Sistema Criptográfico RSA - Teoria e Aplicação

Iago Nicolas Lopes Lima

2 de junho de 2024

Resumo

Este relatório apresenta uma análise detalhada do sistema criptográfico RSA, abordando sua teoria fundamental e aplicação prática. Desenvolvido em 1977 por Ronald Rivest, Adi Shamir e Leonard Adleman, o RSA se destaca como um dos métodos mais seguros e amplamente utilizados na criptografia assimétrica. O trabalho explora os fundamentos teóricos do RSA, suas aplicações práticas, e oferece uma implementação em Python. Os principais resultados mostram que, embora o RSA seja eficaz na proteção de dados, ele enfrenta desafios relacionados ao desempenho e à gestão de chaves. A análise detalha esses desafios e propõe possíveis melhorias, consolidando o RSA como uma ferramenta essencial na segurança da informação.

Introdução/Motivação

A criptografia desempenha um papel vital na proteção das informações na era digital. Com a crescente quantidade de dados sendo transmitidos e armazenados eletronicamente, a segurança dessas informações tornou-se uma prioridade absoluta. O algoritmo RSA, desenvolvido por Ronald Rivest, Adi Shamir e Leonard Adleman em 1977, é um dos marcos mais importantes na evolução da criptografia. Este projeto tem como objetivo explorar profundamente a teoria por trás do RSA e avaliar sua aplicação prática em contextos modernos de segurança da informação.

Objetivos

Os objetivos principais deste projeto incluem:

- Explorar e entender a teoria fundamental do sistema criptográfico RSA.
- Implementar o algoritmo RSA em um ambiente de programação controlado.
- Avaliar a eficácia do RSA na criptografia e descriptografia de dados.
- Identificar desafios e limitações na aplicação prática do RSA, sugerindo possíveis melhorias.

Requisitos

Para a execução deste projeto, são necessários alguns pré-requisitos:

- Conhecimento básico de teoria dos números e álgebra, especialmente conceitos relacionados a números primos e fatoração.
- Ferramentas de programação, preferencialmente Python, devido à sua simplicidade e ampla gama de bibliotecas matemáticas.
- Dados de teste para verificar a eficácia da criptografia e descriptografia.

Roteiro

O projeto segue um roteiro estruturado em várias etapas, que incluem:

1. Revisão de literatura sobre criptografia assimétrica e o algoritmo RSA.
2. Estudo aprofundado da teoria matemática por trás do RSA, incluindo números primos, fatoração e aritmética modular.
3. Implementação prática do RSA utilizando Python.
4. Realização de testes de criptografia e descriptografia com diferentes tipos e tamanhos de dados.
5. Análise dos resultados obtidos, identificando desafios e propondo melhorias.
6. Conclusões e recomendações finais sobre a aplicação do RSA.

A motivação para este projeto é clara: em um mundo cada vez mais digital, a segurança da informação é crucial. O RSA, como um dos algoritmos mais confiáveis e robustos, oferece uma excelente oportunidade para explorar e entender os mecanismos subjacentes à criptografia assimétrica. Além disso, este estudo contribui para o desenvolvimento contínuo de práticas seguras de comunicação e armazenamento de dados.

Fundamentação Teórica

O RSA é um algoritmo de criptografia assimétrica que utiliza duas chaves distintas: uma chave pública para criptografar dados e uma chave privada para descriptografá-los. A segurança do RSA baseia-se na dificuldade de fatorar grandes números primos, um problema matemático que ainda não possui uma solução eficiente para números extremamente grandes.

Teoria dos Números

A teoria dos números é uma área fundamental da matemática que estuda as propriedades dos números inteiros. Dois conceitos essenciais para o RSA são os números primos e a fatoração de inteiros.

Números Primos

Números primos são números naturais maiores que 1 que não podem ser formados pela multiplicação de dois números naturais menores. Exemplos incluem 2, 3, 5, 7, 11, 13, e assim por diante. A importância dos números primos na criptografia RSA reside no fato de que a multiplicação de dois números primos grandes é fácil, mas a fatoração do produto resultante é extremamente difícil.

Fatoração de Inteiros

Fatorar um número inteiro significa expressá-lo como um produto de números primos. Por exemplo, 15 pode ser fatorado em 3 e 5. A dificuldade de fatorar números grandes é a base da segurança do RSA. Se um adversário consegue fatorar o produto de dois números primos grandes usados na chave RSA, ele pode comprometer a segurança do sistema.

Aritmética Modular

A aritmética modular é um sistema de aritmética para inteiros, onde os números "reiniciam"após atingir um certo valor, denominado módulo. A operação mais comum na aritmética modular é o cálculo do resto de uma divisão inteira, conhecida como operação de módulo. Por exemplo, $17 \mod 5 = 2$, porque 17 dividido por 5 é 3 com um resto de 2.

O Algoritmo RSA

O algoritmo RSA envolve três etapas principais: geração de chaves, criptografia e descriptografia.

Geração de Chaves

A geração de chaves no RSA envolve os seguintes passos:

1. Escolha de dois números primos grandes, p e q.
2. Cálculo de $n = p \times q$, onde n é utilizado como o módulo tanto para a chave pública quanto para a chave privada.
3. Cálculo do totiente de n, $\phi(n) = (p-1)(q-1)$.
4. Escolha de um número inteiro e tal que $1 < e < \phi(n)$ e e seja coprimo a $\phi(n)$. O valor e torna-se o expoente da chave pública.
5. Cálculo do expoente da chave privada d, que é o inverso multiplicativo de e módulo $\phi(n)$, ou seja, $d \times e \equiv 1 \pmod{\phi(n)}$.

Justificação dos Teoremas Utilizados

A segurança e funcionamento do RSA dependem de vários teoremas fundamentais da teoria dos números. Abaixo, apresentamos os teoremas utilizados e suas justificações.

Teorema da Aritmética Modular O Teorema da Aritmética Modular, que é a base para o cálculo de e e d, afirma que para qualquer número inteiro a e qualquer módulo m, existe um número inteiro k tal que:

$$a^k \mod m$$

A aritmética modular é usada para garantir que as operações de criptografia e descriptografia permaneçam dentro de um intervalo manejável de valores inteiros.

Pequeno Teorema de Fermat O Pequeno Teorema de Fermat afirma que se p é um número primo e a é um número inteiro tal que p não divide a, então:

$$a^{p-1} \equiv 1 \pmod{p}$$

Este teorema é utilizado para provar que a exponenciação modular usada no RSA é correta. Ele garante que:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

quando a é coprimo a n.

Teorema de Euler O Teorema de Euler generaliza o Pequeno Teorema de Fermat. Ele afirma que se n é um número inteiro positivo e a é um inteiro coprimo a n, então:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

onde $\phi(n)$ é a função totiente de Euler, que conta o número de inteiros positivos menores que n que são coprimos a n. Este teorema é crucial para a definição da chave privada d, garantindo que a operação de descriptografia reverta corretamente a operação de criptografia.

Algoritmo de Euclides Estendido O Algoritmo de Euclides Estendido é utilizado para encontrar o inverso multiplicativo de e módulo $\phi(n)$. Este algoritmo é uma extensão do Algoritmo de Euclides, que encontra o máximo divisor comum (MDC) de dois inteiros. O Algoritmo de Euclides Estendido permite encontrar inteiros x e y tais que:

$$ax + by = \text{MDC}(a, b)$$

No caso do RSA, ele é usado para encontrar d tal que:

$$d \times e \equiv 1 \pmod{\phi(n)}$$

garantindo assim que d seja o inverso multiplicativo de e módulo $\phi(n)$.

Criptografia e Descriptografia

Criptografia

Para criptografar uma mensagem m:

1. Converte-se a mensagem m em um número M tal que $0 \leq M < n$.
2. Calcula-se o texto cifrado C utilizando a fórmula $C = M^e \mod n$.

Descriptografia

Para descriptografar o texto cifrado C:

1. Calcula-se a mensagem M utilizando a fórmula $M = C^d \mod n$.
2. Converte-se o número M de volta para a mensagem original m.

Segurança do RSA

A segurança do RSA depende da dificuldade de fatorar o produto de dois números primos grandes. Embora algoritmos eficientes de fatoração existam, eles não são práticos para números suficientemente grandes. Isso torna a criptografia RSA segura contra ataques de força bruta e fatoração direta.

Algoritmos Relacionados

Além do RSA, existem outros algoritmos de criptografia assimétrica, como o ElGamal e o ECC (Elliptic Curve Cryptography). Cada um desses algoritmos possui suas próprias vantagens e desvantagens, dependendo da aplicação específica. Comparativamente, o RSA é amplamente utilizado devido à sua simplicidade e robustez.

Assinatura

SHA-256 (Secure Hash Algorithm 256-bit) é uma função de hash criptográfica que produz um hash de 256 bits (32 bytes) a partir de uma mensagem de entrada de qualquer tamanho. Ele é utilizado para garantir a integridade e a autenticidade das mensagens.

Geração da assinatura

1. **Hashing da Mensagem:**
   - A mensagem M é processada através do SHA-256 para produzir um hash H;
   - Este hash H é uma representação fixa e única da mensagem M: $H = SHA - 256(M)$.
2. **Criptografia do Hash:**
   - O hash H é então criptografado usando a chave privada d do remetente. O resultado é a assinatura digital S: $S = H^d \bmod n$.
3. **Verificação da Assinatura:**
   - O destinatário calcula o hash H' da mensagem recebida M utilizando SHA-256: $H' = SHA - 256(M)$;
   - A assinatura S é descriptografada usando a chave pública e do remetente para obter o hash H: $H = S^e \bmod n$;
   - O destinatário compara o hash calculado H' com o hash descriptografado H. Se $H' = H$, a assinatura é válida.

Vulnerabilidades do RSA

Vamos considerar um sistema RSA onde o usuário A tem uma chave pública composta por n e e. Suponhamos os seguintes valores:

- $p = 61$
- $q = 53$
- Portanto, $n = p \times q = 61 \times 53 = 3233$
- O expoente público $e = 17$

Considere também uma mensagem $m = 65$.

Como a chave de A é quebrada

Primeiro, verificamos se $\text{MDC}(m, n) \neq 1$. Vamos usar a função de Máximo Divisor Comum (MDC).

Cálculo do MDC

Para $m = 65$ e $n = 3233$:

$$\text{MDC}(65, 3233)$$

Usando o algoritmo de Euclides:

$$\begin{aligned} 3233 &= 65 \times 49 + 48 \\ 65 &= 48 \times 1 + 17 \\ 48 &= 17 \times 2 + 14 \\ 17 &= 14 \times 1 + 3 \\ 14 &= 3 \times 4 + 2 \\ 3 &= 2 \times 1 + 1 \\ 2 &= 1 \times 2 + 0 \end{aligned}$$

Portanto, $\text{MDC}(65, 3233) = 1$.

Para fins de exemplo, suponhamos $m = 3230$:

$$3233 = 3230 \times 1 + 3$$
$$3230 = 3 \times 1076 + 2$$
$$3 = 2 \times 1 + 1$$
$$2 = 1 \times 2 + 0$$

Então, $\text{MDC}(3230, 3233) = 3$.

Fatoração usando o MDC

Encontramos um fator de n, que é 3. Agora podemos encontrar o outro fator:

$$\frac{n}{3} = \frac{3233}{3} = 1077.67$$

Isso indica que 3 e 1077.67 são os fatores de n. Podemos usar a mesma lógica para números mais apropriados onde $\text{MDC}(m, n) \neq 1$ resulta em fatores corretos.

Como falsificar a assinatura de A

Após encontrar p e q, podemos calcular a chave privada d.

Calcular $\phi(n)$

$$\phi(n) = (p-1)(q-1) = (61-1)(53-1) = 60 \times 52 = 3120$$

Encontrar d

Calcular d tal que $d \times e \equiv 1 \mod \phi(n)$:

$$e = 17$$
$$\phi(n) = 3120$$

Usando o algoritmo estendido de Euclides:

$$3120 = 17 \times 183 + 9$$
$$17 = 9 \times 1 + 8$$
$$9 = 8 \times 1 + 1$$
$$8 = 1 \times 8 + 0$$

Portanto, $d \equiv 2753 \mod 3120$.

Falsificação de Assinatura

Suponhamos que a mensagem $m' = 89$.

1. Calcular o hash H de m' usando SHA-256 (simplificado como $H = 89$ para este exemplo).
2. Criar a assinatura falsa S:

$$S = H^d \mod n$$
$$S = 89^{2753} \mod 3233$$

O sistema de decifragem funciona normalmente

Mesmo se $\text{MDC}(m, n) \neq 1$, o processo de decifragem funciona normalmente.

Criptografia

Para um texto cifrado C:

$$C = m^e \mod n$$
$$C = 65^{17} \mod 3233$$
$$C = 2790$$

Decifragem

Para decifrar:

$$m' = C^d \mod n$$
$$m' = 2790^{2753} \mod 3233$$
$$m' = 65$$

Portanto, a decifragem reverte corretamente para m, provando que o sistema funciona mesmo com $\text{MDC}(m, n) \neq 1$.

Calcule a probabilidade de ocorrência dessa situação

Probabilidade de Coprimos

A probabilidade de dois números m e n serem coprimos é $1/\zeta(2)$, onde ζ é a função zeta de Riemann.

$$\zeta(2) = \frac{\pi^2}{6} \approx 1.64493$$

Portanto, a probabilidade de $\text{MDC}(m, n) = 1$ é $1/\zeta(2) \approx 0.60793$.

Probabilidade de Não Serem Coprimos

A probabilidade de $\text{MDC}(m, n) \neq 1$:

$$P(\text{MDC}(m, n) \neq 1) = 1 - 0.60793 \approx 0.39207$$

AES (Advanced Encryption Standard)

O AES (Advanced Encryption Standard) é um algoritmo de criptografia simétrica que utiliza a mesma chave para criptografar e descriptografar dados. Foi adotado como um padrão de criptografia pelo governo dos Estados Unidos e é amplamente utilizado devido à sua segurança e eficiência.

Fundamentação Teórica do AES

O AES opera em blocos de 128 bits e utiliza chaves de 128, 192 ou 256 bits. A segurança do AES é baseada em conceitos da teoria dos números e da álgebra abstrata, como operações em campos finitos.

Estrutura do AES

O AES realiza uma série de operações em cada bloco de dados:

- **SubBytes**: Substituição de cada byte no bloco usando uma tabela de substituição (S-box). A S-box é gerada a partir do inverso multiplicativo em um campo finito ($\mathrm{GF}(2^8)$) combinado com uma transformação afim. Esta operação introduz não-linearidade, fundamental para a segurança do AES.

  $$S(x) = A \cdot x^{-1} \oplus B$$

  onde $S(x)$ é o valor da S-box, x^{-1} é o inverso multiplicativo de x em $\mathrm{GF}(2^8)$, A é uma matriz fixa, e B é um vetor constante.

- **ShiftRows**: Deslocamento circular das linhas da matriz de estado. Esta operação fornece difusão, distribuindo a influência de cada byte sobre o bloco de dados.

  $$\mathrm{ShiftRows}(s) = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

- **MixColumns**: Mistura das colunas da matriz de estado utilizando multiplicações em um campo finito. Esta operação transforma cada coluna da matriz de estado para garantir que a saída de cada rodada dependa de todos os bytes da entrada. A multiplicação em campos finitos, especificamente $\mathrm{GF}(2^8)$, garante que esta operação seja invertível, o que é crucial para a descriptografia.

  $$\mathrm{MixColumns}(a) = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

  onde a multiplicação e adição são realizadas em $\mathrm{GF}(2^8)$.

- **AddRoundKey**: Adição da chave de rodada à matriz de estado utilizando uma operação XOR. Esta operação combina os dados da mensagem com a chave de criptografia, introduzindo segurança dependente da chave.

  $$\mathrm{AddRoundKey}(s, k) = s \oplus k$$

  onde s é a matriz de estado e k é a subchave de rodada.

Cada rodada, exceto a última, consiste em todas as quatro operações. A última rodada omite a operação MixColumns.

Vantagens do AES

- **Eficiência**: AES é altamente eficiente em termos de desempenho, especialmente em hardware, devido à sua capacidade de ser implementado de forma otimizada em diversos dispositivos.

- **Segurança**: AES oferece um alto nível de segurança, sendo resistente a vários tipos de ataques criptográficos conhecidos, como ataques de força bruta e criptoanálise diferencial.

- **Flexibilidade**: AES pode ser usado com diferentes tamanhos de chave (128, 192 e 256 bits), proporcionando um equilíbrio entre segurança e desempenho.

Operações em Campos Finitos (GF(2^8))

A teoria dos números e a álgebra abstrata fornecem a base para muitas operações no AES, especialmente as realizadas no campo finito GF(2^8). Um campo finito é um conjunto de números onde todas as operações (adição, subtração, multiplicação e divisão) são realizadas de acordo com as regras da aritmética modular.

No contexto do AES, cada byte (8 bits) é tratado como um elemento do campo finito GF(2^8). As operações de substituição e mistura no AES garantem que o algoritmo possui uma alta avalanche, onde uma pequena mudança na entrada resulta em uma grande mudança na saída, uma propriedade crucial para a segurança de um algoritmo criptográfico.

Segurança do AES

A segurança do AES baseia-se na complexidade das operações em campos finitos e na dificuldade de resolver sistemas de equações algébricas envolvendo essas operações. A transformação não-linear (SubBytes) e a difusão (ShiftRows e MixColumns) asseguram que o AES resista a criptoanálises lineares e diferenciais, fornecendo um algoritmo robusto e seguro para criptografia de dados.

Metodologia

A metodologia para este projeto envolveu a implementação do algoritmo RSA, seguida por uma série de testes para verificar sua eficácia e desempenho. Abaixo estão as etapas detalhadas do projeto.

Verificação de Requisitos

Os requisitos do sistema foram verificados através de uma série de testes que incluíam:

1. **Teste de Correção**: Verificação se uma mensagem criptografada e, em seguida, descriptografada retorna à sua forma original.
2. **Teste de Desempenho**: Medição do tempo de execução para criptografia e descriptografia de mensagens de diferentes tamanhos.
3. **Teste de Segurança**: Avaliação da resistência do sistema contra tentativas de fatoração e ataques de força bruta.

Resultados Esperados

Esperava-se que o sistema RSA fosse capaz de criptografar e descriptografar mensagens de forma segura e eficiente, com tempos de execução razoáveis para mensagens de tamanho moderado. Além disso, o sistema deveria demonstrar robustez contra tentativas de ataques.

Ambiente de Desenvolvimento

O ambiente de desenvolvimento para a implementação do RSA foi configurado em uma máquina local com as seguintes especificações:

- Sistema Operacional: Fedora 40
- Processador: Intel Core i7-10710u

- Memória RAM: 64GB
- IDE: CLion 2024.1.2
- Padrão C: C99

Procedimento de Teste

O procedimento de teste envolveu a criptografia e descriptografia de mensagens de diferentes tamanhos para avaliar a correção e o desempenho do algoritmo. As mensagens testadas variaram de pequenas frases a longos textos, incluindo dados binários.

Segurança e Gestão de Chaves

A segurança e gestão de chaves são aspectos críticos na implementação do RSA. Foram adotadas práticas recomendadas para a geração segura de números primos e o armazenamento seguro das chaves privadas. A geração de números primos foi realizada utilizando a biblioteca Openssl, que oferece funções robustas para gerar números primos grandes com segurança. As chaves privadas e os componentes que a geram foram armazenadas em um formato não criptografado devido ao contexto de teste que temos.

Implementação do RSA

A implementação do RSA foi realizada utilizando a linguagem de programação C, devido à sua simplicidade e vasta biblioteca de funções matemáticas.

Ferramentas Utilizadas

- Linguagem de programação C
- Bibliotecas: Openssl

Passos da Implementação

1. Geração de um par de chaves RSA;
2. Salvamento das chaves em arquivos;
3. Salvamento dos componentes das chaves em arquivos;
4. Criptografia e descriptografia de dados;
5. Assinatura e verificação de mensagens.

Estrutura do Programa

O programa é estruturado em várias funções para modularizar a funcionalidade, facilitando a manutenção e a clareza do código.

Definição de bibliotecas e Tratamento de Erros

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include <openssl/err.h>

#define RSA_KEY_BITS 1024

void handleErrors(void) {
    ERR_print_errors_fp(stderr);
    abort();
}
```

- `#include <openssl/rsa.h>`, `#include <openssl/pem.h>` e `#include <openssl/err.h>`: Inclusão das bibliotecas necessárias do OpenSSL para RSA, PEM (formato de arquivo de chave) e tratamento de erros;
- `RSA_KEY_BITS 1024`: Define o tamanho da chave RSA em bits;
- `handleErrors(void)`: Função para lidar com erros do OpenSSL, imprimindo-os e abortando a execução do programa.

Geração do Par de Chaves RSA

```
void generate_keypair(RSA **rsa_keypair) {
    BIGNUM *e = BN_new();
    RSA *rsa = RSA_new();

    if (!e || !rsa) { handleErrors(); }

    BN_set_word(e, RSA_F4);

    if (!RSA_generate_key_ex(rsa, RSA_KEY_BITS, e, NULL)) {
        handleErrors();
    }

    *rsa_keypair = rsa;

    BN_free(e);
}
```

A função generate_keypair é responsável por gerar o par de chaves RSA. Vamos detalhar os passos e os cálculos envolvidos:

1. **Inicialização das Estruturas:**
   - `BIGNUM *e = BN_new()` e `RSA *rsa = RSA_new()`: Cria novas estruturas BIGNUM e RSA;
   - Estas estruturas são essenciais para armazenar os componentes das chaves RSA.

2. **Definição do Expoente Público:**

   - `BN_set_word(e, RSA_F4)`: Define o expoente público como 65537 (`RSA_F4`);
   - O valor 65537 é escolhido porque é um número primo que facilita operações eficientes e seguras na criptografia.

3. **Geração das Chaves:**

   - `RSA_generate_key_ex(rsa, RSA_KEY_BITS, e, NULL)`: Gera o par de chaves RSA com o tamanho definido;
   - Este processo envolve a escolha de dois números primos grandes p e q, e o cálculo dos seguintes componentes:
     - **Módulo** $n = p * q$**:** Produto dos dois primos;
     - **Totiente** $\phi(n) = (p-1) * (q-1)$;
     - **Chave pública** (e, n): Em que e é o expoente público e n é o módulo;
     - **Chave privada** (d, n): Onde d é o inverso multiplicativo de e módulo $\phi(n)$, calculado para satisfazer $d * e \equiv 1 \pmod{\phi(n)}$.

Salvamento das Chaves em Arquivos

```
void save_keys(RSA *rsa_keypair) {
    FILE *pub_file = fopen("public_key.txt", "wb");
    if (!pub_file) { handleErrors(); }
    PEM_write_RSA_PUBKEY(pub_file, rsa_keypair);
    fclose(pub_file);

    FILE *priv_file = fopen("private_key.txt", "wb");
    if (!priv_file) { handleErrors(); }
    PEM_write_RSAPrivateKey(priv_file, rsa_keypair, NULL, NULL, 0, NULL
    fclose(priv_file);
}
```

A função save_keys salva as chaves pública e privada em arquivos no formato PEM, que é um formato de arquivo que facilita a leitura e escrita de chaves.

Salvamento dos Componentes das Chaves

```
void write_bn_to_file(FILE *file, const char *label, const BIGNUM *bn)
    char *dec = BN_bn2dec(bn);
    if (!dec) { handleErrors(); }
    fprintf(file, "%s: %s\n", label, dec);
    OPENSSL_free(dec);
}

void save_key_components(RSA *rsa_keypair) {
    FILE *pub_file = fopen("public_key_components_hex.txt", "w");
    if (!pub_file) { handleErrors(); }

    const BIGNUM *n = RSA_get0_n(rsa_keypair);
    const BIGNUM *e = RSA_get0_e(rsa_keypair);
```

```
    write_bn_to_file(pub_file, "Modulus (n)", n);
    write_bn_to_file(pub_file, "Public Exponent (e)", e);

    fclose(pub_file);

    FILE *priv_file = fopen("private_key_components_hex.txt", "w");
    if (!priv_file) { handleErrors(); }

    const BIGNUM *d = RSA_get0_d(rsa_keypair);
    const BIGNUM *p = RSA_get0_p(rsa_keypair);
    const BIGNUM *q = RSA_get0_q(rsa_keypair);
    const BIGNUM *dmp1 = RSA_get0_dmp1(rsa_keypair);
    const BIGNUM *dmq1 = RSA_get0_dmq1(rsa_keypair);
    const BIGNUM *iqmp = RSA_get0_iqmp(rsa_keypair);

    write_bn_to_file(priv_file, "Modulus (n)", n);
    write_bn_to_file(priv_file, "Public Exponent (e)", e);
    write_bn_to_file(priv_file, "Private Exponent (d)", d);
    write_bn_to_file(priv_file, "Prime 1 (p)", p);
    write_bn_to_file(priv_file, "Prime 2 (q)", q);
    write_bn_to_file(priv_file, "Exponent 1 (dmp1)", dmp1);
    write_bn_to_file(priv_file, "Exponent 2 (dmq1)", dmq1);
    write_bn_to_file(priv_file, "Coefficient (iqmp)", iqmp);

    fclose(priv_file);
}
```

Esta função salva os componentes das chaves em arquivos separados, permitindo uma análise detalhada dos valores individuais.

Criptografia e Descriptografia

```
int main() {
    RSA *rsa_keypair = NULL;
    generate_keypair(&rsa_keypair);
    save_keys(rsa_keypair);
    save_key_components(rsa_keypair);

    const char *plaintext = "hello world!";
    int plaintext_len = strlen(plaintext);

    unsigned char *ciphertext = (unsigned char *)malloc(RSA_size(rsa_ke
    if (!ciphertext) { handleErrors(); }

    int ciphertext_len = RSA_public_encrypt(plaintext_len, (unsigned ch
    if (ciphertext_len == -1) { handleErrors(); }

    unsigned char *decrypted_text = (unsigned char *)malloc(RSA_size(rs
    if (!decrypted_text) { handleErrors(); }
```

```
    int decrypted_len = RSA_private_decrypt(ciphertext_len, ciphertext,
    if (decrypted_len == -1) { handleErrors(); }

    printf("Plaintext: %s\n", plaintext);
    printf("Ciphertext: ");
    for (int i = 0; i < ciphertext_len; i++) {
        printf("%02x", ciphertext[i]);
    }
    printf("\n");
    printf("Decrypted text: %s\n", decrypted_text);

    unsigned char *signature = (unsigned char *)malloc(RSA_size(rsa_key
    if (!signature) { handleErrors(); }

    const char *message = "Hello Bob!";
    int message_len = strlen(message);

    unsigned int signature_len;
    if (!RSA_sign(NID_sha256, (unsigned char *)message, message_len, si
        handleErrors();
    }

    int verified = RSA_verify(NID_sha256, (unsigned char *)message, mes

    (verified != 1) ? printf("Signature verification failed!\n") : prin

    RSA_free(rsa_keypair);
    free(ciphertext);
    free(decrypted_text);
    free(signature);

    return 0;
}
```

A função main realiza as operações de criptografia, descriptografia, assinatura e verificação. Vamos detalhar os cálculos envolvidos na criptografia e descriptografia:

1. Criptografia:
   - **RSA_public_encrypt:** Função que aplica a operação de criptografia usando a chave pública;
   - **Fórmula:** $C = M^e \bmod n$, onde M é a mensagem convertida em um numero, e é o expoente público e n o módulo;
   - Essa operação eleva o numero M à potência e e em seguida aplica o módulo n resultando no texto cifrado C.
2. Descriptografia:
   - **RSA_private_decrypt**: Função que aplica a operação de descriptografia usando a chave privada;
   - **Fórmula:** $M = C^d \bmod n$, em que C é o texto cifrado e d é o expoente privado e n é o módulo;

- Esta operação eleva o texto cifrado C à potência d e em seguida aplica o módulo n, resultando na mensagem original M.

3. Assinatura e Verificação:

   - **RSA_sign**: Função que gera uma assinatura digital usando a chave privada;
   - **RSA_verify**: Função que verifica a assinatura digital usando a chave pública;
   - As funções de assinatura e verificação utilizam o algoritmo de hash SHA-256 para garantir a integridade e autenticidade da mensagem.

Assinatura Digital usando SHA-256 e RSA

A autenticação e a integridade da mensagem são garantidas combinando a função de hash SHA-256 com o algoritmo de criptografia RSA.

*Geração de Assinatura Digital A função de hash SHA-256 (Secure Hash Algorithm 256-bit) é um dos algoritmos mais amplamente utilizados para garantir a integridade e a autenticidade de dados em segurança da informação. Abaixo, explicamos como a função de hash SHA-256 funciona em detalhes.

Uma função de hash criptográfica é um algoritmo que recebe uma entrada (ou mensagem) de qualquer tamanho e produz uma saída fixa (chamada de hash) de tamanho determinado. No caso do SHA-256, a saída é um valor hash de 256 bits (32 bytes).

Alguns motívos de nossa escolha é:

- **Determinística:** A mesma mensagem de entrada sempre produzirá o mesmo hash de saída;
- **Rápida de Computar:** É rápido calcular o hash para qualquer mensagem;
- **Resistente a Pré-Imagem:** Dado um hash H, é difícil encontrar uma mensagem M tal que $\text{hash}(M) = H$;
- **Resistente a Segunda Pré-Imagem:** Dada uma mensagem M_1, é difícil encontrar uma mensagem diferente M_2 que tenha $\text{hash}(M_1) = \text{hash}(M_2)$;
- **Resistente a Colisão:** É difícil encontrar duas mensagens diferentes M_1 e M_2 que tenham o mesmo hash.

*Etapas do SHA-256 A função SHA-256 processa a mensagem de entrada em várias etapas para produzir o valor hash final. Aqui estão os passos principais:

1. **Preprocessamento:**

   - **Padding:** A mensagem de entrada é preenchida (ou "padded") para que seu comprimento em bits seja congruente a 448 (mod 512). Isso significa que a mensagem será preenchida para que o comprimento seja 64 bits a menos que um múltiplo de 512;
     - Adiciona-se um bit '1' ao final da mensagem;
     - Adicionam-se bits '0' até que o comprimento seja 448 (mod 512).
   - **Comprimento:** O comprimento original da mensagem é anexado ao final da mensagem preenchida, representado como um inteiro de 64 bits.

2. **Inicialização das Variáveis de Hash:**

   - O SHA-256 utiliza oito variáveis de hash inicializadas com valores constantes específicos. Estas variáveis são:

```
h0 = 0x6a09e667
h1 = 0xbb67ae85
h2 = 0x3c6ef372
h3 = 0xa54ff53a
h4 = 0x510e527f
h5 = 0x9b05688c
h6 = 0x1f83d9ab
h7 = 0x5be0cd19
```

3. **Processamento em Blocos:**

   - A mensagem preenchida é dividida em blocos de 512 bits (64 bytes) para processamento;
   - Cada bloco é então processado em uma série de 64 rodadas utilizando operações lógicas e aritméticas. As operações principais incluem:
     - **Funções Booleanas:** Ch, Maj, $\sum 0$, $\sum 1$, $\sigma 0$, $\sigma 1$;
     - **Constantes de Rodada:** SHA-256 usa uma série de constantes específicas para cada uma das 64 rodadas.

4. **Transformação de Compressão:**

   - Para cada bloco de 512 bits:
     - Expande-se o bloco em uma sequência de 64 palavras de 32 bits;
     - Inicializa-se oito variáveis de trabalho com os valores atuais das variáveis de hash;
     - Realiza-se uma série de 64 rodadas de operações de mistura que incluem adições, rotações e operações lógicas. Cada rodada usa uma constante específica e uma parte do bloco expandido;
     - Atualiza-se as variáveis de hash com os valores das variáveis de trabalho.

5. Combinação dos Resultados:

   - Após processar todos os blocos, os valores finais das variáveis de hash são concatenados para produzir o hash final de 256 bits.

*Exemplo Simplificado Considere a mensagem "abc"para o exemplo simplificado a seguir:

1. **Conversão para Binário:**

   - "abc"em ASCII:
     - 'a' = 97 = 01100001;
     - 'b' = 98 = 01100010;
     - 'c' = 99 = 01100011;
     - "abc"= 01100001 01100010 01100011.

2. **Padding:**

   - Adicione '1':
     - "01100001 01100010 01100011 1";
   - Inicialmente, a mensagem tem 24 bits. Precisamos adicionar 424 bits de padding $448 - 24$ para totalizar 448 bits:
     - 01100001 01100010 01100011 1 00000000 00000000 ... (424 bits de '0')

- Representamos o comprimento da mensagem original (24 bits) como um número binário de 64 bits e adicionamos ao final:
  - 01100001 01100010 01100011 1 00000000 ... 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00011000

3. **Inicialização e Processamento:** Agora que a mensagem tem exatamente 512 bits (64 bytes), o que é necessário para SHA-256, podemos realizar o processamento.

   - Inicialize variáveis de hash (os oito valores citados acima);
   - Divida a mensagem preenchida em blocos de 512 bits (já temos uma mensagem com esse tamanho);
   - Realizam-se as 64 rodadas de operações de mistura para cada bloco, como o exemplo a seguir:

```
for i from 0 to 63:
    S1 := (e rightrotate 6) ^ (e rightrotate 11) ^ (e rightrotate 25)
    ch := (e and f) ^ ((not e) and g)
    temp1 := h + S1 + ch + k[i] + w[i]
    S0 := (a rightrotate 2) ^ (a rightrotate 13) ^ (a rightrotate 22)
    maj := (a and b) ^ (a and c) ^ (b and c)
    temp2 := S0 + maj

h := g
g := f
f := e
e := d + temp1
d := c
c := b
b := a
a := temp1 + temp2
```

4. **Combinação dos Resultados:**

   - Após processar o bloco de 512 bits, atualizam se as variáveis de hash:

```
h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d
h4 := h4 + e
h5 := h5 + f
h6 := h6 + g
h7 := h7 + h
```

5. **Hash Final:**

   - Finalmente, os valores das variáveis de hash (h0, h1, h2, h3, h4, h5, h6, h7) são concatenados para formar o hash final de 256 bits. Cada variável de hash é de 32 bits, totalizando 256 bits. O resultado é o hash SHA-256 da mensagem "abc".

Criptografia Híbrida: Integração de AES com RSA

A criptografia híbrida combina os pontos fortes dos algoritmos de criptografia assimétrica (RSA) e simétrica (AES) para obter maior eficiência e segurança. A seguir, apresentamos a implementação dessa integração utilizando a linguagem C e as bibliotecas OpenSSL.

Implementação da Criptografia Híbrida

A implementação do AES e RSA em conjunto permite a criptografia de mensagens maiores com menor perda de desempenho. A seguir, apresentamos a estrutura do código C que implementa essa abordagem complementando o código RSA anterior.

```
int aes_encrypt(
    const unsigned char *plaintext,
    int plaintext_len, unsigned char *ciphertext,
    unsigned char *iv,
    const unsigned char *key
    ) {
    EVP_CIPHER_CTX *ctx;
    int len;
    int ciphertext_len;

    if (!(ctx = EVP_CIPHER_CTX_new())) handleErrors();
    if (1 != EVP_EncryptInit_ex(
        ctx, EVP_aes_256_cbc(), NULL, key, iv)
        ) handleErrors();
    if (1 != EVP_EncryptUpdate(
        ctx, ciphertext, &len, plaintext, plaintext_len
        )) handleErrors();
    ciphertext_len = len;
    if (1 != EVP_EncryptFinal_ex(
        ctx, ciphertext + len, &len
        )) handleErrors();
    ciphertext_len += len;
    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_len;
}

int aes_decrypt(
    const unsigned char *ciphertext,
    int ciphertext_len,
    unsigned char *plaintext,
    const unsigned char *iv,
    const unsigned char *key
    ) {
    EVP_CIPHER_CTX *ctx;
    int len;
    int plaintext_len;

    if (!(ctx = EVP_CIPHER_CTX_new())) handleErrors();
```

```
    if (1 != EVP_DecryptInit_ex(
        ctx, EVP_aes_256_cbc(), NULL, key, iv
        )) handleErrors();
    if (1 != EVP_DecryptUpdate(
        ctx, plaintext, &len, ciphertext, ciphertext_len
        )) handleErrors();
    plaintext_len = len;
    if (1 != EVP_DecryptFinal_ex(
        ctx, plaintext + len, &len
        )) handleErrors();
    plaintext_len += len;
    EVP_CIPHER_CTX_free(ctx);

    return plaintext_len;
}
```

Com a junção de ambos códigos, utilizamos o RSA para criptografar uma chave AES gerada aleatoriamente. A mensagem original é então criptografada usando a chave AES. Essa abordagem oferece várias vantagens, especialmente quanto à performance.

Análise

A análise dos resultados focou na comparação entre os resultados obtidos e os esperados, destacando os desafios encontrados durante o processo.

Correção dos Resultados

Os testes de correção mostraram que o sistema RSA implementado conseguiu criptografar e descriptografar mensagens corretamente, confirmando a precisão da implementação. As mensagens criptografadas foram recuperadas integralmente após a descriptografia, demonstrando que o algoritmo foi implementado corretamente.

Desempenho

Os testes de desempenho revelaram que, embora o RSA seja eficiente para mensagens curtas, o tempo de execução aumenta significativamente para mensagens maiores. Isso se deve à complexidade da aritmética modular com números grandes. A criptografia e descriptografia de mensagens longas resultaram em tempos de execução elevados, o que pode ser um fator limitante em aplicações práticas que exigem processamento em tempo real. No entanto, ao utilizar AES (Advanced Encryption Standard), é possível criptografar mensagens maiores com menor perda de desempenho. O AES é otimizado para grandes volumes de dados e oferece uma combinação de segurança e eficiência, tornando-se uma escolha ideal para cenários que demandam alta performance e manuseio de grandes quantidades de informação.

Segurança

Os testes de segurança indicaram que o sistema RSA é robusto contra ataques de fatoração, desde que os números primos escolhidos sejam suficientemente grandes. No entanto, foi identificado que a geração de chaves pode ser um ponto fraco se não for realizada com cuidado, pois a escolha inadequada de primos pode comprometer a segurança. A análise de ataques de força bruta mostrou que, com números primos grandes, o RSA oferece uma segurança sólida contra tais ataques.

Desafios e Problemas

Durante a implementação, foram encontradas dificuldades relacionadas à eficiência da geração de números primos grandes e à execução de operações aritméticas com números extremamente grandes. Além disso, a gestão de chaves revelou-se um desafio, destacando a necessidade de um armazenamento seguro e eficiente das chaves privadas. A geração de chaves RSA é um processo computacionalmente intensivo, e a escolha de números primos inadequados pode resultar em uma chave vulnerável.

Além desses desafios, o tamanho da mensagem também se mostrou um problema significativo. O RSA é eficiente para criptografar dados de tamanho pequeno, mas seu desempenho cai drasticamente à medida que o tamanho da mensagem aumenta. Isso ocorre devido à complexidade das operações aritméticas envolvidas, que são exponencialmente mais lentas para mensagens grandes. Para contornar essa limitação, a implementação de uma abordagem híbrida usando AES (Advanced Encryption Standard) foi essencial.

A criptografia RSA, embora altamente segura, não é adequada para mensagens grandes devido ao seu limite de tamanho de bloco, que é diretamente relacionado ao tamanho da chave. Por exemplo, com uma chave RSA de 1024 bits, o tamanho máximo de dados que podem ser criptografados diretamente é de aproximadamente 117 bytes, após descontar o espaço necessário para o padding. Isso torna a criptografia de mensagens maiores um desafio significativo.

Para resolver essa limitação, foi necessário adotar a criptografia híbrida, combinando RSA e AES. Com essa abordagem, a mensagem é criptografada usando AES, que é um algoritmo de criptografia simétrica eficiente para grandes volumes de dados e em seguida, a chave AES gerada aleatoriamente é criptografada usando RSA. Esse método proporciona grandes benefícios relacionados à performance.

Ao implementar a criptografia híbrida, conseguimos superar as limitações do RSA para mensagens grandes, aproveitando a eficiência do AES. Esta abordagem não só melhora o desempenho, mas também mantém um alto nível de segurança. A combinação de RSA e AES é ideal para aplicações que necessitam de criptografia robusta e eficiente para grandes volumes de dados.

Análise Comparativa

Foi realizada uma análise comparativa entre o RSA e outros algoritmos de criptografia assimétrica, como o ElGamal e o ECC. A análise mostrou que o RSA oferece uma boa combinação de segurança e facilidade de implementação, mas enfrenta desafios de desempenho em comparação com algoritmos baseados em curvas elípticas (ECC), que podem oferecer uma segurança equivalente com chaves menores e operações mais rápidas.

Aplicações Práticas

O RSA tem uma ampla gama de aplicações práticas, desde a segurança de comunicação em e-mails e navegadores da web até a assinatura digital de documentos e transações financeiras. No entanto, a aplicação do RSA em dispositivos com recursos limitados, como dispositivos móveis e IoT, pode ser desafiadora devido aos requisitos computacionais elevados.

Melhorias Propostas

Para melhorar o desempenho do RSA, podem ser consideradas otimizações como a Exponenciação Modular Rápida e algoritmos eficientes de geração de primos. Além disso, a combinação do RSA com outros métodos de criptografia pode oferecer um equilíbrio entre segurança e eficiência. A implementação de hardware específico para operações aritméticas pode também melhorar o desempenho em aplicações críticas.

Conclusão

O sistema criptográfico RSA, embora complexo, é uma ferramenta poderosa para a criptografia de dados. A implementação prática do RSA mostrou-se eficaz na proteção de informações, confirmando a teoria subjacente. No entanto, desafios relacionados ao desempenho e à gestão de chaves foram identificados.

Para superar esses desafios, a integração do AES (Advanced Encryption Standard) como parte de uma abordagem híbrida se mostrou essencial. O AES permite a criptografia de grandes volumes de dados com alta eficiência, enquanto o RSA é utilizado para a criptografia segura da chave AES. Essa combinação aproveita as vantagens de ambos os algoritmos, resultando em uma solução criptográfica que é ao mesmo tempo segura e eficiente. A abordagem híbrida não apenas melhora o desempenho, mas também facilita a gestão de chaves, tornando-a mais adequada para aplicações práticas que exigem processamento rápido e seguro de grandes volumes de dados.

Comentários sobre os Modelos Utilizados

O modelo teórico do RSA é sólido, baseado em princípios matemáticos bem estabelecidos. A implementação prática demonstrou a viabilidade do RSA para aplicações reais, embora a eficiência possa ser um problema para grandes volumes de dados. A escolha cuidadosa de números primos e a geração segura de chaves são cruciais para a eficácia do RSA.

A adição do AES (Advanced Encryption Standard) ao modelo de criptografia híbrida melhora significativamente a eficiência para grandes volumes de dados. O AES, sendo um algoritmo de criptografia simétrica, é muito mais rápido que o RSA para processar grandes quantidades de informações. Ao utilizar o RSA para criptografar apenas a chave AES e o AES para criptografar os dados, conseguimos um equilíbrio entre segurança e desempenho. Essa abordagem híbrida permite aproveitar a robustez do RSA para troca segura de chaves e a velocidade do AES para criptografia de dados volumosos, tornando a solução mais prática e eficaz para aplicações que exigem processamento rápido e seguro de grandes volumes de dados.

Possíveis Aplicações

O RSA pode ser utilizado em diversas áreas, incluindo comunicação segura, assinatura digital e proteção de dados sensíveis. Sua capacidade de fornecer segurança robusta torna-o adequado para aplicações em setores financeiros, governamentais e de TI. No entanto, a aplicação do RSA em ambientes com restrições de recursos requer atenção especial aos requisitos computacionais.

Indicação de Aperfeiçoamento

Para melhorar o desempenho do RSA, pode-se considerar o uso de otimizações como a Exponenciação Modular Rápida e algoritmos eficientes de geração de primos. Além disso, a combinação do RSA com outros métodos de criptografia pode oferecer um equilíbrio entre segurança e eficiência. A implementação de hardware específico para operações aritméticas pode também melhorar o desempenho em aplicações críticas. A adoção de práticas de gestão de chaves seguras é essencial para garantir a integridade e a confidencialidade dos dados.

Referências

Referências

[1] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2), 120-126.

[2] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press.

[3] Trappe, W., & Washington, L. C. (2006). *Introduction to Cryptography with Coding Theory*. Pearson.

[4] Stallings, W. (2016). *Cryptography and Network Security: Principles and Practice*. Pearson.

[5] Schneier, B. (1996). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons.

[6] OpenSSL Project. (n.d.). OpenSSL Crypto Library. Retrieved from `https://www.openssl.org/docs/man3.0/man7/crypto.html`

[7] Wikipedia. (n.d.). RSA (cryptosystem). Retrieved from `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`

[8] Wikipedia. (n.d.). Advanced Encryption Standard. Retrieved from `https://pt.wikipedia.org/wiki/Advanced_Encryption_Standard`

[9] Opensource.com. (2019). Cryptography basics with OpenSSL, part 1. Retrieved from `https://opensource.com/article/19/6/cryptography-basics-openssl-part-1`

[10] Red Hat. (n.d.). Using OpenSSL. Retrieved from `https://access.redhat.com/documentation/pt-br/red_hat_enterprise_linux/7/html/security_guide/sec-using_openssl`

[11] Linux Journal. (2004). An Introduction to OpenSSL Programming. Retrieved from `https://www.linuxjournal.com/article/6826`

[12] Rajesh, B. N. (2019). Understanding RSA Public Key. Medium. Retrieved from `https://medium.com/@bn121rajesh/understanding-rsa-public-key-70d900b1033c`