Explicacão do Código Python para Criptografia RSA com OAEP e Base64

Davi de Araújo Garcez Bueno - 211060586
Erick Hideki Taira - 222011525

Janeiro 2025

1 Introdução

Os códigos apresentados implementam um sistema de criptografia baseado no algoritmo RSA, combinando técnicas como o OAEP (Optimal Asymmetric Encryption Padding), codificacão em Base64 e teste de primalidade de Miller-Rabin para identificar números primos. Além disso, ele realiza assinaturas digitais utilizando um esquema que utiliza o RSA e também é possível verifica a autenticidade da assinatura.

2 Introdução Teórica

O algoritmo RSA é um sistema de criptografia assimétrica que utiliza chaves públicas e privadas para a criptografia e descriptografia de mensagens. A segurança do RSA é baseada na dificuldade de fatorar o produto de dois números primos grandes. Porém, o algoritmo RSA, sozinho, para uma mesma mensagem, retorna sempre o mesmo texto criptografado. Isso torna o algoritmo suscetível a ataques de análise de frequência, onde o atacante pode identificar padrões e inferir informações sobre a mensagem original.

Para mitigar essa vulnerabilidade, técnicas adicionais como o OAEP (Optimal Asymmetric Encryption Padding) são utilizadas. O OAEP adiciona um preenchimento aleatório à mensagem antes da criptografia, garantindo que textos iguais resultem em criptogramas diferentes. Além disso, a codificação Base64 é empregada para converter dados binários em uma representação textual, facilitando a manipulação e transmissão segura das mensagens.

Outro componente essencial do sistema é o teste de primalidade de Miller-Rabin, utilizado para garantir que os números gerados para as chaves sejam realmente primos. A assinatura digital com RSA também é implementada para assegurar a autenticidade e integridade das mensagens, permitindo que o receptor verifique se a mensagem foi enviada pelo remetente legítimo e não foi alterada durante a transmissão.

3 Fluxo Geral da Implementação

1. Geração de dois números primos p e q.
    - Miller Rabin
        - Decompondo
    - Gerar Primos de 1024 bits
2. Cálculo de n e de phi.
3. Definição da chave pública e cálculo da chave privada.
4. Codificação da mensagem em Base64, seguida do algoritmo OAEP.
5. Criptografia da mensagem com RSA.
6. Descriptografia da mensagem e reversão do processo de OAEP e Base64.
7. Geração e verificação de uma assinatura digital.
8. Teste de execução do programa.
9. Impressão dos resultados.

4 Geração dos números primos

Função `decompondo(n)`

Esta funcão decompõe um número no seguinte formato. Ela será utilizada em uma das etapas do algoritmo de Miller Rabin.

$$n = 2^e \cdot m$$

```
def decompondo(n):
    # Decompoe n-1 como 2 elevado a e vezes m, onde n e impar.
    # Args:
    #    n: numero para decompor
    # Returns:
    #    int: expoente
    #    int: numero
        e = 0
        m = n - 1
        while m % 2 == 0:
            m //= 2
            e += 1
        return e, m
```

4.1 Função `miller_rabin`

Implementa o teste de primalidade de Miller-Rabin, retornando verdadeiro se o número for provavelmente primo e falso caso contrário.

```
    def miller_rabin(n, rodadas=10):
    """
    Miller-Rabin primality test
    Args:
        n: numero para ser testado
        rodadas: quantidade de rodadas a serem feitas
    Returns:
        bool: True se e primo, False se nao e primo
    """
        if n <= 1 or n == 4:
            return False
        if n <= 3:
            return True
        if n % 2 == 0:
            return False

        expo_k, m = decompondo(n)  #exponte K e m tal que n - 1 =
    2^k * m
        for _ in range(rodadas):
            a = random.randrange(2, n - 2) # pega um numero
    aleatorio entre 2 e n-2
            """ x = a^m mod n """
            x = pow(a, m, n)
            if x == 1 or x == n-1: # se x for 1 ou n-1, entao n e
    primo
                continue
            else:
                teste = False
                for _ in range(expo_k - 1): # para i de 0 ate k-1
                    """ x = x^2 mod n """
                    x = pow(x, 2, n)
                    if x == n-1:
                        teste = True
                        break
                if(teste):
                    continue
                else:
                    return False
        return True

```

Teste de primalidade de Miller-Rabin

Teoria do Teste de Primalidade de Miller-Rabin

O teste de primalidade de Miller-Rabin é um algoritmo probabilístico utilizado para determinar se um número é primo. Ele é baseado no pequeno teorema de Fermat, que afirma que se p é um número primo e a é um inteiro tal que $1 \leq a < p$, então:

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

O teste de Miller-Rabin estende essa ideia, verificando se um número composto pode ser identificado por meio de várias iterações de um teste baseado em exponenciação modular. O algoritmo funciona da seguinte forma:
1. **Decompor**: Decompõe $n-1$ na forma $2^k \cdot m$, onde m é ímpar.
2. **Escolha de Testes**: Escolhe um número aleatório a tal que $2 \leq a \leq n-2$.
3. **Exponenciação Modular**: Calcula $x = a^m \mod n$.
4. **Verificação Inicial**: Se $x = 1$ ou $x = n-1$, então n pode ser primo.
5. **Iterações**: Para $k-1$ iterações, calcula $x = x^2 \mod n$. Se em qualquer iteração $x = n-1$, então n pode ser primo.
6. **Conclusão**: Se nenhuma das condições acima for satisfeita, n é composto.

O teste é repetido várias vezes com diferentes valores de a para aumentar a confiança no resultado. Se o número passar em todas as iterações, ele é considerado provavelmente primo.

4.2 Função `prime_numbers`

Gera dois números primos utilizando o teste de Miller-Rabin.

```
    def prime_numbers(): #descobre os numeros primos usando a
    funcao de Miller-Rabin
    """
    Descobrindo os numeros primos
    Args: none
    Returns:
        list: lista com os numeros primos
    """
        n = 1 << 1024
        contador = 0
        par_de_numeros_primos = []
        while True:
            if miller_rabin(n):
                par_de_numeros_primos.append(n)
                contador += 1
            if contador == 2:
                break
            n += 1
        return par_de_numeros_primos

```

É uma maneira bem rudimentar de descobrir o par de números primos com mais de 1024 bits, mas é suficiente para a nossa aplicação. Porém é importante ressaltar que para aplicações reais, é necessário um método mais robusto para a geração dos números primos, pois no nosso caso sempre será gerado os mesmos números primos.

5 Implementação do OAEP

5.1 Função `mgf1`

Mascara uma entrada utilizando uma funcão de hash para gerar uma saida pseudoaleatória de comprimento específico.

```
1     def mgf1(seed: int, length: int, hash_func=hashlib.sha1) ->
      bytes: #mascara de geracao de funcao
2         if length > (hash_func().digest_size * (2**32)): #tamanho
      da mascara
3             raise ValueError("mask too long")
4         # Converte o tamanho de bits para bytes
5         byte_length = (length + 7) // 8
6         # Converte a seed para um tamanho fixo de bytes
7         seed_bytes = seed.to_bytes((seed.bit_length() + 7) // 8,
      byteorder='big')
8         T = b""
9         counter = 0
10        while len(T) < byte_length: # enquanto o tamanho de T for
      menor que o tamanho de bytes
11            C = counter.to_bytes(4, byteorder='big')
12            T += hash_func(seed_bytes + C).digest()
13            counter += 1
14        return T[:byte_length]
15
```

Não criamos o código acima retiramos de uma implementaçaõ encontrada na web. O link estará disponibilizado nas referências.

5.2 Funções básicas

`Shift_left` e `Shift_right` realizam o deslocamento dos bits para a esquerda e direita, respectivamente. `MSB` calcula o bit mais significativo de um número. `Number_of_bits` calcula o número de bits de um número.

```
    def shift_left(shift_values, n): #deslocamento para a esquerda
    """
    Shift Left
    Args:
        shift_values: quantidade de shift
        n: numero para ser shiftado
    Returns:
        int: numero shiftado
    """
        return n << shift_values

    def shift_right(shift_values, n): #deslocamento para a direita
    """
    Shift Right
    Args:
        shift_values: quantidade de shift
        n: numero para ser shiftado
    Returns:
        int: numero shiftado
    """
        return n >> shift_values

    def msb(n): #bit mais significativo
    """
    Most Significant Bit
    Args:
        n: numero para ser calculado o MSB
    Returns:
        int: MSB
    """
        msb = 0
        while n > 0:
            n >>= 1
            msb += 1
        return msb

    def number_of_bits(n): #calcula o numero de bits de um certo
    numero
    """
    Number of Bits
    Args:
        n: numero para ser calculado a quantidade de bits
    Returns:
        int: quantidade de bits
    """
        temp = n
        count = 0
        while(temp != 0):
            temp = shift_right(1, temp)
            count += 1
        return count
```

5.3 Função `DB`

Concatena um hash e uma mensagem, adicionando um preenchimento de no minimo 8 bits.

```
1     def DB(pHash, mensagem): #faz a concatenacao de pHash e
      mensagem
2     """
3     DB
4     Obs:
5         padding minimo de 8 bits
6     Args:
7         pHash: pHash
8         mensagem: mensagem
9     Returns:
10        int: DB com padding
11    """
12        pHash = shift_left(1536 , pHash)
13        padding = shift_left(number_of_bits(mensagem), 1)
14        pHash = pHash | padding
15        return (pHash | mensagem)
16
```

A função `DB` é responsável por concatenar um hash e uma mensagem, adicionando um preenchimento de no mínimo 8 bits, padding. O preenchimento é necessário para garantir que a mensagem tenha um comprimento adequado para o processo de criptografia. A função realiza as seguintes etapas:

1. Calcula o hash de um parámetro pré-definido (neste caso, `pHash(NULL)`).
2. Desloca o hash para a esquerda em 1536 bits. Isso é necessário, pois o pHash deve ficar encaixado nos 256 bits mais significativos do DB.
3. Calcula o padding necessário para a mensagem, setando em 1 o bit à esquerda do MSB da mensagem.
4. Realiza a operação lógica OR com o pHash e o padding.
5. Realiza a operação lógica OR com o resultado anterior e a mensagem.

A função retorna o valor concatenado de 1792 bits, que será utilizado nas etapas subsequentes do processo de encapsulamento OAEP.

5.4 Função `enc_oaep`

Implementa a etapa de encapsulamento OAEP para preparacão da mensagem antes da criptografia RSA.

```
1     def enc_oaep(mensagem): #encriptacao OAEP
2     """
3     Encapsulamento OAEP
4     Args:
5         mensagem: mensagem
6     Returns:
7         int: EM
8     """
9         seed = random.getrandbits(256).to_bytes(32, byteorder="big"
      )                                 # gera uma seed aleatoria
10        seed = int.from_bytes(seed, byteorder='big')
                                        # converte a seed para int
11        pHash = hashlib.sha3_256().digest()
                                        # hash de NULL
12        pHash = int.from_bytes(pHash, byteorder='big')
                                        # converte o hash da seed para
       int
13        maskedDB = DB(pHash, mensagem) ^ int.from_bytes(mgf1(seed,
      1792, hashlib.sha3_256))          # faz a concatenacao de pHash
      e mensagem
14        maskedSeed = seed ^ int.from_bytes(mgf1(maskedDB, 32,
      hashlib.sha3_256))                    # faz a mascara da seed
15        maskedSeed = shift_left(1792, maskedSeed)
                                        # desloca a seed para a
      esquerda em 1792 bits
16        EM = maskedSeed | maskedDB
                                        # concatena a mascara da seed
      e a mascara da mensagem
17        return EM
18
```

1. Geração da Seed: Uma seed aleatória de 256 bits é gerada e convertida para um inteiro.
2. pHash de NULL: Um hash SHA-3 de 256 bits de NULL é gerado e convertido para um inteiro.
3. maskedDB: A função `DB` concatena a `seed` com o `pHash` e realiza um (XOR) com a máscara gerada pela função `mgf1` na seed.
4. maskedSeed: A MaskedSeed é criada pela seed `XOR` com a máscara gerada pela função `mgf1` no `maskedDB`.
5. Deslocamento da maskedSeed: A maskedSeed é deslocada para a esquerda em 1792 bits.
6. Concatenação Final: A maskedSeed é concatenada com um `OR` com o `maskedDB`.

5.5 Função `dec_oaep`

Realiza a decodificacão OAEP para recuperar a mensagem original após a descriptografia RSA.

```
    def dec_oaep(c): #decriptacao OAEP
    """
    Decriptando OAEP
    Args:
        c: EM
    Returns:
        int: m
    """
        bitMask = shift_left(1792, 1) - 1
                                        # mascara de bits
        maskedDB = c & bitMask
        maskedSeed = shift_right(1792, c)
                                                # mascara de DB
        seed = maskedSeed ^ int.from_bytes(mgf1(maskedDB, 32,
    hashlib.sha3_256), byteorder='big')   #
        db = maskedDB ^ int.from_bytes(mgf1(seed, 1792, hashlib.
    sha3_256), byteorder='big')          #
        bitMask = shift_left(1536, 1) - 1
                                        #
        m = db & bitMask
                                        #
        MSBit = msb(m) - 1
                                        #
        MSBit = shift_left(MSBit, 1)
                                        #
        m = m ^ MSBit
                                        #
        return m

```

1. Geração da Máscara de Bits: Uma máscara de bits é criada para isolar os últimos 1792 bits de c. A função `shift_left(1792, 1)` desloca o número 1 para a esquerda em 1792 bits, resultando em um número com 1792 zeros seguidos de um 1. Subtraindo 1, obtemos uma máscara com 1792 uns.

2. Extraindo `maskedDB`: A operação & (AND bit a bit) é usada para extrair os últimos 1792 bits de c, que correspondem ao `maskedDB`.

3. Extraindo `maskedSeed`: A função `shift_right(1792, c)` desloca c para a direita em 1792 bits, isolando assim a `maskedSeed`.

4. Recuperando a Seed: A seed original é recuperada aplicando uma operação XOR ($\wedge$) entre a `maskedSeed` e uma máscara gerada pela função `mgf1` aplicada ao `maskedDB`.

5. Recuperando `db`: O `db` original é recuperado aplicando uma operação XOR entre o `maskedDB` e uma máscara gerada pela função `mgf1` aplicada à seed.

6. Máscara de Bits para `m`: Uma nova máscara de bits é criada para isolar os últimos 1536 bits de `db`.

7. Extraindo a Mensagem `m`: A mensagem original m é extraída aplicando a operação & (AND bit a bit) entre `db` e a máscara de bits.

8. Ajustando o Bit Mais Significativo (MSB): O bit mais significativo (MSB) da mensagem é ajustado para garantir que a mensagem seja corretamente decodificada. Isso é feito através de operações de deslocamento e XOR.

6 RSA e Base64

6.1 Função `multiplicative_inverse`

Calcula o inverso multiplicativo de um número utilizando o algoritmo estendido de Euclides.

```
1     def multiplicative_inverse(a, b): # calcula o inverso
      multiplicativo usando o Algoritmo extendido de Euclides
2     """
3     Inverso multiplicativo
4     Obs
5         a > b
6         a x t1 e congruente 1 mod b
7     Args:
8         a: numero
9         b: numero
10    Returns:
11        int: inverso multiplicativo
12    """
13        if b > a:
14            a, b = b, a
15        old_a = a
16        if gcd(a, b) != 1:
17            return None
18        q = a // b
19        r = a % b
20        t1 = 0
21        t2 = 1
22        t = t1 - t2 * q
23        while b != 0:
24            a, b = b, r
25            if b == 0:
26                t1 = t2
27                if t1 < 0:
28                    t1 += old_a
29                return t1
30            q = a // b
31            r = a % b
32            t1, t2 = t2, t
33            t = t1 - t2 * q
34
```

6.2 Funções `base64_encode` e `base64_decode`

Implementam codificacão e decodificacão Base64.

```
    def base64_encode(mensagem): #codificacao usando a base64
    """
    Base64 Encode
    Args:
        mensagem: mensagem
    Returns:
        bytes: mensagem codificada
    """
        mensagem = mensagem.encode('utf-8')
        mensagem = base64.b64encode(mensagem)
        return mensagem

    def base64_decode(mensagem_encriptada): #decodificacao usando a
     base64
    """
    Base64 Decode
    Args:
        mensagem_encriptada: int mensagem encriptada
    Returns:
        str: mensagem decodificada
    """
        mensagem_encriptada = mensagem_encriptada.to_bytes((
    mensagem_encriptada.bit_length() + 7) // 8, byteorder='big')
        mensagem_encriptada = base64.b64decode(mensagem_encriptada)
    .decode('utf-8')
        return mensagem_encriptada

```

6.3 Funções `enc_rsa` e `dec_rsa`

Realizam a criptografia e descriptografia RSA.

```
    def enc_rsa(n, e, m): #encriptacao RSA
    """
    RSA Encryption
    Args:
        n: produto dos primos n
        e: chave public e
        m: mensagem
    Returns:
        int: c
        int: n
    """
        c = pow(m, e, n)
        return c, n

    def dec_rsa(p, q, e, c): #decriptacao RSA
    """
    RSA Decryption
    Args:
        p: primo p
        q: primo q
        e: chave publica e
        c: mensagem criptografada
    Returns:
        int: mensagem descriptografada m
    """
        n = p * q
        phi = (p - 1) * (q - 1)
        d = multiplicative_inverse(phi, e)
        m = pow(c, d, n)
        return m

```

6.4 Funções `assinatura_com_rsa` e `verificar_assinatura_com_rsa`

Geram uma assinatura digital usando RSA e verificam sua autenticidade.

```
    def assinatura_com_rsa(message, key, n): #assinatura com RSA
        message = base64_encode(message)
        message = int.from_bytes(message, byteorder='big')
        hash_message = hashlib.sha3_256(message.to_bytes(32,
    byteorder='big')).digest()
        hash_message = int.from_bytes(hash_message, byteorder='big'
    )
        encrypted_hash = enc_rsa(n, key, hash_message)[0]
        return message, encrypted_hash

    def verificar_assinatura_com_rsa(message, enc_hash, e, n):
        decrypted_hash = enc_rsa(n, e, enc_hash)
        hash_message = hashlib.sha3_256(message.to_bytes(32,
    byteorder='big')).digest()
        hash_message = int.from_bytes(hash_message, byteorder='big'
    )
        return decrypted_hash[0] == hash_message
```

```

```

6.5 Código de teste

O código de teste gera dois números primos, calcula e , codifica uma mensagem em Base64, realiza a criptografia RSA, descriptografa a mensagem, gera e verifica uma assinatura digital.

```
    # Teste de execucao do programa

lista = prime_numbers()
p = lista[0]
q = lista[1]
n = p * q
phi = (p - 1) * (q - 1)
e = 65537
message = "Bluey Heeler"

print("Texto original:", message)
print("=========================================================")
message = base64_encode(message)
print("Message com BASE64:", message)
print("=========================================================")
message = enc_oaep(int.from_bytes(message, byteorder='big'))
print("Message com OAEP:", message)
print("=========================================================")
enc_message = enc_rsa(p * q, e, message)[0]
print("Message encriptada com RSA:", enc_message)
print("=========================================================")
dec_message = dec_rsa(p, q, e, enc_message)
print("Message descriptografada com RSA:", dec_message)
print("=========================================================")
dec_message = dec_oaep(dec_message)
print("Message descriptografada com OAEP:", dec_message)
print("=========================================================")
dec_message = base64_decode(dec_message)
print("Message descriptografada com BASE64:", dec_message)
print("=========================================================")
d = multiplicative_inverse(e, phi)          #Private key
message = "Bluey Heeler"
c = (assinatura_com_rsa(message, d, n))
print("Mensagem: ", c[0], "Assinatura: ", c[1])
print("=========================================================")
print(verificar_assinatura_com_rsa(c[0], c[1], e, n))
```

6.6 Impressão dos resultados

Por fim, o código imprime os resultados obtidos durante o teste.

```
Texto original: Bluey Heeler
============================================================
Message com BASE64: b'Qmx1ZXkgSGVlbGVy'
============================================================
Message com OAEP: 123934030549754492491210231517666468099730
369442567688928578558286419964863381939669809836534438834640
574310406666447300575050808586264352549897458907546763535965
984452608526562908810643416967711008306866672881682194684283
740557775781735790577949197826806275977650831281749354860437
523127708065751832847683773984717204032294259713046385753159
905749941944973553402221672293014068464668257642064375871658
774153288925901118853075942714803710368797697455635279938888
554817566393562761016967537375626053087500121574139942083547
872647694721457958554497259380122494023896678864888018821502
070572652576557617081555911780555546
============================================================
Message encriptada com RSA: 150560028313053864293695780953555
508273616502034331236071775124786842864065033001042693931720
646944781588159691880277391553798572617769934379623251912328
846197149916658589532294448631519904152853405610787998586710
899393634545821672318441986195833808438244444473133093529842
908609702557069293870537639358475093499886731555097184848691
033170877550709922480953798163720998271677316482433149367349
280445062421288233406709124016609068591255854451085685936615
294531036455966095834693157492962824461428369243671654428706
376324900791509868420949038355712002219951420371594635475017
482891061696236046241908471012372146994055492
============================================================
Message descriptografada com RSA: 123934030549754492491210231
517666468099730369442567688928578558286419964863381939669809
836534438834640574310406666447300575050808586264352549897458
907546763535965984452608526562908810643416967711008306866672
288168219468428374055777578173579057794919782680627597765083
128174935486043752312770806575183284768377398471720403229425
971304638575315990574994194497355340222167229301406846466825
764206437587165877415328892590111885307594271480371036879769
745563527993888855481756639356276101696753737562605308750012
157413994208354787264769472145795855449725938012249402389667
886488801882150207057265257655761708155591178055546
============================================================
Message descriptografada com OAEP: 108235865815451426871397
21985126684281
============================================================
Message descriptografada com BASE64: Bluey Heeler
============================================================
Mensagem:  108235865815451426871397921985126684281
Assinatura:  728694378666921043938648876085590937341527573594866616645083096396778552077440714538114677783048794813411106662660503528912736529339480593131057878936512377639978300115477854921121687808970267976392688488330426889052886907463852153168136819800362355204085495211101774442033600117732415113904087201078885025370667727196636373187563728173386641832487702598810874893689161679756039555302004211992535248386443307293499932184742860376681042555413699142511827445368738364985214014548490393474548863981140651156333027703207590893293500649663306627437620811791115980110073081809266403854733494531692738400067704588472958098 5
```

```
58 ============================================================
59 True
```

7 Conclusão

O trabalho de implementação do RSA somado ao OAEP e Base64 foi concluído com sucesso. O código foi testado e os resultados obtidos foram satisfatórios. E com isso acreditamos que melhoramos nosso entendimento sobre o funcionamento do algoritmo RSA e suas aplicações na criptografia moderna.

8 Referências

- https://en.wikipedia.org/wiki/Mask_generation_function
- https://www.youtube.com/watch?v=8i0UnX7Snkc&t=502s
- https://www.youtube.com/watch?v=YwaQ4m1eHQo&t=499s
- https://www.inf.pucrs.br/calazans/graduate/TPVLSI_I/RSA-oaep_spec.pdf
- Slides da disciplina de segurança Computacional UnB

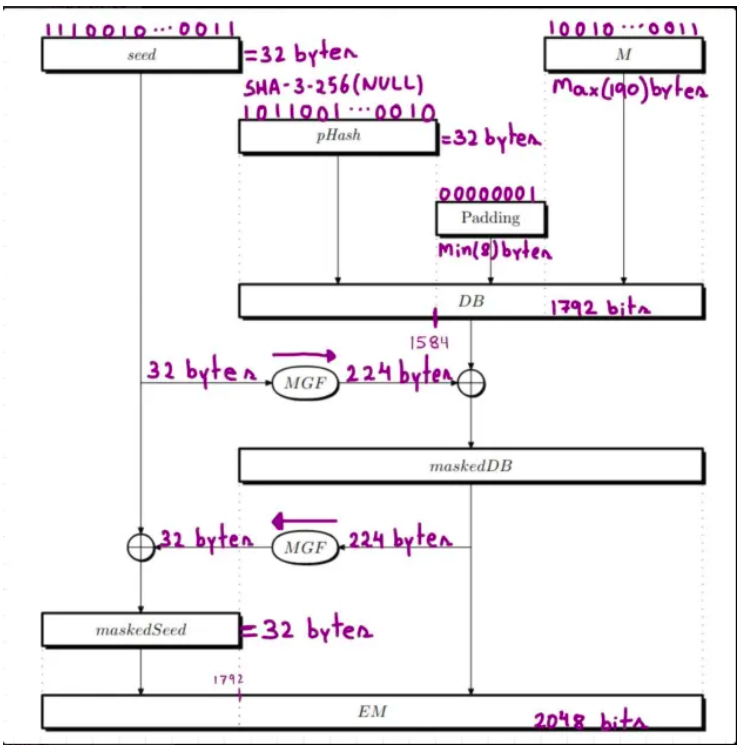

Figura 1: Diagrama do processo de encapsulamento OAEP

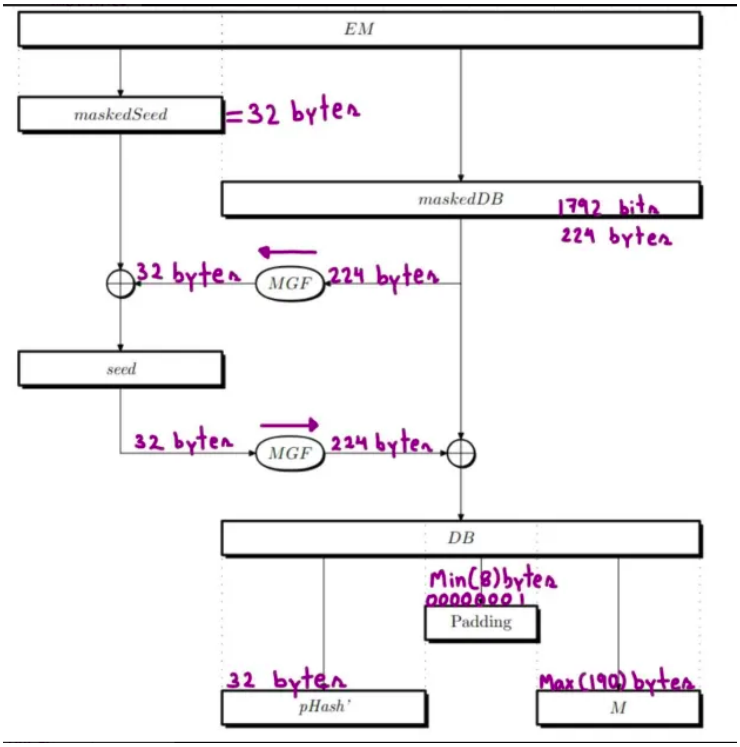

Figura 2: Diagrama do processo de decapsulamento OAEP