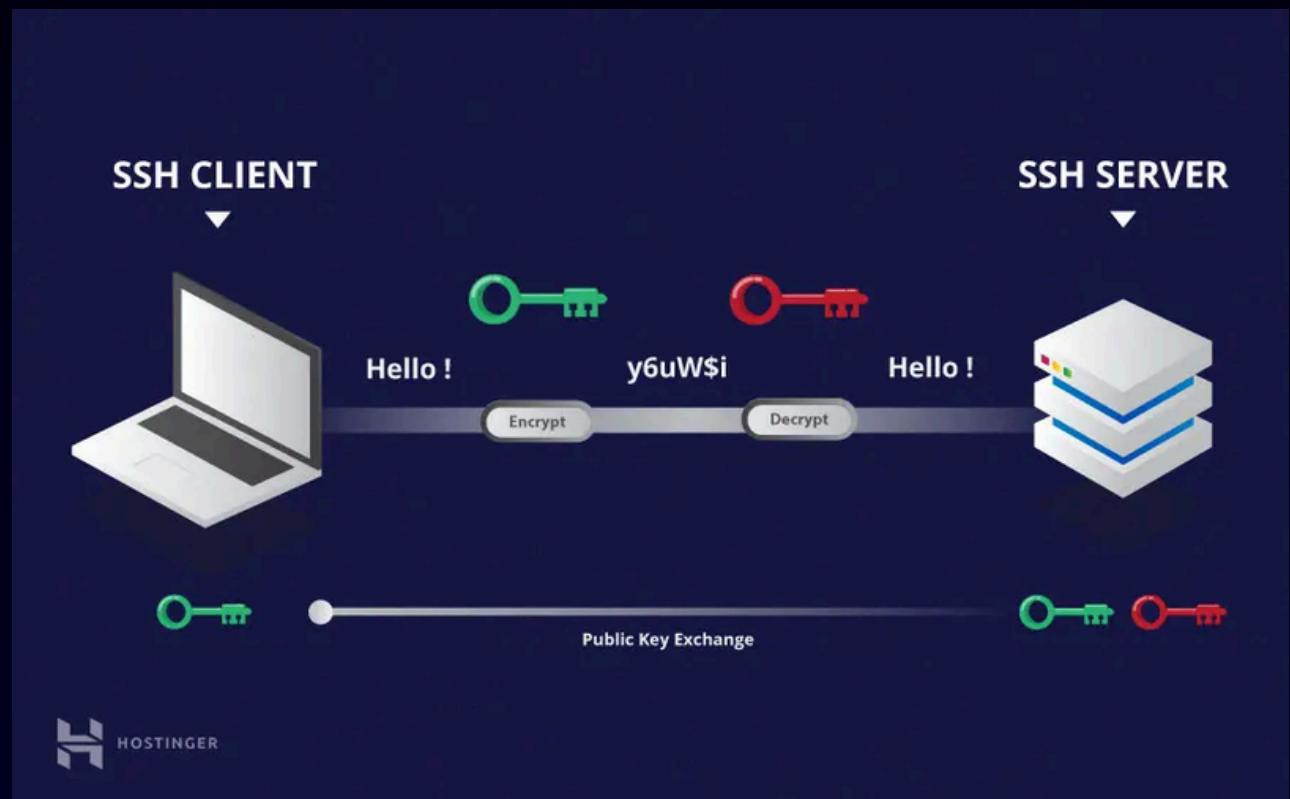
Cifrado de Datos en
Almacenamiento

Aplicaciones en seguridad más
importantes que usan RSA

SSH (Secure Shell)

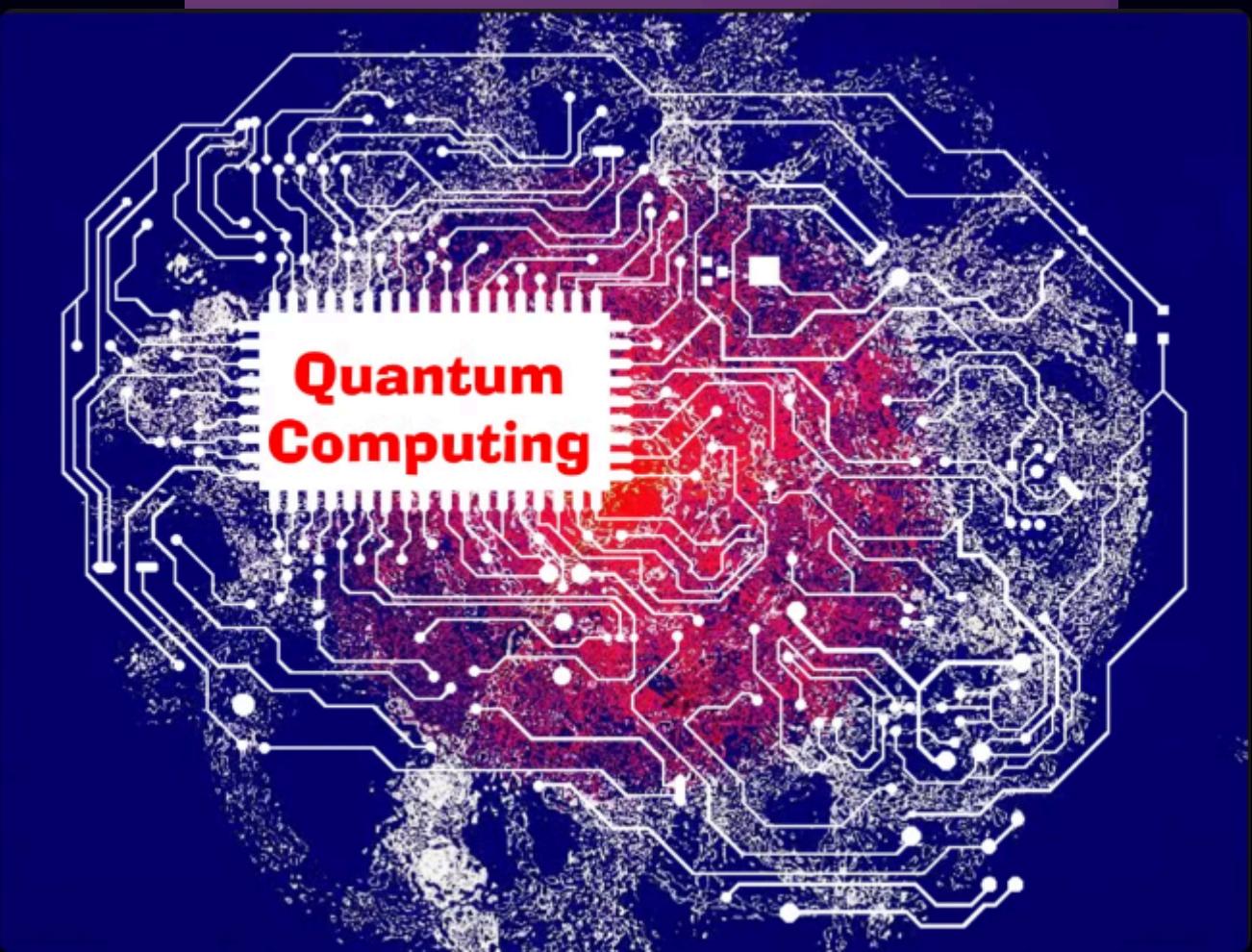
SSL (Secure Socket
Layer)

PGP (Pretty Good
Privacy).



ALGORITMO DE SHOR

El Algoritmo de Shor, diseñado para computadoras cuánticas, puede factorizar eficientemente números grandes, amenazando la seguridad de RSA. Aunque la implementación del algoritmo enfrenta obstáculos debido a la falta de qubits y la necesidad de ejecución coherente, su capacidad para factorizar números de 1024 bits en segundos con una computadora cuántica de 100 MIPS destaca la urgencia de desarrollar técnicas criptográficas postcuánticas para proteger la información en el futuro.

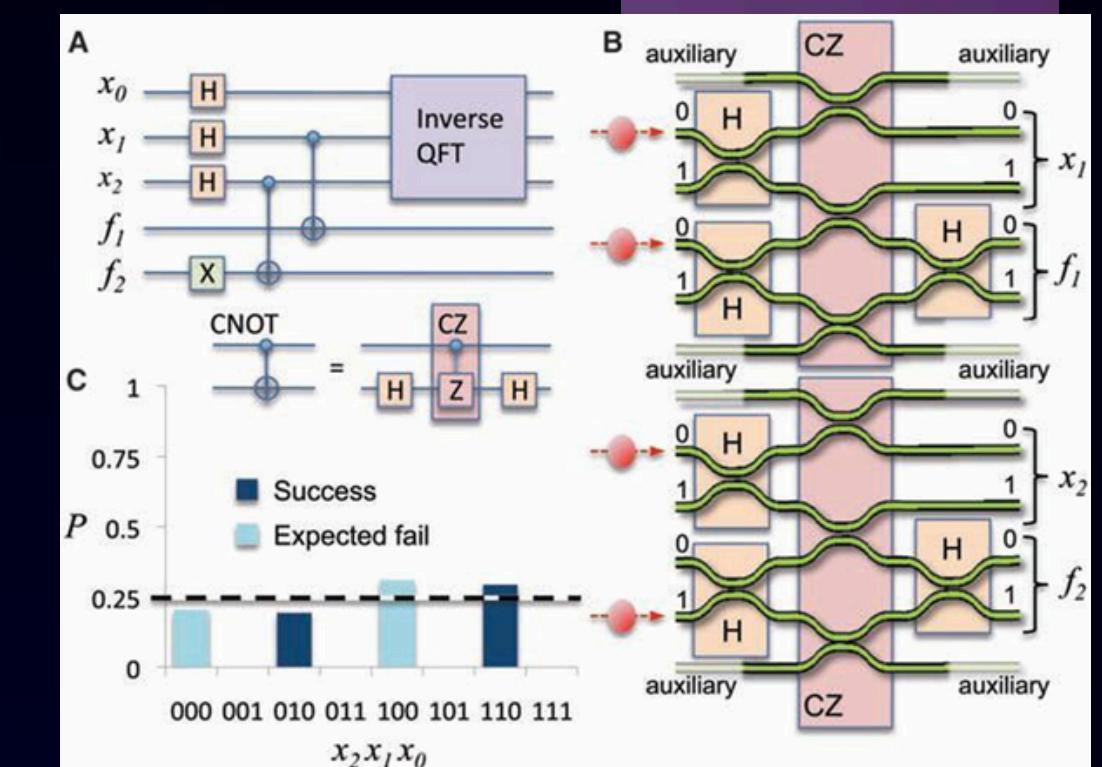
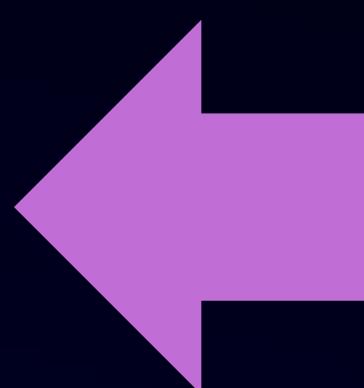


Aplicaciones del Algoritmo de Shor con un ordenador cuántico

• El algoritmo de Shor puede realizar la factorización de números primos muy grandes en un ordenador cuántico.

• El algoritmo de Shor puede resolver el problema de encontrar el periodo de una función

• El algoritmo de Shor puede utilizarse potencialmente para hackear RSA y otras formas de datos seguros



Algoritmo de Shor



- 1
- 2
- 3

Elegir Número Aleatorio

Se elige un número aleatorio a tal que $1 < a < N$.

Encontrar Período

Se utiliza la transformada de Fourier cuántica para encontrar el período r de la función $f(x) = a^x \text{ mod } N$.

Calcular Factores

Con el período r, se calculan los factores de N mediante el máximo común divisor.

DESCIFRAR RSA CON COMPUTACIÓN CUÁNTICA

La computación cuántica, especialmente a través del Algoritmo de Shor, desarrollado por Peter Shor en 1994, promete revolucionar la criptografía. Este algoritmo puede factorizar números enteros grandes en tiempo polinómico, lo que representa un desafío significativo para las computadoras clásicas y amenaza la seguridad del cifrado RSA.



El Algoritmo de Shor y la Factorización de Números Grandes

El Algoritmo de Shor aprovecha dos principios fundamentales de la mecánica cuántica: la superposición y el entrelazamiento. Mediante estos principios, puede realizar cálculos en paralelo a una velocidad exponencialmente superior a la de los algoritmos clásicos.



Implicaciones para el Cifrado RSA

El cifrado RSA se basa en la dificultad de factorizar grandes números, pero el Algoritmo de Shor en computadoras cuánticas podría comprometer su seguridad, lo que impulsa la necesidad de criptografía postcuántica.



Estado Actual y Futuro de la Computación Cuántica

Aunque el Algoritmo de Shor ha sido probado en sistemas cuánticos experimentales, la implementación práctica en números lo suficientemente grandes para romper RSA aún no es posible debido a las limitaciones tecnológicas actuales, como el control de errores cuánticos y la estabilidad de los qubits.



IMPLEMENTACIÓN

Obteniendo nuestras dependencias

```
!pip install qiskit==0.46.0
```

```
collecting qiskit==0.46.0
```

```
  Downloading qiskit-0.46.0-py3-none-any.whl.metadata (12.4 MB)
Collecting qiskit-terra==0.46.0 (from qiskit==0.46.0)
  Downloading qiskit_terra-0.46.0-cp38-abi3-manylinux2014_x86_64.whl.metadata (12.4 MB)
Collecting rustworkx>=0.13.0 (from qiskit-terra==0.46.0)
  Downloading rustworkx-0.15.1-cp38-abi3-manylinux2014_x86_64.whl.metadata (12.4 MB)
Requirement already satisfied: numpy<2,>=1.17 in /usr/local/lib/python3.10/dist-packages (from qiskit-terra==0.46.0)
Collecting ply>=3.10 (from qiskit-terra==0.46.0)
  Downloading ply-3.11-py2.py3-none-any.whl.metadata (12.4 MB)
Requirement already satisfied: psutil>=5 in /usr/local/lib/python3.10/dist-packages (from qiskit-terra==0.46.0)
Requirement already satisfied: scipy>=1.5 in /usr/local/lib/python3.10/dist-packages (from qiskit-terra==0.46.0)
Requirement already satisfied: sympy>=1.3 in /usr/local/lib/python3.10/dist-packages (from qiskit-terra==0.46.0)
Collecting dill>=0.3 (from qiskit-terra==0.46.0)
  Downloading dill-0.3.8-py3-none-any.whl.metadata (12.4 MB)
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-terra==0.46.0)
Collecting stevedore>=3.0.0 (from qiskit-terra==0.46.0)
  Downloading stevedore-5.2.0-py3-none-any.whl.metadata (12.4 MB)
Collecting symengine>=0.11 (from qiskit-terra==0.46.0)
  Downloading symengine-0.11.0-cp310-cp310-manylinux2014_x86_64.whl (12.4 MB)
```

```
pip install qiskit-aer
```

```
Collecting qiskit-aer
```

```
  Downloading qiskit_aer-0.14.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (12.4 MB)
Requirement already satisfied: qiskit>=0.45.2 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: numpy>=1.16.3 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: psutil>=5 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: qiskit-terra==0.46.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: rustworkx>=0.13.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: ply>=3.10 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: sympy>=1.3 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: dill>=0.3 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: stevedore>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: symengine>=0.11 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from pygments)
Requirement already satisfied: pbr!=2.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer)
  Downloading qiskit_aer-0.14.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.4 MB) 12.4/12.4 MB 45.5 MB/s eta 0:00:00
```

Installing collected packages: qiskit-aer
Successfully installed qiskit-aer-0.14.2

Instalación de las librerías

```
s ➜ pip install qiskit-aer-gpu
→ Collecting qiskit-aer-gpu
  Downloading qiskit_aer_gpu-0.14.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.
Requirement already satisfied: qiskit>=0.45.2 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer-gpu)
Requirement already satisfied: numpy>=1.16.3 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer-gpu)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer-gpu)
Requirement already satisfied: psutil>=5 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer-gpu)
Collecting nvidia-cuda-runtime-cu12>=12.1.105 (from qiskit-aer-gpu)
  Downloading nvidia_cuda_runtime_cu12-12.5.82-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-nvjitlink-cu12 (from qiskit-aer-gpu)
  Downloading nvidia_nvjitlink_cu12-12.5.82-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cublas-cu12>=12.1.3.1 (from qiskit-aer-gpu)
  Downloading nvidia_cublas_cu12-12.5.3.2-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12>=11.4.5.107 (from qiskit-aer-gpu)
```

```
7s ➜ pip install qiskit-aer-gpu-cu11
→ Collecting qiskit-aer-gpu-cu11
  Downloading qiskit_aer_gpu_cu11-0.14.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.
Requirement already satisfied: qiskit>=0.45.2 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer-gpu-cu11)
Requirement already satisfied: numpy>=1.16.3 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer-gpu-cu11)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer-gpu-cu11)
Requirement already satisfied: psutil>=5 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer-gpu-cu11)
Collecting nvidia-cuda-runtime-cu11>=11.8.89 (from qiskit-aer-gpu-cu11)
  Downloading nvidia_cuda_runtime_cu11-11.8.89-py3-none-manylinux1_x86_64.whl.metadata (1.5 kB)
```

Importación de herramientas

```
[5] import numpy as np
from qiskit import *
from math import sqrt,log,gcd
import random
from random import randint
import rsa
```

Implementación de módulos principales

▼ Calcular la inversa modular

```
[6] def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return -1
```

▼ Comprobando la primalidad

```
▶ def isprime(n):
    if n < 2:
        return False
    elif n == 2:
        return True
    else:
        for i in range(1, int(sqrt(n)) + 1):
            if n % i == 0:
                return False
    return True
```

Implementación de módulos principales

+ Código + Texto Copiar en Drive

▼ Generando pares clave-valor

```
0s  def generate_keypair(keysize):
    p = randint(1, 1000)
    q = randint(1, 1000)
    nMin = 1 << (keysize - 1)
    nMax = (1 << keysize) - 1
    primes = [2]
    start = 1 << (keysize // 2 - 1)
    stop = 1 << (keysize // 2 + 1)
    if start >= stop:
        return []
    for i in range(3, stop + 1, 2):
        for p in primes:
            if i % p == 0:
                break
            else:
                primes.append(i)
    while (primes and primes[0] < start):
        del primes[0]
    # Seleccione dos números primos aleatorios P y Q
    while primes:
        p = random.choice(primes)
        primes.remove(p)
        q_values = [q for q in primes if nMin <= p * q <= nMax]
        if q_values:
            q = random.choice(q_values)
            break
    # Calcule n
    n = p * q
    # Calcule phi
    phi = (p - 1) * (q - 1)
    # Seleccione e
    e = random.randrange(1, phi)
    g = gcd(e, phi)
    # Calcule d
    while True:
        e = random.randrange(1, phi)
        g = gcd(e, phi)
        d = mod_inverse(e, phi)
        if g == 1 and e != d:
            break
    return ((e, n), (d, n))
```

Cifrado y descifrado

Paso de cifrado

```
[9] def encrypt(plaintext, package):  
    e, n = package  
    ciphertext = [pow(ord(c), e, n) for c in pl  
    return ''.join(map(lambda x: str(x), cipher
```

▼ Ahora probemos con un mensaje de muestra.

▼ Paso de descifrado

▼ Generar claves

```
[10] def decrypt(ciphertext, package):  
    d, n = package  
    plaintext = [chr(pow(c, d, n)) for c in cipher  
    return (''.join(plaintext))
```

→ Collecting pycryptodome

```
  Downloading pycryptodome-3.20.0-cp35-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.4 kB)  
  Downloading pycryptodome-3.20.0-cp35-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.1 MB)  
           2.1/2.1 MB 29.4 MB/s eta 0:00:00  
Installing collected packages: pycryptodome  
Successfully installed pycryptodome-3.20.0
```

```
[12] from Crypto.PublicKey import RSA
```

```
    bit_length = int(input("Ingrese la longitud del bit: "))  
  
    public_k, private_k = generate_keypair(2**bit_length)
```

→ Ingrese la longitud del bit: 3

Algoritmo Shor

```
[17] from qiskit import QuantumCircuit, Aer
    from qiskit_aer import AerSimulator

    import numpy as np

    from random import randint

    from qiskit_aer import AerSimulator
    qasm_sim = Aer.get_backend('qasm_simulator')
    def period(a,N):

        available_qubits = 16
        r=-1

        if N >= 2**available_qubits:
            print(str(N) +' es demasiado grande para IBMQX')

        qr = QuantumRegister(available_qubits)
        cr = ClassicalRegister(available_qubits)
        qc = QuantumCircuit(qr,cr)
        x0 = randint(1, N-1)
        x_binary = np.zeros(available_qubits, dtype=bool)

        for i in range(1, available_qubits + 1):
            bit_state = (N%(2**i)!=0)
            if bit_state:
                N -= 2**(i-1)
            x_binary[available_qubits-i] = bit_state

        for i in range(0,available_qubits):
            if x_binary[available_qubits-i-1]:
                qc.x(qr[i])
        x = x0
```

```
+ Código + Texto Copiar en Drive
if bit_state:
    N -= 2**(i-1)
x_binary[available_qubits-i] = bit_state

for i in range(0,available_qubits):
    if x_binary[available_qubits-i-1]:
        qc.x(qr[i])
x = x0

while np.logical_or(x != x0, r <= 0):
    r+=1
    qc.measure(qr, cr)
    for i in range(0,3):
        qc.x(qr[i])
        qc.cx(qr[2],qr[1])
        qc.cx(qr[1],qr[2])
        qc.cx(qr[2],qr[1])
        qc.cx(qr[1],qr[0])
        qc.cx(qr[0],qr[1])
        qc.cx(qr[1],qr[0])
        qc.cx(qr[3],qr[0])
        qc.cx(qr[0],qr[1])
        qc.cx(qr[1],qr[0])

    result = execute(qc,backend = qasm_sim, shots=1024).result()
    counts = result.get_counts()

    results = [[],[]]
    for key,value in counts.items():
        results[0].append(key)
        results[1].append(int(value))
    s = results[0][np.argmax(np.array(results[1]))]
    return r

→ <ipython-input-18-be62a101e701>:6: DeprecationWarning: The 'qiskit.Aer' entry point is deprecated and will be removed in Qiskit 1.0. Y
    qasm_sim = Aer.get_backend('qasm_simulator')
```

Algoritmo Shor

```
[19] def shors_breaker(N):
    N = int(N)
    while True:
        a=randint(0,N-1)
        g=gcd(a,N)
        if g!=1 or N==1:
            return g,N//g
        else:
            r=period(a,N)
            if r % 2 != 0:
                continue
            elif pow(a,r//2,N)==-1:
                continue
            else:
                p=gcd(pow(a,r//2)+1,N)
                q=gcd(pow(a,r//2)-1,N)
                if p==N or q==N:
                    continue
                return p,q
```

```
def modular_inverse(a,m):
    a = a % m;
    for x in range(1, m) :
        if ((a * x) % m == 1) :
            return x
    return 1
```

```
N_shor = public_k[1]
assert N_shor>0,"La entrada debe ser positiva"
p,q = shors_breaker(N_shor)
phi = (p-1) * (q-1)
d_shor = modular_inverse(public_k[0], phi)
```

+ Código + Texto

DESCIFRAMOS

- ▼ Descifremos nuestro texto cifrado usando el algoritmo de Shor

```
✓  print('Mensaje se agrietó usando el algoritmo de Shor: {}'.format(decrypt(cipher_obj, (d_shor,N_shor))))  
⇒ Mensaje se agrietó usando el algoritmo de Shor: ejecutar
```

Conclusión

La evolución de la criptografía y la computación ha sido impulsada por la adaptación a nuevos desafíos tecnológicos. La computación cuántica, especialmente a través del Algoritmo de Shor, podría revolucionar la seguridad de la información al factorizar números grandes en tiempo polinómico, amenazando sistemas como el cifrado RSA. Aunque las computadoras cuánticas adecuadas aún no están disponibles, su desarrollo inminente resalta la necesidad urgente de adoptar criptografía postcuántica para proteger la información en un futuro cuántico.