

# Explicação 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), codificação 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 descryptografia 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 função decompõe um número no seguinte formato. Ela será utilizada em uma das etapas do algoritmo de Miller Rabin.

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

```
1 def decompondo(n):
2     # Decompoe n-1 como 2 elevado a e vezes m, onde n e impar.
3     # Args:
4     #     n: numero para decompor
5     # Returns:
6     #     int: expoente
7     #     int: numero
8     e = 0
9     m = n - 1
10    while m % 2 == 0:
11        m //= 2
12        e += 1
13    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.

```
1  def miller_rabin(n, rodadas=10):
2      """
3      Miller-Rabin primality test
4      Args:
5          n: numero para ser testado
6          rodadas: quantidade de rodadas a serem feitas
7      Returns:
8          bool: True se e primo, False se nao e primo
9      """
10     if n <= 1 or n == 4:
11         return False
12     if n <= 3:
13         return True
14     if n % 2 == 0:
15         return False
16
17     expo_k, m = decompondo(n) #exponente K e m tal que n - 1 =
2^k * m
18     for _ in range(rodadas):
19         a = random.randrange(2, n - 2) # pega um numero
20         aleatorio entre 2 e n-2
21         """ x = a^m mod n """
22         x = pow(a, m, n)
23         if x == 1 or x == n-1: # se x for 1 ou n-1, entao n e
24             primo
25             continue
26         else:
27             teste = False
28             for _ in range(expo_k - 1): # para i de 0 ate k-1
29                 """ x = x^2 mod n """
30                 x = pow(x, 2, n)
31                 if x == n-1:
32                     teste = True
33                     break
34             if(teste):
35                 continue
36             else:
37                 return False
38     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. **Decompõe:** 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  $x = a^m \pmod{n}$ .
4. **Verificação Inicial:** Se  $x = 1$  ou  $x = n - 1$ , então  $n$  pode ser primo.
5. **Iterações:** Para  $k - 1$  iterações, calcula  $x = x^2 \pmod{n}$ . 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.

```

1  def prime_numbers(): #descobre os numeros primos usando a
    funcao de Miller-Rabin
2  """
3  Descobrindo os numeros primos
4  Args: none
5  Returns:
6      list: lista com os numeros primos
7  """
8      n = 1 << 1024
9      contador = 0
10     par_de_numeros_primos = []
11     while True:
12         if miller_rabin(n):
13             par_de_numeros_primos.append(n)
14             contador += 1
15             if contador == 2:
16                 break
17             n += 1
18     return par_de_numeros_primos
19 
```

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 função de hash para gerar uma saída 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 implementação 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.

```

1  def shift_left(shift_values, n): #deslocamento para a esquerda
2  """
3  Shift Left
4  Args:
5      shift_values: quantidade de shift
6      n: numero para ser shiftado
7  Returns:
8      int: numero shiftado
9  """
10     return n << shift_values
11
12  def shift_right(shift_values, n): #deslocamento para a direita
13  """
14  Shift Right
15  Args:
16      shift_values: quantidade de shift
17      n: numero para ser shiftado
18  Returns:
19      int: numero shiftado
20  """
21     return n >> shift_values
22
23  def msb(n): #bit mais significativo
24  """
25  Most Significant Bit
26  Args:
27      n: numero para ser calculado o MSB
28  Returns:
29      int: MSB
30  """
31     msb = 0
32     while n > 0:
33         n >>= 1
34         msb += 1
35     return msb
36
37  def number_of_bits(n): #calcula o numero de bits de um certo
38  numero
39  """
40  Number of Bits
41  Args:
42      n: numero para ser calculado a quantidade de bits
43  Returns:
44      int: quantidade de bits
45  """
46     temp = n
47     count = 0
48     while(temp != 0):
49         temp = shift_right(1, temp)
50         count += 1
51     return count
52
53

```

### 5.3 Função DB

Concatena um hash e uma mensagem, adicionando um preenchimento de no mínimo 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 parâmetro 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 preparação 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"
10     )
11     seed = int.from_bytes(seed, byteorder='big')
12     pHash = hashlib.sha3_256().digest()
13     pHash = int.from_bytes(pHash, byteorder='big')
14     maskedDB = DB(pHash, mensagem) ^ int.from_bytes(mgf1(seed,
15     1792, hashlib.sha3_256))
16     maskedSeed = seed ^ int.from_bytes(mgf1(maskedDB, 32,
17     hashlib.sha3_256))
18     maskedSeed = shift_left(1792, maskedSeed)
19     EM = maskedSeed | maskedDB
20     return EM
```

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 decodificação OAEP para recuperar a mensagem original após a descriptografia RSA.

```
1  def dec_oaep(c): #decriptacao OAEP
2  """
3  Decriptando OAEP
4  Args:
5  c: EM
6  Returns:
7  int: m
8  """
9
10     bitMask = shift_left(1792, 1) - 1
11                                     # mascara de bits
12
13     maskedDB = c & bitMask
14     maskedSeed = shift_right(1792, c)
15                                     # mascara de DB
16
17     seed = maskedSeed ^ int.from_bytes(mgf1(maskedDB, 32,
18     hashlib.sha3_256), byteorder='big') #
19     db = maskedDB ^ int.from_bytes(mgf1(seed, 1792, hashlib.
20     sha3_256), byteorder='big') #
21
22     bitMask = shift_left(1536, 1) - 1
23                                     #
24
25     m = db & bitMask
26                                     #
27
28     MSBit = msb(m) - 1
29                                     #
30
31     MSBit = shift_left(MSBit, 1)
32                                     #
33
34     m = m ^ MSBit
35                                     #
36
37     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. Extraíndo `maskedDB`: A operação `&` (AND bit a bit) é usada para extrair os últimos 1792 bits de  $c$ , que correspondem ao `maskedDB`.
3. Extraíndo `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. Extraíndo 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
2  multiplicativo usando o Algoritmo estendido de Euclides
3  """
4  Inverso multiplicativo
5  Obs
6      a > b
7      a x t1 e congruente 1 mod b
8  Args:
9      a: numero
10     b: numero
11 Returns:
12     int: inverso multiplicativo
13 """
14     if b > a:
15         a, b = b, a
16     old_a = a
17     if gcd(a, b) != 1:
18         return None
19     q = a // b
20     r = a % b
21     t1 = 0
22     t2 = 1
23     t = t1 - t2 * q
24     while b != 0:
25         a, b = b, r
26         if b == 0:
27             t1 = t2
28             if t1 < 0:
29                 t1 += old_a
30             return t1
31         q = a // b
32         r = a % b
33         t1, t2 = t2, t
34         t = t1 - t2 * q

```

## 6.2 Funções base64\_encode e base64\_decode

Implementam codificação e decodificação Base64.

```
1  def base64_encode(mensagem): #codificacao usando a base64
2  """
3  Base64 Encode
4  Args:
5      mensagem: mensagem
6  Returns:
7      bytes: mensagem codificada
8  """
9      mensagem = mensagem.encode('utf-8')
10     mensagem = base64.b64encode(mensagem)
11     return mensagem
12
13 def base64_decode(mensagem_encriptada): #decodificacao usando a
14     base64
15     """
16     Base64 Decode
17     Args:
18         mensagem_encriptada: int mensagem encriptada
19     Returns:
20         str: mensagem decodificada
21     """
22     mensagem_encriptada = mensagem_encriptada.to_bytes((
23     mensagem_encriptada.bit_length() + 7) // 8, byteorder='big')
24     mensagem_encriptada = base64.b64decode(mensagem_encriptada)
25     .decode('utf-8')
26     return mensagem_encriptada
```

### 6.3 Funções enc\_rsa e dec\_rsa

Realizam a criptografia e descriptografia RSA.

```
1  def enc_rsa(n, e, m): #encriptacao RSA
2  """
3  RSA Encryption
4  Args:
5      n: produto dos primos n
6      e: chave public e
7      m: mensagem
8  Returns:
9      int: c
10     int: n
11 """
12     c = pow(m, e, n)
13     return c, n
14
15 def dec_rsa(p, q, e, c): #decriptacao RSA
16 """
17 RSA Decryption
18 Args:
19     p: primo p
20     q: primo q
21     e: chave publica e
22     c: mensagem criptografada
23 Returns:
24     int: mensagem descriptografada m
25 """
26     n = p * q
27     phi = (p - 1) * (q - 1)
28     d = multiplicative_inverse(phi, e)
29     m = pow(c, d, n)
30     return m
31
```

### 6.4 Funções assinatura\_com\_rsa e verificar\_assinatura\_com\_rsa

Geram uma assinatura digital usando RSA e verificam sua autenticidade.

```
1  def assinatura_com_rsa(message, key, n): #assinatura com RSA
2      message = base64_encode(message)
3      message = int.from_bytes(message, byteorder='big')
4      hash_message = hashlib.sha3_256(message.to_bytes(32,
5      byteorder='big')).digest()
6      hash_message = int.from_bytes(hash_message, byteorder='big')
7      )
8      encrypted_hash = enc_rsa(n, key, hash_message)[0]
9      return message, encrypted_hash
10
11 def verificar_assinatura_com_rsa(message, enc_hash, e, n):
12     decrypted_hash = enc_rsa(n, e, enc_hash)
13     hash_message = hashlib.sha3_256(message.to_bytes(32,
14     byteorder='big')).digest()
15     hash_message = int.from_bytes(hash_message, byteorder='big')
16     )
17     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.

```

1      # Teste de execucao do programa
2
3      lista = prime_numbers()
4      p = lista[0]
5      q = lista[1]
6      n = p * q
7      phi = (p - 1) * (q - 1)
8      e = 65537
9      message = "Bluey Heeler"
10
11     print("Texto original:", message)
12     print("=====")
13     message = base64_encode(message)
14     print("Message com BASE64:", message)
15     print("=====")
16     message = enc_oaep(int.from_bytes(message, byteorder='big'))
17     print("Message com OAEP:", message)
18     print("=====")
19     enc_message = enc_rsa(p * q, e, message)[0]
20     print("Message encriptada com RSA:", enc_message)
21     print("=====")
22     dec_message = dec_rsa(p, q, e, enc_message)
23     print("Message descriptografada com RSA:", dec_message)
24     print("=====")
25     dec_message = dec_oaep(dec_message)
26     print("Message descriptografada com OAEP:", dec_message)
27     print("=====")
28     dec_message = base64_decode(dec_message)
29     print("Message descriptografada com BASE64:", dec_message)
30     print("=====")
31     d = multiplicative_inverse(e, phi)          #Private key
32     message = "Bluey Heeler"
33     c = (assinatura_com_rsa(message, d, n))
34     print("Mensagem: ", c[0], "Assinatura: ", c[1])
35     print("=====")
36     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.

```

1 Texto original: Bluey Heeler
2 =====
3 Message com BASE64: b'Qmx1ZXkgSGVlbGVy'
4 =====
5 Message com OAEP: 123934030549754492491210231517666468099730
6 369442567688928578558286419964863381939669809836534438834640
7 574310406666447300575050808586264352549897458907546763535965
8 984452608526562908810643416967711008306866672881682194684283
9 740557775781735790577949197826806275977650831281749354860437
10 523127708065751832847683773984717204032294259713046385753159
11 905749941944973553402221672293014068464668257642064375871658
12 774153288925901118853075942714803710368797697455635279938888
13 554817566393562761016967537375626053087500121574139942083547
14 872647694721457958554497259380122494023896678864888018821502
15 07057265257655761708155591178055546
16 =====
17 Message encriptada com RSA: 15056002831305386429369578095355
18 508273616502034331236071775124786842864065033001042693931720
19 646944781588159691880277391553798572617769934379623251912328
20 846197149916658589532294448631519904152853405610787998586710
21 89939363454582167231844198619583380843824444473133093529842
22 908609702557069293870537639358475093499886731555097184848691
23 033170877550709922480953798163720998271677316482433149367349
24 280445062421288233406709124016609068591255854451085685936615
25 294531036455966095834693157492962824461428369243671654428706
26 376324900791509868420949038355712002219951420371594635475017
27 482891061696236046241908471012372146994055492
28 =====
29 Message descriptografada com RSA: 12393403054975449249121023
30 151766646809973036944256768892857855828641996486338193966980
31 983653443883464057431040666644730057505080858626435254989745
32 890754676353596598445260852656290881064341696771100830686667
33 288168219468428374055777578173579057794919782680627597765083
34 128174935486043752312770806575183284768377398471720403229425
35 971304638575315990574994194497355340222167229301406846466825
36 764206437587165877415328892590111885307594271480371036879769
37 745563527993888855481756639356276101696753737562605308750012
38 157413994208354787264769472145795855449725938012249402389667
39 886488801882150207057265257655761708155591178055546
40 =====
41 Message descriptografada com OAEP: 1082358658154514268713979
42 21985126684281
43 =====
44 Message descriptografada com BASE64: Bluey Heeler
45 =====
46 Mensagem: 108235865815451426871397921985126684281
47 Assinatura: 72869437866692104393864887608559093734152757359
48 486661664508309639677855207744071453811467778304879481341110
49 666266050352891273652933948059313105787893651237763997830011
50 547785492112168780897026797639268848833042688905288690746385
51 215316813681980036235520408549521110177444203360011773241511
52 390408720107888502537066772719663637318756372817338664183248
53 77025988108748936891616797560395530200421199253524838644330
54 729349993218474286037668104255541369914251182744536873836498
55 521401454849039347454886398114065115633302770320759089329350
56 064966330662743762081179111598011007308180926640385473349453
57 16927384000677045884729580985

```

```
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://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](https://www.inf.pucrs.br/calazans/graduate/TPVLSI_I/RSA-oaep_spec.pdf)
- Slides da disciplina de segurança Computacional UnB

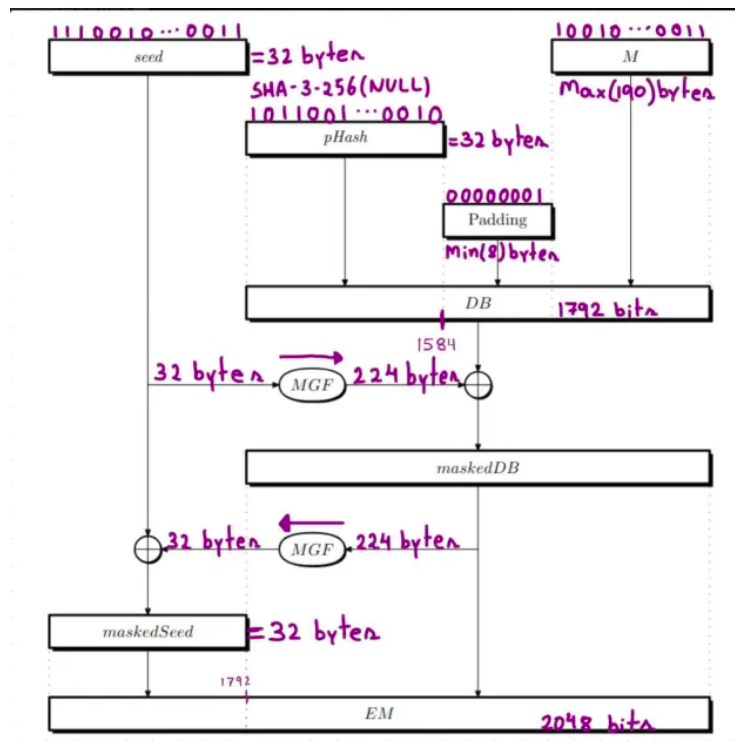


Figura 1: Diagrama do processo de encapsulamento OAEP



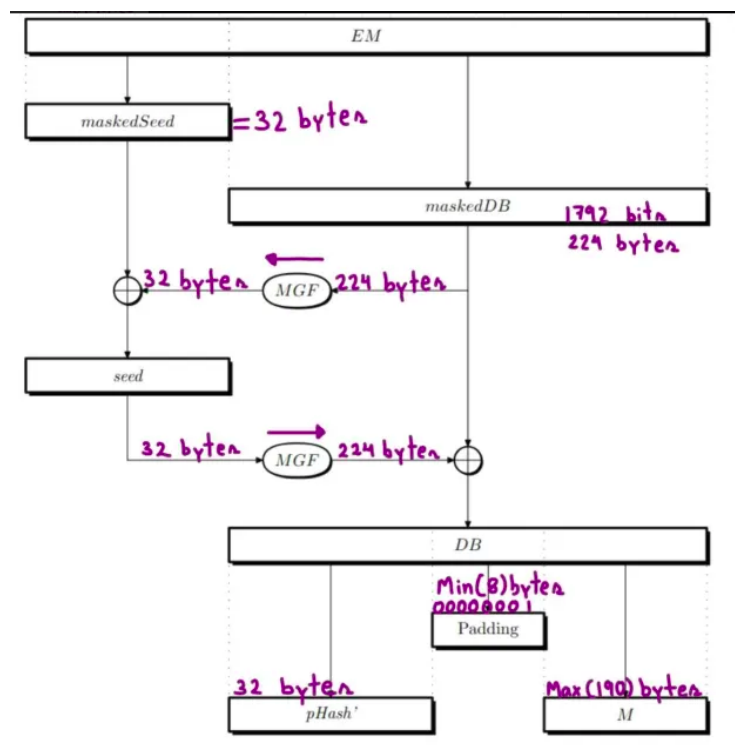


Figura 2: Diagrama do processo de decapsulamento OAEP