Relatorio: Gerador e Verificador de Assinatura RSA

Igor Bastos de Oliveira, 18/0122177

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)

1. Introdução

O RSA (Rivest-Shamir-Adleman), como um dos pioneiros nas tecnologias assíncronas de criptografia de chave pública, destaca-se como uma ferramenta altamente eficiente na segurança computacional digital [2]. Ampla e frequentemente utilizada para garantir a transmissão segura de dados em diversas aplicações, como transações online e envio de e-mails, o RSA se destaca por sua notável robustez, sendo considerado um dos métodos mais seguros para a criptografia de dados.

A característica essencial do RSA fundamenta-se em sua abordagem assimétrica, onde a chave de encriptação pública é claramente distinta da chave de decriptação privada. Essa assimetria é estabelecida com base na complexidade prática associada à fatorização do produto de dois números primos grandes. Essa complexidade matemática constitui a espinha dorsal da segurança do RSA, uma vez que a fatorização eficiente de grandes números primos é uma tarefa desafiadora e intensiva computacionalmente.[1]

Além disso, o RSA representa uma das primeiras inovações a possibilitar tanto a criptografia quanto a assinatura digital. O algoritmo opera por meio da criação e divulgação de uma chave pública composta por dois números primos significativos, acompanhados de um valor auxiliar. A manutenção da confidencialidade dos números primos subjacentes é crucial para a eficácia do sistema.

Ao empregar a chave pública, qualquer pessoa pode criptografar uma mensagem, mas a decodificação bem-sucedida exige um profundo entendimento dos números primos envolvidos. Esse desafio, conhecido como o ”problema RSA”, é a essência da resistência do RSA contra tentativas de quebra.[3]

Neste relatório, abordaremos a criação de um programa de criptografia e descriptografia utilizando o RSA, explorando assim as potencialidades e desafios associados a essa renomada técnica de segurança.

2. RSA Implementação

Nesta seção, discutiremos a relevância e a função do código apresentado, que se concentra na implementação do algoritmo RSA (Rivest-Shamir-Adleman) para geração de chaves, criptografia e descriptografia de mensagens.

2.1. RSAKeyGenerator

A classe RSAKeyGenerator é central para a implementação do algoritmo RSA, desempenhando um papel crucial na geração das chaves pública e privada, bem como na facilitação dos processos de criptografia e descriptografia. Abaixo estão detalhes específicos sobre os principais métodos e atributos desta classe:

```
def __init__(self, key_size=1024):
self.key_size = key_size
```

```
        self.p = self.q = self.n = self.fn = self.e = self.d = None
        self.generate_keys()
```

- O método de inicialização define o tamanho padrão da chave como 1024 bits, mas permite a personalização desse tamanho.
- Atributos importantes, como p, q, n, fn, e, e d, são inicializados como None.
- Automaticamente invoca o método generateKeys() para gerar as chaves durante a criação de uma instância da classe.

```
    def miller_rabin(self, n, k):
        if n in {2, 3}:
            return True

        if n % 2 == 0:
            return False

        r, s = 0, n - 1
        while s % 2 == 0:
            r += 1
            s //= 2

        for _ in range(k):
            a = random.randint(2, n - 2)
            x = pow(a, s, n)

            if x in {1, n - 1}:
                continue

            for _ in range(r - 1):
                x = pow(x, 2, n)
                if x == n - 1:
                    break
            else:
                return False

        return True
```

- Implementação do teste probabilístico de primalidade de Miller-Rabin, crucial para verificar se um número é primo.
- Utiliza parâmetros n e k, representando o número a ser testado e o número de iterações do teste, respectivamente.
- A lógica envolve manipulação eficiente de números primos e aleatoriedade para garantir a precisão do teste.

```
    def extended_gcd(self, a, m):
        if math.gcd(a, m) != 1:
            return None
```

```
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3
        v1, v2, v3, u1, u2, u3 = (
            u1 - q * v1,
            u2 - q * v2,
            u3 - q * v3,
            v1,
            v2,
            v3,
        )
    return u1 % m
```

- Implementa o algoritmo de Euclides Estendido para calcular o inverso multiplicativo modular de a módulo m.
- Garante que a e m sejam primos entre si (coprimos), um requisito fundamental para a determinação da chave privada.

```
        def generate_keys(self):
    self.generate_prime_pair()
    self.calculate_n_and_fn()
    self.calculate_public_key()
    self.calculate_private_key()

def generate_prime_pair(self):
    while True:
        try:
            self.p = self.generate_primes()
            self.q = self.generate_primes()
            break
        except RecursionError:
            pass

def calculate_n_and_fn(self):
    self.n = self.p * self.q
    self.fn = (self.p - 1) * (self.q - 1)

def calculate_public_key(self):
    self.e = 0
    while math.gcd(self.fn, self.e) != 1:
        self.e = random.randrange(2, self.fn)

def calculate_private_key(self):
    self.d = self.extended_gcd(self.e, self.fn)
```

- Coordena a geração completa das chaves, invocando métodos específicos para gerar o par de primos, calcular n e fn, bem como determinar as chaves pública e privada.

- Gera um par de primos utilizando o método generateprimes, evitando recursões infinitas e garantindo a obtenção de primos distintos.
- Calcula os valores de n e fn com base nos primos gerados, essenciais para a construção das chaves.
- Determina a chave pública (e) garantindo que seja coprima em relação a fn.
- Utiliza o algoritmo de Euclides Estendido (extendedgcd) para calcular a chave privada (d).

```
            def rsa_encrypt(self, message, public_key):
    e, n = public_key
    cipher_text = [pow(ord(char), e, n) for char in message]
    return cipher_text

def rsa_decrypt(self, cipher_text, private_key):
    d, n = private_key
    decrypted_text = ''.join([chr(pow(char, d, n)) for char in ciph
    return decrypted_text
```

O RSA encrypt é responsável por criptografar uma mensagem utilizando a chave pública do algoritmo RSA. A lógica por trás dessa criptografia é baseada no uso da exponenciação modular para garantir eficiência computacional. pontualmente, ele faz:

- O método descompacta os valores da chave pública (e e n). Itera sobre cada caractere da mensagem, aplicando a função ord(char) para obter o valor numérico correspondente ao caractere na tabela ASCII.
- Em seguida, utiliza a função pow para realizar a exponenciação modular, onde a base é o valor numérico do caractere, o expoente é e (componente da chave pública), e o módulo é n (outro componente da chave pública).
- O resultado dessa operação é uma lista de valores criptografados, que representa a mensagem criptografada..
- Determina a chave pública (e) garantindo que seja coprima em relação a fn.
- Utiliza o algoritmo de Euclides Estendido (extendedgcd) para calcular a chave privada (d).

Ja o RSA descrypt tem como função descriptografar uma mensagem criptografada utilizando a chave privada do algoritmo RSA. Da mesma forma que a etapa de criptografia, utiliza a exponenciação modular para garantir eficiência durante o processo.

- O método descompacta os valores da chave privada (d e n).
- Itera sobre cada valor na lista ciphertext. (outro componente da chave pública).
- Aplica a função pow para realizar a exponenciação modular inversa, onde a base é o valor criptografado, o expoente é d (componente da chave privada), e o módulo é n (outro componente da chave privada).
- O resultado dessa operação é uma lista de valores numéricos, que são convertidos de volta para caracteres usando a função chr. A concatenação desses caracteres resulta na mensagem original descriptografada.

Essas são algumas classes importantes para a geração de chave publica e privada, alem delas existem classes auxiliares para geração de numeros aleatorios, todas são fundamentais para a criptografia e descriptografia RSA

3. Assinatura Digital com RSA e Esquema OAEP

esta seção, exploraremos a implementação de assinaturas digitais utilizando o algoritmo RSA em conjunto com o esquema OAEP (Optimal Asymmetric Encryption Padding). Essa abordagem visa garantir a autenticidade e integridade das mensagens, fornecendo uma camada adicional de segurança. Abaixo estão os principais pontos relacionados à classe RSASignatureGenerator:

3.1. Modulos de importação

Para o desenvolvimento da assinatura digital, foi necessario a utilização de alguns modulos necessarios para auxiliar na construção do algoritmo, entre eles temos:

- hashlib: utilizado para implementar funções hash seguras, sendo fundamental para calcular o hash SHA3-224 da mensagem. O SHA3-224 é escolhido devido à sua resistência comprovada a colisões e propriedades criptográficas sólidas
- Os: acessar funcionalidades relacionadas ao sistema operacional. Em particular, a função os.urandom() é usada para gerar bytes aleatórios, essenciais para o esquema OAEP, onde aleatoriedade é crucial para a
- base64: È utilizado para codificar e decodificar dados em formato base64. Aqui, ele desempenha um papel importante na representação compacta e segura dos hashes e das assinaturas, facilitando a integração com sistemas que esperam dados em formato de texto.
- Math: operações matemáticas específicas.
- RSAKeyGenerator: Realça a necessidade de um sistema RSA completamente funcional para operações de criptografia e descriptografia

3.2. Métodos da Classe

```
    def sha3_224(self, m):
        sha3 = hashlib.sha3_224()
        sha3.update(m)
        return sha3.digest()

def mgf1(self, seed, mlen):
    t = b''
    hlen = 28

    for c in range(ceil(mlen / hlen)):
        c_bytes = c.to_bytes(4, byteorder='big')
        t += self.sha3_224(seed + c_bytes)

    return t[:mlen]
```

- sha3224(self, m): Este método recebe uma mensagem m e calcula o hash SHA3-224 usando o módulo hashlib. O resultado é um resumo criptográfico de 224 bits da mensagem. Ele é essencial para a obtenção do hash da mensagem, que é uma etapa crítica no processo de assinatura digital.
- mgf1(self, seed, mlen): Implementa a função de geração de máscara (MGF1) Parte integrante do processo de codificação OAEP, fornecendo máscaras seguras para o padding.

```
    def codifica_oaep(self, m, k, label=b''):
mlen = len(m)
lhash = self.sha3_224(label)
hlen = len(lhash)

ps_len = k - mlen - 2 * hlen - 2
ps = b'\x00' * ps_len

db = lhash + ps + b'\x01' + m

seed = os.urandom(hlen)
db_mask = self.mgf1(seed, k - hlen - 1)
masked_db = self.xor_bloco(db, db_mask)

seed_mask = self.mgf1(masked_db, hlen)
masked_seed = self.xor_bloco(seed, seed_mask)

return b'\x00' + masked_seed + masked_db
```

- codificaoaep(self, m, k, label=b”): Realiza a codificação OAEP na mensagem m com padding e máscaras pseudoaleatórias para melhorar a segurança durante a cifragem. Prepara a mensagem para cifragem, incorporando o esquema OAEP.

```
      def xor_bloco(self, a, b):
return bytes(x ^ y for x, y in zip(a, b))
return b'\x00' + masked_seed + masked_db
```

- xorBloco(self, a, b)(self, m, k, label=b”): Realiza a operação de XOR elemento a elemento entre dois blocos de bytes a e b. Utilizado internamente para combinar blocos de dados de maneira segura, sendo uma operação fundamental na implementação de esquemas criptográficos.

```
def cifra(self, mensagem, chave_publica):
e, n = chave_publica
return pow(mensagem, e, n)
```

- cifra: Realiza a cifragem de uma mensagem numérica usando a chave pública do RSA.

```
 def assina_mensagem(self, mensagem):
mensagem_bytes = mensagem.encode('utf-8')
resultado_sha = self.sha3_224(mensagem_bytes)

mensagem_hash = base64.encodebytes(resultado_sha).strip()
k = (self.public_key[1].bit_length() + 7) // 8

cifrado_oaep = self.codifica_oaep(mensagem_hash, k, self.label)
c = self.cifra_raw(cifrado_oaep, self.private_key)

return base64.encodebytes(c).strip()
```

- assinaMensagem: Recebe uma mensagem, gera seu hash, codifica-a com OAEP, cifra o resultado e retorna a assinatura final como bytes codificados em base64. Principal método para gerar assinaturas digitais, integrando todas as etapas do processo.

3.3. Testes

Quando o programa é inicializado, a primeira parte sera o menu inicial que vai mostrar todas as opções para o usuario

Figure 1. Menu inicial

Ele tem tres opções de funções, sendo a terceira sair do programa, ao apertar a primeira o programa vai gerar automaticamente a chave publica e privada e vai pedir ao usuario digitar algo a ser criptografado

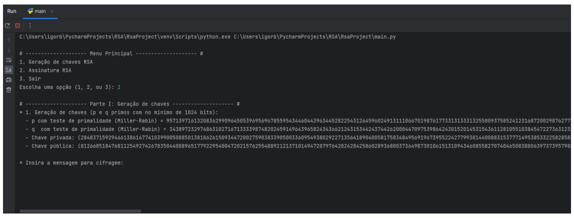

Figure 2. Ao apertar a opção 1

Figure 3. Mensagem e o processo de criptografia

Ao apertar a opção 2, o programa vai pedir ao usuario uma mensagem para o processo de criptografia e assinatura digital

Figure 4. Opção de assinatura digital

O programa vai ficar em loop ate o usuario apertar o 3 e finalizar.

4. Conclusão

O presente trabalho abordou detalhadamente a implementação de assinaturas digitais utilizando o algoritmo RSA em conjunto com o esquema OAEP (Optimal Asymmetric Encryption Padding). Ao longo da exploração das classes RSAKeyGenerator e RSASignatureGenerator, foi possível compreender os processos de geração de chaves, criptografia e descriptografia no contexto do RSA, bem como a aplicação do esquema OAEP para garantir a segurança e integridade das assinaturas digitais.

A classe RSAKeyGenerator destacou-se como uma ferramenta robusta para a criação de chaves pública e privada, fundamentais para a implementação segura do algoritmo RSA. A escolha criteriosa de métodos, como o teste de primalidade de Miller-Rabin e o algoritmo de Euclides Estendido, demonstrou um compromisso claro com a segurança matemática subjacente ao RSA.

A classe RSASignatureGenerator ampliou esse cenário ao incorporar o esquema OAEP, proporcionando uma camada adicional de segurança na assinatura digital de mensagens. A utilização de funções hash seguras, geração de máscaras pseudoaleatórias e operações XOR contribuíram para a robustez do esquema OAEP, resguardando contra potenciais vulnerabilidades.

Em conclusão, a implementação apresentada destaca-se não apenas pela aplicação efetiva do algoritmo RSA e do esquema OAEP. Este trabalho proporciona uma base sólida para compreensão e implementação de sistemas de segurança digital, contribuindo significativamente para o entendimento do papel do RSA na segurança da informação.

References

[1] BLAKLEY, G. R., AND BOROSH, I. Rivest-shamir-adleman public key cryptosystems do not always conceal messages. *Computers & mathematics with applications 5*, 3 (1979), 169–178.

[2] KUMAR, M. Advanced rsa cryptographic algorithm for improving data security. In *Cyber Security: Proceedings of CSI 2015* (2018), Springer, pp. 11–15.

[3] SIHOTANG, H. T., EFENDI, S., ZAMZAMI, E. M., AND MAWENGKANG, H. Design and implementation of rivest shamir adleman's (rsa) cryptography algorithm in text file data security. In *Journal of Physics: Conference Series* (2020), vol. 1641, IOP Publishing, p. 012042.