🔐 Criptografia RSA: Explicação Técnica Detalhada

📊 Índice

🚀 Introdução ao RSA

O que é RSA?

RSA (Rivest-Shamir-Adleman) é um algoritmo de **criptografia assimétrica** desenvolvido em 1977. É um dos primeiros e mais amplamente utilizados sistemas de chave pública.

Características Principais

- **Assimétrico**: Usa duas chaves diferentes (pública e privada)
- **Baseado em problema matemático**: Dificuldade de fatorar números grandes
- **Bidirecional**: Pode criptografar e assinar digitalmente
- **Amplamente adotado**: Base de muitos protocolos (HTTPS, SSH, etc.)

Princípio Básico

```
Alice possui:      Bob possui:
- Chave Pública    - Chave Pública de Alice
- Chave Privada    - Sua própria chave privada

Fluxo:
1. Bob usa a chave pública de Alice para criptografar
2. Apenas Alice pode descriptografar (com sua chave privada)
```

🧮 Fundamentos Matemáticos

1. Aritmética Modular

A **aritmética modular** é fundamental para o RSA:

```
a ≡ b (mod n)  significa que  a mod n = b mod n

Propriedades importantes:
- (a + b) mod n = ((a mod n) + (b mod n)) mod n
- (a × b) mod n = ((a mod n) × (b mod n)) mod n
- (a^k) mod n pode ser calculado eficientemente
```

2. Números Primos

Por que primos são importantes:

- Teorema Fundamental da Aritmética: Todo número tem fatoração prima única
- **Problema da fatoração**: Dado n = p × q, encontrar p e q é computacionalmente difícil para números grandes

3. Função Totiente de Euler φ(n)

```
ϕ(n) = quantidade de números menores que n que são coprimos com n

Para números primos:
- ϕ(p) = p - 1

Para produto de primos:
- ϕ(p × q) = ϕ(p) × ϕ(q) = (p-1) × (q-1)
```

4. Teorema de Euler

```
Se gcd(a, n) = 1, então:
a^ϕ(n) ≡ 1 (mod n)

Corolário importante para RSA:
a^(k×ϕ(n) + 1) ≡ a (mod n)
```

5. Inverso Modular

```
d é o inverso modular de e módulo ϕ(n) se:
e × d ≡ 1 (mod ϕ(n))

Calculado usando o Algoritmo Euclidiano Estendido
```

⚙️ Algoritmo RSA Passo-a-Passo

Fase 1: Geração de Chaves

Passo 1: Gerar Números Primos

```
1. Gere dois números primos distintos: p e q
2. p e q devem ter tamanhos similares
3. Para chaves de n bits: p e q ≈ n/2 bits cada
```

Implementação:

```
let p = gerar_primo(bits / 2);  // Ex: 256 bits
let q = gerar_primo(bits / 2);  // Ex: 256 bits
```

Passo 2: Calcular o Módulo

```
n = p × q

n será público e define o "tamanho" da chave
```

Implementação:

```
let n = &p * &q;  // Módulo público (ex: 512 bits)
```

Passo 3: Calcular φ(n)

```
ϕ(n) = ϕ(p × q) = ϕ(p) × ϕ(q) = (p-1) × (q-1)

ϕ(n) deve ser mantido em segredo!
```

Implementação:

```
let phi_n = (&p - 1) * (&q - 1);
```

Passo 4: Escolher Expoente Público e

```
Escolha e tal que:
1. 1 < e < ϕ(n)
```

```
2. gcd(e, ϕ(n)) = 1

Valor comum: e = 65537 = 2^16 + 1
```

Por que 65537?

- É primo
- Tem poucos bits "1" em binário → exponenciação eficiente
- Amplamente testado e considerado seguro

Passo 5: Calcular Expoente Privado d

```
d é o inverso modular de e módulo ϕ(n):
e × d ≡ 1 (mod ϕ(n))

Calculado usando Algoritmo Euclidiano Estendido
```

Implementação:

```rust
let d = inverso_modular(&e, &phi_n);
```

Resultado Final

```
Chave Pública:  (n, e)  - pode ser compartilhada
Chave Privada:  (n, d)  - deve ser mantida em segredo
```

Fase 2: Criptografia

Para criptografar uma mensagem m:

```
c = m^e mod n

Onde:
- m = mensagem (número < n)
- c = texto criptografado
- (n, e) = chave pública
```

Fase 3: Descriptografia

Para descriptografar um texto criptografado c:

```
m = c^d mod n

Onde:
- c = texto criptografado
- m = mensagem original
- (n, d) = chave privada
```

Prova Matemática (Por que funciona?)

```
Queremos provar: (m^e)^d ≡ m (mod n)

1. c = m^e mod n
2. m' = c^d mod n = (m^e)^d mod n = m^(ed) mod n

Como e × d ≡ 1 (mod ϕ(n)), então:
ed = k × ϕ(n) + 1 para algum inteiro k

Portanto:
m^(ed) = m^(k×ϕ(n) + 1) = (m^ϕ(n))^k × m

Pelo Teorema de Euler: m^ϕ(n) ≡ 1 (mod n)
Logo: m^(ed) ≡ 1^k × m ≡ m (mod n)
```

💻 Implementação Detalhada

1. Teste de Primalidade Miller-Rabin

```rust
fn eh_primo(n: &BigInt, k: u32) -> bool {
    // Escrever n-1 como d × 2^r onde d é ímpar
    let mut r = 0;
    let mut d = n - 1;
    while &d % 2 == 0 {
        d /= 2;
        r += 1;
    }

    // Executar k rounds de teste
    for _ in 0..k {
        let a = random_range(2, n-1);
        let mut x = mod_exp(&a, &d, n);

        if x == 1 || x == n - 1 {
            continue; // Provavelmente primo
        }

        for _ in 0..r-1 {
```

```
                x = mod_exp(&x, &2, n);
                if x == n - 1 {
                    continue 'outer; // Provavelmente primo
                }
            }
            return false; // Definitivamente composto
        }
        true // Provavelmente primo
    }
```

Complexidade: O(k × log³ n)
Precisão: Probabilidade de erro < (1/4)^k

2. Exponenciação Modular Rápida

```
fn exponenciacao_modular(base: &BigInt, exp: &BigInt, modulo: &BigInt) ->
BigInt {
    let mut resultado = 1;
    let mut base = base % modulo;
    let mut exp = exp.clone();

    while exp > 0 {
        if &exp % 2 == 1 {
            resultado = (resultado * &base) % modulo;
        }
        exp >>= 1;                    // exp = exp / 2
        base = (&base * &base) % modulo; // base = base²
    }
    resultado
}
```

Algoritmo "Square-and-Multiply":

- Complexidade: O(log exp)
- Evita calcular números gigantescos
- Fundamental para viabilizar o RSA

Exemplo: Calcular 7^13 mod 11

```
13 = 1101₂ (binário)

7¹ mod 11 = 7
7² mod 11 = 5
7⁴ mod 11 = 3
7⁸ mod 11 = 9

7^13 = 7^(8+4+1) = 7⁸ × 7⁴ × 7¹ = 9 × 3 × 7 = 189 ≡ 2 (mod 11)
```

3. Algoritmo Euclidiano Estendido

```
fn algoritmo_euclidiano_estendido(a: &BigInt, b: &BigInt) -> (BigInt, BigInt,
BigInt) {
    if *a == 0 {
        return (b.clone(), 0, 1); // gcd = b, x = 0, y = 1
    }

    let (gcd, x1, y1) = algoritmo_euclidiano_estendido(&(b % a), a);
    let x = &y1 - (b / a) * &x1;
    let y = x1;

    (gcd, x, y)  // Retorna (gcd, x, y) onde ax + by = gcd
}
```

Encontra coeficientes x, y tal que: ax + by = gcd(a, b)

Para inverso modular: Se gcd(e, φ(n)) = 1, então e × x ≡ 1 (mod φ(n))

4. Conversão Mensagem ↔ Números

```
fn string_para_numeros(texto: &str) -> Vec<BigInt> {
    texto.bytes()
         .map(|byte| byte.to_bigint().unwrap())
         .collect()
}

fn numeros_para_string(numeros: &[BigInt]) -> String {
    numeros.iter()
           .map(|num| num.to_bytes_be().1[0] as char)
           .collect()
}
```

Limitação atual: Processa um caractere por vez
Em produção: Usa-se padding (OAEP) e processa blocos maiores

🔒 Análise de Segurança

1. Segurança Baseada em Problemas Matemáticos

Problema da Fatoração:

```
Dado: n = p × q (onde p, q são primos)
Encontrar: p e q

Dificuldade: Não existe algoritmo eficiente conhecido para números grandes
```

Melhor algoritmo clássico: General Number Field Sieve (GNFS)

- Complexidade: O(exp((64/9 × log n × log log n)^(1/3)))
- Para n de 2048 bits: ~2^112 operações

2. Tamanhos de Chave Recomendados

Ano	Tamanho Mínimo	Equivalência Simétrica	Status
2010	1024 bits	80 bits	❌ Quebrado
2015	2048 bits	112 bits	☑ Seguro atual
2025	3072 bits	128 bits	☑ Recomendado
2030	4096 bits	140 bits	🔮 Futuro

3. Ataques Conhecidos

A. Ataques à Implementação

- **Side-Channel**: Análise de tempo, consumo de energia
- **Fault Injection**: Induzir erros durante cálculos
- **Cache Attacks**: Explorar padrões de acesso à memória

B. Ataques Matemáticos

- **Pequeno expoente privado**: Se d é pequeno
- **Comum modulo**: Reusar n com diferentes e
- **Baixa entropia**: p e q não verdadeiramente aleatórios

C. Ataques de Padding

- **Bleichenbacher**: Contra PKCS#1 v1.5
- **Chosen Ciphertext**: Explorar oráculos de padding

4. Contramedidas

```rust
// 1. Usar números verdadeiramente aleatórios
let mut rng = OsRng;  // Gerador criptograficamente seguro

// 2. Verificar qualidade dos primos
assert!(p != q);                        // Primos diferentes
assert!((p-1).gcd(&(q-1)) < threshold); // Evitar fatores comuns

// 3. Usar padding seguro (OAEP)
let padded = oaep_pad(message, n.bits());
```

```
// 4. Proteger contra timing attacks
let result = constant_time_exp(base, exp, modulo);
```

⚡ Otimizações e Considerações

1. Otimizações Matemáticas

Chinese Remainder Theorem (CRT)

```
Em vez de calcular: m = c^d mod n
Calcular:
- m₁ = c^(d mod (p-1)) mod p
- m₂ = c^(d mod (q-1)) mod q
- Combinar m₁ e m₂ para obter m

Speedup: ~4x mais rápido
```

Pre-computação

```
struct ChavePrivadaOtimizada {
    n: BigInt,
    d: BigInt,
    p: BigInt,          // Primo secreto
    q: BigInt,          // Primo secreto
    dp: BigInt,         // d mod (p-1)
    dq: BigInt,         // d mod (q-1)
    qinv: BigInt,       // q^(-1) mod p
}
```

2. Considerações de Performance

Operações mais custosas:

1. Geração de primos: $O(\log^4 n)$
2. Exponenciação modular: $O(\log^3 n)$
3. Inversão modular: $O(\log^2 n)$

Estratégias:

- Cache de primos pequenos
- Uso de Montgomery multiplication
- Paralelização quando possível

3. Gerenciamento de Memória

```
// Limpar dados sensíveis da memória
impl Drop for ChavePrivada {
    fn drop(&mut self) {
        self.d.assign(&0.into());       // Zerar expoente privado
        self.p.assign(&0.into());       // Zerar primo secreto
        self.q.assign(&0.into());       // Zerar primo secreto
    }
}
```

🚨 Limitações e Ataques

1. Limitações Fundamentais

Tamanho da Mensagem

```
Mensagem deve ser: m < n

Para n de 2048 bits: máximo ~255 bytes por bloco
Solução: Usar com criptografia simétrica (híbrida)
```

Performance

```
RSA é ~1000x mais lento que AES
Uso típico: Criptografar chave simétrica, não dados diretamente
```

2. Ataques Quânticos

Algoritmo de Shor (1994):

- Quebra RSA em tempo polinomial
- Requer computador quântico com ~4096 qubits lógicos
- Estimativa atual: 10-20 anos para implementação prática

Contramedida: Migrar para criptografia pós-quântica

3. Implementações Inseguras

Problemas Comuns

```
// ❌ NUNCA fazer isso:
if message.len() > key_size {
    panic!("Message too long");  // Timing attack
```

```
}

// ❌ Expoente privado pequeno
if d.bits() < n.bits() / 4 {
    // Vulnerable to Wiener's attack
}

// ❌ Reusar números aleatórios
let r = thread_rng().gen_bigint(256);  // Deve ser único!
```

Implementação Segura

```
// ✅ Fazer assim:
fn secure_decrypt(ciphertext: &[u8], key: &PrivateKey) -> Result<Vec<u8>,
Error> {
    // 1. Validação constante de tempo
    let valid = constant_time_validate(ciphertext, key);

    // 2. Descriptografia sempre executada
    let result = rsa_decrypt_raw(ciphertext, key);

    // 3. Retorno baseado na validação
    if valid {
        Ok(result)
    } else {
        Err(Error::InvalidCiphertext)
    }
}
```

🔄 Comparação com Outros Algoritmos

RSA vs. ECC (Elliptic Curve Cryptography)

Aspecto	RSA 2048	ECC 256
Segurança	~112 bits	~128 bits
Tamanho chave	2048 bits	256 bits
Velocidade	Lento	Mais rápido
Adoção	Universal	Crescente
Pós-quântico	❌ Vulnerável	❌ Vulnerável

RSA vs. Algoritmos Pós-Quânticos

Algoritmo	Tipo	Tamanho Chave	Status NIST

Algoritmo	Tipo	Tamanho Chave	Status NIST
Kyber	Lattice-based	~1KB	☑ Padrão
Dilithium	Lattice-based	~1.3KB	☑ Padrão
SPHINCS+	Hash-based	~32 bytes	☑ Padrão
RSA	Number theory	256 bytes	⚠ Legado

📋 Resumo e Conclusões

Pontos-Chave do RSA

1. **Fundamento Matemático Sólido**
   - Baseado em problemas bem estudados
   - Segurança demonstrável matematicamente
2. **Versatilidade**
   - Criptografia e assinatura digital
   - Base para muitos protocolos
3. **Maturidade**
   - 45+ anos de análise criptográfica
   - Implementações bem testadas

Limitações Importantes

1. **Performance**
   - Lento comparado a algoritmos simétricos
   - Requer otimizações cuidadosas
2. **Vulnerabilidade Quântica**
   - Quebrado pelo algoritmo de Shor
   - Necessita migração futura
3. **Complexidade de Implementação**
   - Muitos detalhes críticos para segurança
   - Fácil de implementar incorretamente

Uso Recomendado

```
☑ USAR RSA para:
- Troca de chaves simétricas
- Assinatura digital (com PSS)
```

```
- Sistemas legados que requerem

❌ NÃO usar RSA para:
- Criptografia de dados grandes
- Novos sistemas (preferir ECC)
- Sistemas que precisam ser pós-quânticos
```

Implementação Educacional vs. Produção

Esta implementação é educacional porque:

- Chaves pequenas (512 bits)
- Sem padding seguro
- Sem proteções contra side-channel
- Foco na clareza, não na segurança

Para produção, use:

- Bibliotecas testadas (OpenSSL, BoringSSL)
- Chaves ≥ 2048 bits
- Padding OAEP para criptografia
- PSS para assinatura digital
- Proteções contra timing attacks

🎓 **"O RSA nos ensina que a matemática pode ser tanto elegante quanto prática, fornecendo segurança através da beleza dos números primos."**

Documentação técnica criada para fins educacionais - Use bibliotecas profissionais em produção