Advantages of Quantum Computing in Factor Attacks on RSA Encryption

Srinivas Rao Tammireddy
Roll No. 217Y1A05C0
Computer Science and Engineering
Guide: Dr. S Pratap Singh (Professor)

March 20, 2025

Contents

List of Figures

List of Tables

1 Abstract

The aim of this technical seminar is to explore the advantages of quantum computing in breaking RSA encryption. This report discusses the fundamentals of RSA encryption, the limitations of classical computing in breaking RSA, and how quantum computing, specifically Shor's algorithm, can efficiently factor large integers, posing a significant threat to RSA encryption. The report also explores post-quantum cryptographic methods and the future of cybersecurity in the quantum era. This seminar highlights the paradigm shift that quantum computing brings to cryptography and emphasizes the need for quantum-resistant security solutions.

2 Introduction

2.1 Background and Motivation

RSA encryption is a widely used public-key cryptosystem that relies on the difficulty of factoring large integers. The security of RSA is based on the assumption that factoring the product of two large prime numbers is computationally infeasible. However, with the advent of quantum computing, this assumption is being challenged.

2.2 Problem Statement

Classical computers face fundamental limitations when attempting to factor large numbers used in RSA encryption. This report explores how quantum computing overcomes these limitations through Shor's algorithm and examines the implications for current cryptographic systems.

2.3 Objectives of the Study

- To analyze the mathematical foundations of RSA encryption
- To investigate the limitations of classical computing in breaking RSA
- To examine how quantum computing, particularly Shor's algorithm, accelerates factorization
- To explore quantum-resistant cryptographic solutions
- To assess the timeline and practical implications of quantum threats to RSA

2.4 Scope and Limitations

This seminar focuses on the theoretical advantages of quantum computing for factorization and its implications for RSA security. While implementation

aspects are discussed, detailed quantum circuit implementations and physical realization challenges are beyond the scope of this report.

2.5 Historical Development of RSA

RSA was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT. It was one of the first practical public-key cryptosystems and is widely used for secure data transmission.

2.6 Classical Approaches to Breaking RSA

Researchers have developed various classical algorithms to factor large integers, including the quadratic sieve and the general number field sieve. Despite continuous improvements, these algorithms still require exponential time for large integers, making RSA secure against classical attacks for sufficiently large keys.

2.7 Evolution of Quantum Computing

Since Peter Shor's breakthrough algorithm in 1994, quantum computing research has grown significantly. Theoretical work has been complemented by experimental progress in building quantum computers with increasing numbers of qubits and reducing error rates.

2.8 Current State of Quantum Computing Technology

As of [current year], quantum computers have reached [X] qubits, though with significant error rates. Companies like IBM, Google, and Microsoft are investing heavily in quantum hardware development, with roadmaps projecting fault-tolerant quantum computers within the next decade.

3 Background: Classical Cryptography and RSA

3.1 Key Generation Process

1. Choose two distinct large prime numbers p and q.
2. Compute $n = pq$. The number n is used as the modulus for both the public and private keys.
3. Compute the totient function $\phi(n) = (p-1)(q-1)$.
4. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$. The integer e is the public exponent.
5. Compute the private exponent d such that $ed \equiv 1 \pmod{\phi(n)}$.

The public key is (e, n) and the private key is (d, n).

3.2 Encryption and Decryption Processes

3.2.1 RSA Algorithm

The RSA algorithm can be summarized in the following steps:

Algorithm 1 RSA Algorithm

Input: Message M, Public key (e, n), Private key (d, n)
Output: Ciphertext C, Decrypted message M'
Key Generation:

1. Choose two distinct large prime numbers p and q.
2. Compute $n = pq$.
3. Compute the totient function $\phi(n) = (p-1)(q-1)$.
4. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$.
5. Compute the private exponent d such that $ed \equiv 1 \pmod{\phi(n)}$.

Encryption:

1. Given a message M, compute the ciphertext C as:

$$C = M^e \mod n$$

Decryption:

1. Given a ciphertext C, compute the decrypted message M' as:

$$M' = C^d \mod n$$

3.2.2 Numerical Example

Let's consider an example with $p = 17$, $q = 19$, and $M = 12$.

1. Compute $n = pq = 17 \times 19 = 323$.
2. Compute the totient function $\phi(n) = (p-1)(q-1) = 16 \times 18 = 288$.
3. Choose $e = 5$ (since $1 < 5 < 288$ and $\gcd(5, 288) = 1$).
4. Compute the private exponent d such that $ed \equiv 1 \pmod{288}$. Here, $d = 173$ (since $5 \times 173 \equiv 1 \pmod{288}$).

The public key is $(5, 323)$ and the private key is $(173, 323)$.

Encryption To encrypt a message $M = 12$:

$$C = M^e \mod n = 12^5 \mod 323 = 248832 \mod 323 = 269$$

where C is the ciphertext.

Decryption To decrypt the ciphertext $C = 269$:

$$M = C^d \mod n = 269^{173} \mod 323 = 12$$

3.3 Implementation in Python

The following Python code demonstrates the RSA key generation, encryption, and decryption process:

```
# Step 1: Key Generation
import random
from sympy import gcd, isprime, mod_inverse

def generate_rsa_keypair(p, q):
    # Check if both numbers are prime
    if not (isprime(p) and isprime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be the same')

    # Compute n (modulus) and Euler's Totient (phi_n)
    n = p * q
    phi_n = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi_n and gcd(e, phi_n) = 1
    e = 5 # Example choice of e

    g = gcd(e, phi_n)
    while g != 1:
        e = random.randrange(2, phi_n - 1)
        g = gcd(e, phi_n)

    # Compute d such that (d * e) % phi_n = 1
    d = mod_inverse(e, phi_n)

    # Return public key (e, n) and private key (d, n)
    return ((e, n), (d, n))

# Example usage
p, q = 17, 19

# Generally Large primes

# Generate public key (e, n) and private key (d, n)
public_key, private_key = generate_rsa_keypair(p, q)

# Step 2: Encryption
def encrypt(message, public_key):
    e, n = public_key
    # Encrypt the message using the public key
```

```
    return pow(message, e, n)

# Step 3: Decryption
def decrypt(ciphertext, private_key):
    d, n = private_key
    # Decrypt the ciphertext using the private key
    return pow(ciphertext, d, n)

# Example usage
message = 12 # Example message
ciphertext = encrypt(message, public_key)
decrypted_message = decrypt(ciphertext, private_key)
```

4 Drawbacks of Classical Systems for Breaking RSA

4.1 Limitations of Classical Factoring Algorithms

Classical computing relies on algorithms that are inefficient for factoring large integers. Some of these algorithms include:

- Trial division: Simple but extremely inefficient for large numbers
- Pollard's rho algorithm: Faster than trial division but still impractical for RSA-sized numbers
- Quadratic sieve: More efficient but still exponential complexity
- General number field sieve: The fastest known classical algorithm, but still requires exponential time

4.2 Computational Barriers

The best-known classical algorithms, such as the General Number Field Sieve, have sub-exponential but still prohibitive time complexity:

$$O(e^{(c \log N)^{1/3} (\log \log N)^{2/3}})$$

This makes it computationally infeasible to factor large integers (such as 2048-bit RSA keys) within a reasonable timeframe using classical computers.

4.3 Security by Computational Difficulty

The security of RSA in the classical computing paradigm is not based on mathematical proof of security, but rather on the practical difficulty of factoring. This creates a potential vulnerability if more efficient factoring algorithms or computing methods are discovered.

5 Proposed System: Quantum Computing and Shor's Algorithm

5.1 Quantum Computing Foundations

5.1.1 Quantum Bits and Superposition

Unlike classical bits that can be either 0 or 1, a qubit can exist in a superposition of states, represented mathematically as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers satisfying $|\alpha|^2 + |\beta|^2 = 1$, and $|0\rangle$ and $|1\rangle$ represent the computational basis states.

5.1.2 Quantum Gates and Circuits

Quantum gates are represented by unitary matrices that transform qubits while preserving the norm of the quantum state.

Common Quantum Gates

- **Hadamard Gate (H):** Creates superposition

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

- **Pauli Gates:**

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

- **CNOT Gate:** Two-qubit gate that flips the target qubit if the control qubit is $|1\rangle$

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

5.1.3 Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is a key component of Shor's algorithm and many other quantum algorithms. It is the quantum counterpart of the discrete Fourier transform.

Mathematical Definition For an n-qubit state $|j\rangle$, the QFT is defined as:

$$QFT|j\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi i jk/2^n} |k\rangle$$

Circuit Implementation The QFT can be implemented using Hadamard and controlled rotation gates. For an n-qubit system, the circuit depth is $O(n^2)$.

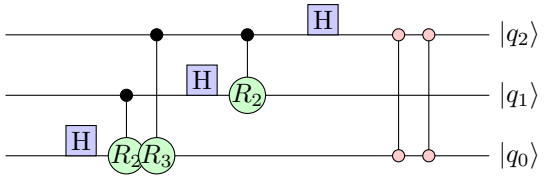

Figure 1: Quantum circuit for 3-qubit Quantum Fourier Transform

5.2 Shor's Algorithm: Quantum Threat to RSA

5.2.1 Overview of Shor's Algorithm

Quantum computing introduces a significant threat to RSA encryption through Shor's algorithm. Shor's algorithm can factor large integers in polynomial time, which is exponentially faster than the best-known classical algorithms.

5.2.2 Period Finding in Shor's Algorithm

The core of Shor's algorithm is the period-finding problem. Given a function $f(x) = a^x \mod N$ for some integer a coprime to N, the goal is to find the period r such that $f(x + r) = f(x)$ for all x.

5.2.3 Quantum Circuit for Period Finding

The quantum circuit for period finding uses two registers:

1. First register: n qubits initialized to $|0\rangle$
2. Second register: m qubits initialized to $|0\rangle$, where $m \approx \log_2 N$

The steps are:

1. Apply Hadamard gates to all qubits in the first register to create a superposition
2. Apply the function $f(x) = a^x \mod N$ as a unitary operation
3. Apply the Quantum Fourier Transform to the first register
4. Measure the first register

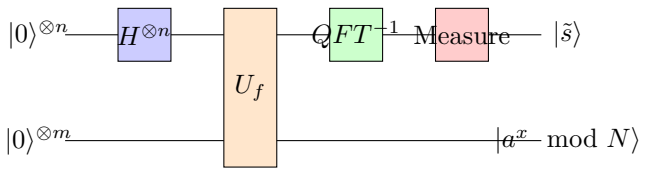

Figure 2: Quantum circuit for period finding in Shor's algorithm

5.2.4 Detailed Algorithm Steps

Algorithm 2 Shor's Algorithm

Input: Integer N to be factored
Output: Prime factors of N
Choose a random integer a such that $1 < a < N$ Compute $\gcd(a, N)$ **if** $\gcd(a, N) \neq 1$ **then**
  return $\gcd(a, N)$;
end
Use quantum period finding to find the period r of the function $f(x) = a^x \bmod N$ **if** *r is odd or* $a^{r/2} \equiv -1 \pmod{N}$ **then**
  return "Failure, try again";
end
Compute $p = \gcd(a^{r/2} - 1, N)$ and $q = \gcd(a^{r/2} + 1, N)$ **return** p and q;

5.2.5 Complexity Analysis and Comparison

Shor's algorithm achieves exponential speedup over classical factoring algorithms:

Algorithm	**Time Complexity**	**Space Complexity**
General Number Field Sieve (Classical)	$O(e^{(c \log N)^{1/3} (\log \log N)^{2/3}})$	$O(e^{(c \log N)^{1/3} (\log \log N)^{2/3}})$
Shor's Algorithm (Quantum)	$O((\log N)^2 (\log \log N)(\log \log \log N))$	$O(\log N)$ qubits

Table 1: Complexity comparison between classical and quantum factoring algorithms

6 Advantages of Quantum Computing for Factorization

6.1 Exponential Speedup

Quantum computing offers exponential speedup over classical factoring methods:

- Classical methods require exponential time: $O(e^{(c \log N)^{1/3} (\log \log N)^{2/3}})$

- Shor's algorithm runs in polynomial time: $O((\log N)^2(\log\log N)(\log\log\log N))$

This represents a fundamental breakthrough in computational efficiency for the factorization problem.

6.2 Algorithmic Advantages

The key advantages of quantum computing for factorization include:

- Quantum parallelism: Ability to simultaneously evaluate a function for multiple inputs
- Quantum Fourier Transform: Efficiently extracts the periodicity information crucial for factoring
- Quantum entanglement: Allows correlation between qubits that enhances computational power
- Quantum superposition: Enables exponentially many computational paths simultaneously

6.3 Practical Impact on RSA Security

The impact of quantum factorization on RSA security is profound:

- A sufficiently powerful quantum computer could break RSA encryption regardless of key size
- Most public key infrastructure would need to be replaced
- Data encrypted today could be decrypted in the future when quantum computers become available (harvest now, decrypt later attack)

7 Applications of Quantum-Resistant Cryptography

7.1 Post-Quantum Cryptographic Methods

7.1.1 Alternative Cryptographic Methods

Post-quantum cryptography aims to develop cryptographic algorithms that are secure against quantum attacks. Some promising alternatives to RSA include:

- **Lattice-based Cryptography:** Relies on the hardness of lattice problems, which are believed to be resistant to quantum attacks.
- **Hash-based Cryptography:** Uses hash functions to create secure digital signatures.
- **Code-based Cryptography:** Based on error-correcting codes, providing security against quantum attacks.

7.1.2 NIST Standardization Process

The National Institute of Standards and Technology (NIST) has been conducting a standardization process for post-quantum cryptography since 2017:

- **Round 3 Finalists (2020):** Selected algorithms including CRYSTALS-Kyber, CRYSTALS-Dilithium, FALCON, and SPHINCS+
- **First Standards (2022):** CRYSTALS-Kyber (key encapsulation) and CRYSTALS-Dilithium, FALCON, and SPHINCS+ (digital signatures)
- **Ongoing work:** Additional candidates under consideration for future standardization

7.1.3 Hybrid Cryptographic Approaches

During the transition period, hybrid approaches combining classical and post-quantum cryptography offer a pragmatic solution:

- **TLS hybridization:** Combining classical (e.g., ECDHE) and post-quantum (e.g., Kyber) key exchange
- **Hybrid signatures:** Using both RSA/ECDSA and post-quantum signature schemes

7.2 Critical Infrastructure Protection

Quantum-resistant cryptography has crucial applications in:

- Banking and financial services
- Government communications and national security
- Healthcare data protection
- Smart grid and critical infrastructure
- Blockchain and cryptocurrency technology

7.3 Implementation Challenges

Transitioning to post-quantum cryptography presents several challenges:

- Legacy system compatibility
- Performance overhead of post-quantum algorithms
- Standardization and certification processes
- Hardware and software implementation costs
- User education and adoption barriers

8 Future Enhancements and Conclusion

8.1 Future of Quantum Computing

The development of quantum computing technology is progressing rapidly:

- Increasing qubit counts and coherence times
- Improved error correction techniques
- Development of quantum networking and quantum internet
- Hybrid classical-quantum computing approaches

8.2 Cryptographic Evolution

The future of cryptography will likely involve:

- Continual development of quantum-resistant algorithms
- Quantum cryptography (e.g., quantum key distribution)
- Homomorphic encryption for privacy-preserving computation
- Zero-knowledge proof systems for authentication
- New mathematical foundations for cryptographic security

8.3 Conclusion

The security of RSA encryption relies on the difficulty of factoring large integers. Quantum computing, with its ability to efficiently factor large integers using Shor's algorithm, poses a significant threat to the RSA cryptosystem. As quantum computing technology advances, it becomes increasingly important to develop new cryptographic methods that can withstand quantum attacks and prepare for the transition to a post-quantum cryptographic landscape. The race between quantum computing capabilities and quantum-resistant cryptography will define the next era of information security.

A Glossary of Terms

- **Qubit**: The fundamental unit of quantum information, analogous to a classical bit.
- **Superposition**: A quantum mechanical property where a qubit can exist in multiple states simultaneously.
- **Quantum Entanglement**: A quantum mechanical phenomenon where qubits become correlated in such a way that the quantum state of each qubit cannot be described independently.
- **RSA**: A public-key cryptosystem named after its inventors Rivest, Shamir, and Adleman.
- **Shor's Algorithm**: A quantum algorithm for integer factorization developed by Peter Shor in 1994.
- **Post-Quantum Cryptography**: Cryptographic algorithms believed to be secure against attacks by quantum computers.

B Sample Code Implementation

B.1 RSA Implementation in Python

```
# Complete RSA Implementation
import random
from sympy import gcd, isprime, mod_inverse

def generate_rsa_keypair(p, q):
    # Check if both numbers are prime
    if not (isprime(p) and isprime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be the same')

    # Compute n (modulus) and Euler's Totient (phi_n)
    n = p * q
    phi_n = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi_n and gcd(e, phi_n) = 1
    e = 5 # Example choice of e

    g = gcd(e, phi_n)
    while g != 1:
        e = random.randrange(2, phi_n - 1)
        g = gcd(e, phi_n)

    # Compute d such that (d * e) % phi_n = 1
```

```
    d = mod_inverse(e, phi_n)

    # Return public key (e, n) and private key (d, n)
    return ((e, n), (d, n))

# Example usage
p, q = 17, 19

# Generate public key (e, n) and private key (d, n)
public_key, private_key = generate_rsa_keypair(p, q)

# Encryption function
def encrypt(message, public_key):
    e, n = public_key
    # Encrypt the message using the public key
    return pow(message, e, n)

# Decryption function
def decrypt(ciphertext, private_key):
    d, n = private_key
    # Decrypt the ciphertext using the private key
    return pow(ciphertext, d, n)

# Demonstration
message = 12 # Example message
ciphertext = encrypt(message, public_key)
decrypted_message = decrypt(ciphertext, private_key)
print(f"Original message: {message}")
print(f"Encrypted message: {ciphertext}")
print(f"Decrypted message: {decrypted_message}")
```

B.2 Quantum Period Finding Simulation (Simplified)

```
# Simplified simulation of period finding using Qiskit
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
import numpy as np
import matplotlib.pyplot as plt

# Parameters
N = 15 # Number to factor
a = 7  # Random number coprime to N

# Create quantum circuit for period finding
def create_period_finding_circuit(N, a, n_count):
    # n_count: number of counting qubits

    # Create quantum circuit
```

```
    qc = QuantumCircuit(n_count, n_count)

    # Apply Hadamard to all counting qubits
    for i in range(n_count):
        qc.h(i)

    # Apply controlled-U^(2^j) operations
    # In a real implementation, this would implement
    # modular exponentiation
    # For simplicity, we're just showing the structure
    for j in range(n_count):
        qc.x(j) # Placeholder for controlled modular exponentiation
        qc.x(j) # Reverse the placeholder

    # Apply inverse QFT
    for i in range(n_count//2):
        qc.swap(i, n_count-i-1)

    for i in range(n_count):
        qc.h(i)
        for j in range(i+1, n_count):
            qc.cp(-np.pi/float(2**(j-i)), j, i)

    # Measure all qubits
    qc.measure(range(n_count), range(n_count))

    return qc

# Create and simulate the circuit
n_count = 8 # Number of counting qubits
circuit = create_period_finding_circuit(N, a, n_count)
simulator = Aer.get_backend('qasm_simulator')
job = execute(circuit, simulator, shots=1024)
result = job.result()
counts = result.get_counts()

# In a real implementation, we would analyze the results to find
# the period and continue with Shor's algorithm to find
# the factors
```